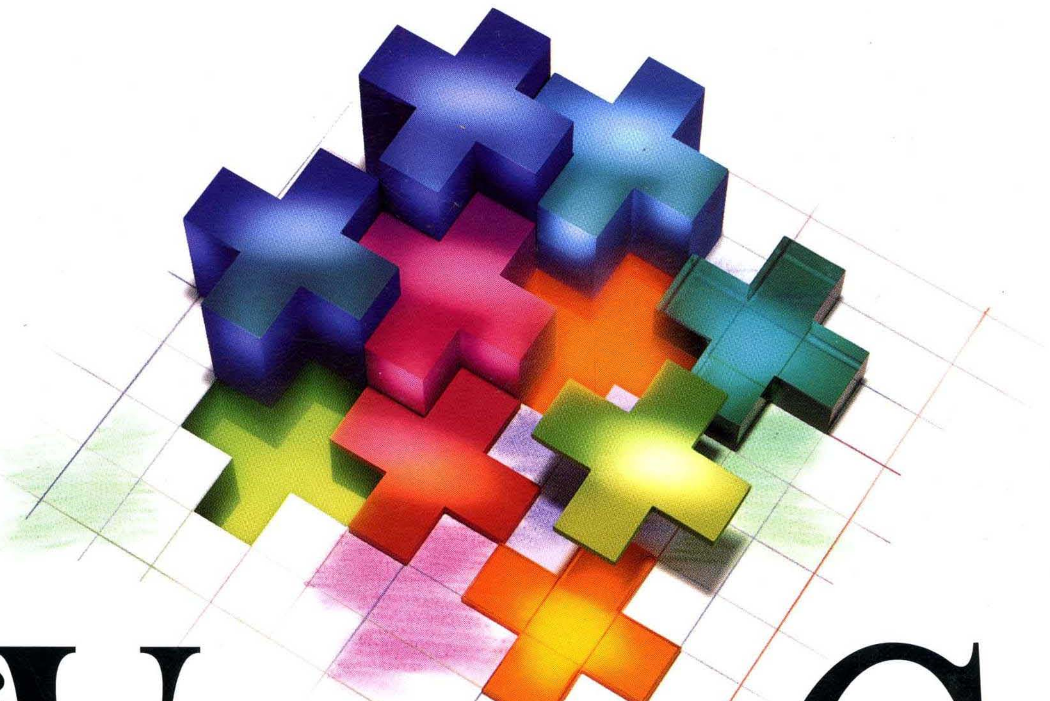
Reference Volume I

Class Library Reference

For the Microsoft Foundation Class Library



Microsoft® Visual C++™

Development System for Windows™

Reference Volume I
Class Library Reference

For the Microsoft® Foundation Class Library

Microsoft® Visual C++™

Development System for Windows™
Version 1.0

Microsoft Corporation

Document No. DB35743-0193
Printed in the United States of America.

Contents

Part 2 The Microsoft Foundation Class Library Reference

Figures and Tables

Figures

Tables

Introduction

The *Class Library Reference* covers the classes, global functions, global variables, and macros that make up the Microsoft® Foundation Class Library version 2.0, which is included with Microsoft Visual C++™ Development System for Windows™ version 1.0. Figure I.1 at the end of this introduction is a class hierarchy chart that details the class relationships in the class library. This book is divided into two parts:

Part 1 Introduction to the Microsoft Foundation Class Library

Part 2 The Microsoft Foundation Class Reference

Part 1 contains overview material designed to help you learn about and use the Microsoft Foundation Class Library. Chapter 1 lists the classes in helpful categories. Use these lists to help locate a class that contains the functionality you are interested in. Chapters 2 through 6 describe the Microsoft Foundation Class Library and the "application framework" that it provides to help you program for the Microsoft Windows™ operating system. Use these chapters to learn how the framework operates and how your code fits into the framework. Practical examples and techniques are provided in the *Class Library User's Guide*.

Material applicable to programs for MS-DOS® as well as to programs for Windows is covered in Chapter 6. This includes diagnostics, file handling, exception handling, and collection classes.

Part 2 contains the following components:

- An alphabetical listing of the classes
- A section that explains the global functions, global variables, and macros used with the class library

The hierarchy chart and the subset charts included with each class are useful for locating base classes. Be aware that the class documentation does not include repeated descriptions of inherited member functions, inherited operators, and overridden virtual member functions. You must always refer to the base classes depicted in the hierarchy diagrams.

In the alphabetical listing, each class description includes a member summary by category followed by alphabetical listings of:

- Member functions (public, protected, and private intermixed)
- Overloaded operators
- Data members

Public and protected class members are documented only when they are normally used in application programs or derived classes. Occasionally, private members are listed because they override a public or protected member in the base class. See the class header files for a complete listing of class members.

Many member functions of the Microsoft Foundation classes encapsulate calls to Windows API functions that are specific to Microsoft Windows version 3.1. These functions (and other material) are marked as "Windows 3.1 Only" in the alphabetical reference. To clearly distinguish Windows 3.1–specific material, each such section begins with the heading "Windows 3.1 Only" and ends with a diamond icon (♦).

Some C-language structures defined by Windows are so widely applicable that their descriptions have been reproduced completely in pertinent places in the alphabetical reference. Similarly, styles, such as window styles, are listed in appropriate places in the alphabetical reference.

In Part 2, please note that the "See Also" sections refer to Windows functions by prefacing them with the scope resolution operator (::). For example, **::EqualRect**. More information on these functions can be found in the *Windows Programmer's Reference*, other Windows references, and Help.

The "Macros and Globals" section at the end of the alphabetical class reference details the global functions, global variables, and macros supplied with the Microsoft Foundation Class Library. The section lists data types used with the class library, diagnostic and exception-handling services available, and message-map information. Macros, global functions, and global variables are listed alphabetically. See the beginning of the "Macros and Globals" section for a list of the topics covered.

Document Conventions

This book uses the following typographic conventions:

Examples	Description
STDIO.H	Uppercase letters indicate filenames, segment names, registers, and terms used at the operating-system command level.

char, **CObject**, **GetTime**, **TRACE**, **MF_STRING**, **CREATESTRUCT**, **__far**	Bold type indicates C and C++ keywords, operators, language-specific characters, and library routines. This includes the classes and member functions of the Microsoft Foundation Class Library, macros, flags, data structures and their members, and enumerators. Within discussions of syntax, bold type indicates that the text must be entered exactly as shown.
	Many functions and constants begin with either a single or double underscore. These are part of the name and are mandatory. For example, to have the **__cplusplus** manifest constant be recognized by the compiler, you must enter the leading double underscore.
expression	Words in italics indicate placeholders for information you must supply, such as a filename. Italic type is also used occasionally for emphasis in the text.
[[*option*]]	Items inside double square brackets are optional.
#pragma pack {1 | 2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets ([[ ]]) surround the braces.
`#include <io.h>`, `MyObject`	This font is used for examples, user input, program output, and error messages in text.
CL [[*option*]]*file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.
`while( ) { . . . }`	A column or row of three dots tells you that part of an example program has been intentionally omitted.
CTRL+ENTER	Small capital letters are used to indicate the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second.
	The carriage-return key, sometimes marked as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
`"C string"`	Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form `"  "` and `'  '` rather than " " and ' '.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is usually spelled out.
♦	This symbol denotes the end of a section of "Windows 3.1 Only" material or a "Protected" or "Private" class member.

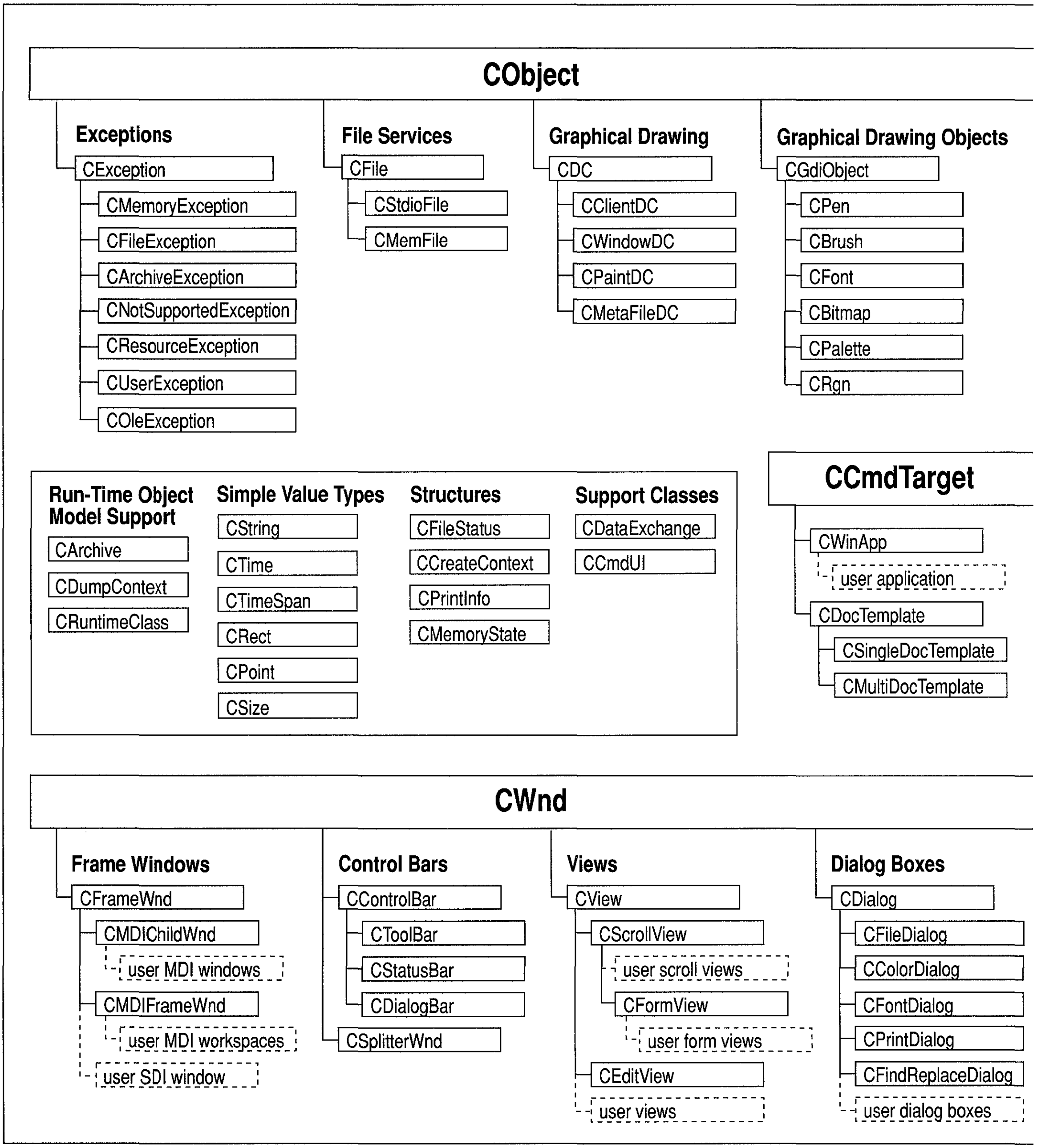
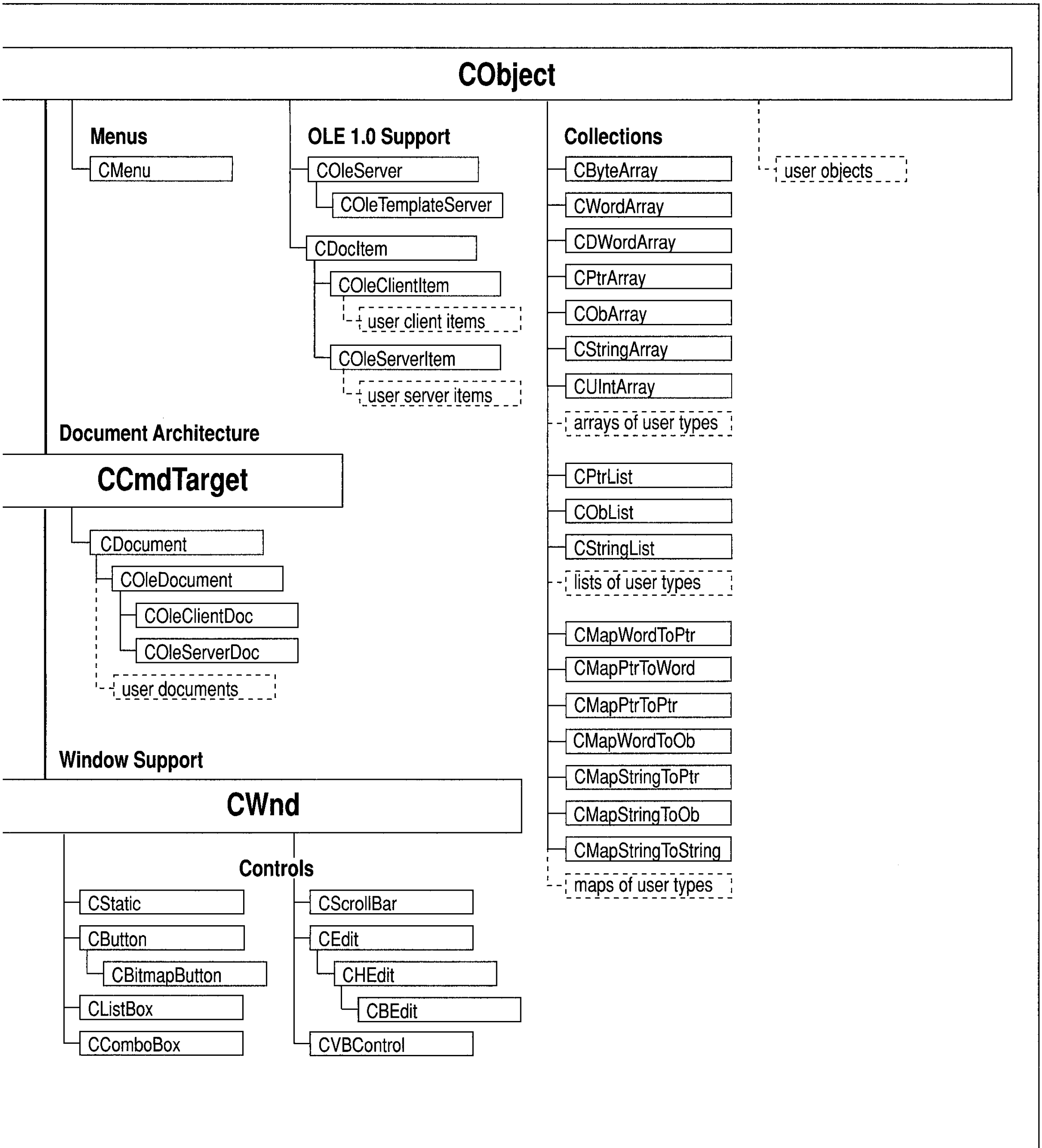


Figure I.1 Microsoft Foundation Class Library Hierarchy Chart


CObject
Menus
CMenu
OLE 1.0 Support
COleServer
COleTemplateServer
CDocItem
COleClientItem
user client items
COleServerItem
user server items
Collections
CByteArray
CWordArray
CDWordArray
CPtrArray
CObArray
CStringArray
CUIntArray
arrays of user types
CPtrList
CObList
CStringList
lists of user types
CMapWordToPtr
CMapPtrToWord
CMapPtrToPtr
CMapWordToOb
CMapStringToPtr
CMapStringToOb
CMapStringToString
maps of user types
user objects
Document Architecture
CCmdTarget
CDocument
COleDocument
COleClientDoc
COleServerDoc
user documents
Window Support
CWnd
Controls
CStatic
CButton
CBitmapButton
CListBox
CComboBox
CScrollBar
CEdit
CHEdit
CBEdit
CVBControl

PART 1

Introduction to the Microsoft Foundation Class Library

CHAPTER 1

The Microsoft Foundation Class Library

This chapter categorizes and describes the classes in the Microsoft Foundation Class Library version 2.0. These classes support application development for Microsoft Windows versions 3.0 and later.

Because the class library supports programming for Windows, its Windows classes are the largest and most important group of classes. Taken together, they constitute an "application framework"—the framework of an application written for Windows. Your programming task is to fill in the code that is unique to your application.

The library's classes are presented here in the following categories:

- Root Class
- Application Architecture Classes
  - Windows Application Class
  - Command-Related Classes
  - Document/View Classes
- Visual Object Classes
  - Window Classes
  - View Classes
  - Dialog Classes
  - Control Classes
  - Menu Class
  - Device-Context Classes
  - Drawing Object Classes
- General-Purpose Classes
  - File Classes
  - Diagnostics
  - Exceptions

  - Collections
  - Miscellaneous Support Classes
- Object Linking and Embedding (OLE) Classes
  - OLE Base Classes
  - OLE Client Classes
  - OLE Server Classes
  - OLE Exception Class
- Macros and Globals

The section "General Class Design Philosophy" at the end of this chapter explains how the Microsoft Foundation Class Library was designed.

The framework is explained in detail in Chapters 2 through 6.

Some of the classes listed above are general-purpose classes that can be used either with the framework or in MS-DOS programs. Chapter 6 details these classes, which provide useful abstractions such as collections, exceptions, files, and strings. The Object Linking and Embedding (OLE) classes support programming for OLE. See Chapter 18 in the *Class Library User's Guide* for more information about the OLE classes.

Class Summary

The following is a brief summary of the classes in the Microsoft Foundation Class Library, divided by category to help you locate what you need. In some cases, a class is listed in more than one category. To see a class's inheritance, use the class hierarchy diagram on page xvi.

Root Class

Most of the classes in the Microsoft Foundation Class Library are derived from a single base class at the root of the class hierarchy. **CObject** provides a number of useful capabilities to all classes derived from it, with very low overhead. For more information about **CObject** and its capabilities, see "CObject Services" on page 121 in Chapter 6.

CObject
The ultimate base class of nearly all other classes. Supports serializing data and obtaining run-time information about a class.

Application Architecture Classes

Classes in this category contribute to the architecture of a framework application. They supply functionality common to most applications written for Windows. You fill in the framework to add application-specific functionality. Typically, you do so by deriving new classes from the architecture classes, sometimes adding new members or overriding existing member functions.

The framework consists of a group of class objects that cooperate at run time to function as an application for Windows. The principal objects are:

- An application object derived from class **CWinApp**.
- One or more document objects derived from class **CDocument** and associated with a window.
- One or more view objects derived from class **CView**, each attached to a document and associated with a window.

Windows Application Class

Each application has one and only one application object; this object coordinates other objects in the running program and is derived from **CWinApp**.

CWinApp
Encapsulates the code to initialize, run, and terminate the application.

Command-Related Classes

As the user interacts with the application by choosing menus or control-bar buttons with the mouse, the application sends messages from the affected user-interface object to an appropriate command-target object, which is of class **CCmdTarget**. Command-target classes derived from **CCmdTarget** include **CWinApp**, **CWnd**, **CDocTemplate**, **CDocument**, **CView**, and the classes derived from them. Class **CCmdUI** represents a command user-interface object, such as a menu or button, for updating the object's state.

CCmdTarget
Serves as the base class for all classes of objects that can receive and respond to messages.

CCmdUI
Provides a programmatic interface for updating user-interface objects such as menu items or control-bar buttons. The command-target object enables, disables, checks, and/or unchecks the user-interface object via this proxy object.

Document/View Classes

Document objects, created by document template objects, manage the application's data. View objects, which represent the client area of a window, display a document's data and allow users to interact with it.

CDocTemplate
The base class for document templates. A document template coordinates the creation of document, view, and frame window objects.

CSingleDocTemplate
A template for documents in the single document interface (SDI). SDI applications have only one document open at a time.

CMultiDocTemplate
A template for documents in the multiple document interface (MDI). MDI applications can have multiple documents open at a time.

CDocument
The base class for application-specific documents. Derive your document class(es) from **CDocument**.

CView
The base class for application-specific views of a document's data. Views display data and take user input to edit or select the data. Derive your view class(es) from **CView**. See the description of **CView** and its derived classes under "View Classes."

CPrintInfo
A structure containing information about a print or print preview job. Used by **CView**'s printing architecture.

CCreateContext
A structure passed by a document template to window-creation functions to coordinate the creation of document, view, and frame window objects.

Visual Object Classes

Classes in this category represent visual user-interface objects: windows, dialog boxes, controls, and menus. Also included are associated objects employed in rendering the contents of a window: device contexts and drawing objects such as pens and brushes.

Window Classes

Class **CWnd** and its derived classes encapsulate an **HWND**, a handle to a Windows window. **CWnd** can be used by itself or as a base for deriving new classes. The derived classes supplied by the class library represent various kinds of windows.

CWnd
The base class for all windows. Use the derived classes below, or derive your own classes directly from **CWnd**.

CFrameWnd
The base class for an SDI application's main frame window.

CMDIFrameWnd
The base class for an MDI application's main frame window.

CMDIChildWnd
The base class for an MDI application's document frame windows.

View Classes

Class **CView** and its derived classes are child windows that represent the client area of a frame window and that show and accept input for a document.

CView
The base class for application-specific views of a document's data. Views display data and take user input to edit or select the data. Derive your view classes from **CView** or use **CScrollView** for automatic scrolling.

CScrollView
The base class for views with scrolling capabilities. Derive your view class from **CScrollView** for automatic scrolling.

CFormView
A scroll view whose layout is defined in a dialog resource. Derive classes from **CFormView** to quickly implement user interfaces based on dialog resources.

CEditView
A view with text-editing, searching, replacing, and scrolling capabilities. Use this class to provide a text-based user interface to a document.

Dialog Classes

Class **CDialog** and its derived classes encapsulate dialog-box functionality. Since a dialog box is a special kind of window, **CDialog** is derived from **CWnd**. Derive your dialog classes from **CDialog** or use one of the common dialog classes for standard dialog boxes such as opening or saving a file, printing, selecting a font or color, or initiating a search-and-replace operation.

CDialog
The base class for all dialog boxes—both modal and modeless.

CDataExchange
Supplies initialization and validation information for dialog boxes.

CFileDialog
Provides a standard dialog box for opening or saving a file.

CPrintDialog
Provides a standard dialog box for printing a file.

CFontDialog
Provides a standard dialog box for selecting a font.

CColorDialog
Provides a standard dialog box for selecting a color.

CFindReplaceDialog
Provides a standard dialog box for a search-and-replace operation.

Control Classes

Control classes encapsulate standard Windows controls such as buttons, list boxes, and combo boxes, as well as new controls, including buttons with bitmaps, edit controls for Microsoft Windows for Pen computing, control bars, and VBX custom controls. The Visual C++ class provides a programmatic interface to the Windows control.

CStatic
A static-text control window. Static controls are used to label, box, or separate other controls in a dialog box or window.

CButton
A button control window. The class provides a programmatic interface to a pushbutton, check box, or radio button in a dialog box or window.

CEdit
An editable-text control window. Edit controls are used to take textual input from the user.

CScrollBar
A scroll-bar control window. The class provides the functionality of a scroll bar for use as a control in a dialog box or window through which the user can specify a position within a range.

CListBox
A list-box control window. A list box displays a list of items that the user can view and select.

CComboBox
A combo-box control window. A combo box consists of an edit control plus a list box.

CHEdit
A Windows for Pens edit control in which the user can enter and modify text using standard pen editing gestures.

CBEdit
A Windows for Pens edit control in which the user can enter and modify text using standard pen editing gestures. This control differs from **CHEdit** in that it provides boxes to guide text entry.

CControlBar
A window aligned to the top or bottom of a frame window that contains **HWND**-based child controls or controls not based on an **HWND**, such as toolbar buttons. The base class for control bars such as toolbars and status bars.

CStatusBar
The base class for status-bar control windows.

CToolBar
Toolbar control windows that contain bitmap command buttons not based on an **HWND**.

CDialogBar
A modeless dialog box in the form of a control bar.

CBitmapButton
A button with a bitmap rather than a text caption.

CVBControl
A window whose implementation is a VBX control.

CSplitterWnd
A window that the user can split into multiple panes.

Menu Class

Class **CMenu** provides an interface through which to access your application's menus. It's useful for manipulating menus dynamically at run time; for example, you may want to add or delete menu items according to context.

CMenu
Encapsulates an **HMENU** handle to the application's menu bar and pop-up menus.

Device-Context Classes

Most of the following classes encapsulate a handle to a Windows device context. A device context is a Windows object that contains information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object. Additional classes derived from **CDC** encapsulate specialized device-context functionality, including support for Windows metafiles.

CDC
The base class for device contexts; used directly for accessing the whole display and for accessing nondisplay contexts such as printers.

CPaintDC
A display context used in **OnPaint** member functions of windows and **OnDraw** member functions of views. Automatically calls **BeginPaint** on construction and **EndPaint** on destruction.

CClientDC
A display context for client areas of windows. Used, for example, to draw in an immediate response to mouse events.

CWindowDC
A display context for entire windows, including both the client and frame areas.

CMetaFileDC
A device context for Windows metafiles. A Windows metafile contains a sequence of graphics device interface (GDI) commands that can be replayed to create an image. Calls made to the member functions of a **CMetaFileDC** are recorded in a metafile.

Drawing Object Classes

The following classes encapsulate handle-based GDI objects. They allow you to manipulate common GDI drawing objects with C++ syntax.

CGdiObject
The base class for GDI drawing tools.

CBitmap
Encapsulates a GDI bitmap, providing an interface for manipulating bitmaps.

CBrush
Encapsulates a GDI brush that can be selected as the current brush in a device context.

CFont
Encapsulates a GDI font that can be selected as the current font in a device context.

CPalette
Encapsulates a GDI color palette for use as an interface between the application and a color output device such as a display.

CPen
Encapsulates a GDI pen that can be selected as the current pen in a device context.

CRgn
Encapsulates a GDI region for manipulating an elliptical or polygonal area within a window. Used in conjunction with the clipping member functions in class **CDC**.

General-Purpose Classes

Classes in this category provide a variety of general-purpose services such as file I/O, diagnostics, and exception handling. Also included are classes such as arrays and lists for storing aggregates of data.

File Classes

Use the following classes, particularly **CArchive** and **CFile**, if you write your own input/output processing. Normally you don't need to derive from these classes. If you use the application framework, the default implementations of the Open and Save commands on the File menu handle file I/O (using class **CArchive**), provided you supply details about how a document "serializes" its contents. For more information about the file classes and serialization, see "The File Classes" on page 124 and Chapter 14, "Files and Serialization," in the *Class Library User's Guide*.

CFile
Provides a programmatic interface to binary disk files.

CMemFile
Provides a programmatic interface to in-memory files.

CStdioFile
Provides a programmatic interface to buffered stream disk files, usually in text mode.

CArchive
Cooperates with a **CFile** object to implement persistent storage for objects through serialization (see **CObject::Serialize**).

Diagnostics

Use classes **CDumpContext** and **CMemoryState** during development to assist with debugging, as described in Chapter 15, "Diagnostics," in the *Class Library User's Guide*. Use **CRuntimeClass** to determine the class of any object at run time, as described in Chapter 12, "The CObject Class," in the *Class Library User's Guide*. The framework uses **CRuntimeClass** to dynamically create objects of a particular class.

CDumpContext
Provides a destination for diagnostic dumps.

CMemoryState
Provides snapshots of memory use. The class is also used to compare earlier and later snapshots.

CRuntimeClass
Used to determine the exact class of an object at run time.

Exceptions

The class library provides an exception-handling mechanism based on class **CException**. The application framework uses exceptions in its code; you can also use them in yours. For more information, see "Exception Handling" on page 128. You can derive your own exception types from **CException**.

CException
The base class for exceptions.

CArchiveException
An archive exception.

CFileException
A file-oriented exception.

CMemoryException
An out-of-memory exception.

CNotSupportedException
An exception resulting from the invocation of an unsupported feature.

CResourceException
An exception resulting from a failure to load a Windows resource.

COleException
An exception resulting from failures in OLE processing. This class is used by both clients and servers.

CUserException
An exception used to stop a user-initiated operation. The user has typically been notified of the problem before this exception is thrown.

Collections

For handling aggregates of data, the class library provides a group of collection classes—arrays, lists, and "maps"—that can hold a variety of object and pre-defined types. The collections are dynamically sized. These classes can be used in any program, whether written for Windows or not. However, they are most useful for implementing the data structures that define your document classes in the application framework. You can readily derive specialized collection classes from these, or you can create them with a template tool supplied with the class library. For more information about these approaches, see "The Collection Classes" on page 124.

CByteArray
Stores elements of type **BYTE** in an array.

CDWordArray
Stores elements of type doubleword in an array.

CObArray
Stores pointers to objects of class **CObject** or to objects of classes derived from **CObject** in an array.

CPtrArray
Stores pointers to **void** (generic pointers) in an array.

CStringArray
Stores **CString** objects in an array.

CWordArray
Stores elements of type **WORD** in an array.

CUIntArray
Stores elements of type **UINT** in an array.

CObList
Stores pointers to objects of class **CObject** or to objects of classes derived from **CObject** in a linked list.

CPtrList
Stores pointers to **void** (generic pointers) in a linked list.

CStringList
Stores **CString** objects in a linked list.

CMapPtrToWord
Maps void pointers to data of type **WORD**. Uses void pointers as keys for finding data of type **WORD**.

CMapPtrToPtr
Maps void pointers to void pointers. Uses void pointers as keys for finding other void pointers.

CMapStringToOb
Maps **CString** objects to **CObject** pointers. Uses **CString** objects as keys for finding **CObject** pointers.

CMapStringToPtr
Maps **CString** objects to void pointers. Uses **CString** objects as keys for finding void pointers.

CMapStringToString
Maps **CString** objects to **CString** objects. Uses **CString** objects as keys for finding other **CString** objects.

CMapWordToOb
Maps data of type **WORD** to **CObject** pointers. Uses data of type **WORD** to find **CObject** pointers.

CMapWordToPtr
Maps data of type **WORD** to void pointers. Uses data of type **WORD** to find void pointers.

Miscellaneous Support Classes

The following classes encapsulate drawing coordinates, character strings, and time and date information, allowing convenient use of C++ syntax. These objects are used widely as parameters to the member functions of Windows classes in the Microsoft Foundation Class Library. Because **CPoint**, **CSize**, and **CRect** correspond to the **POINT**, **SIZE**, and **RECT** structures, respectively, in the Windows *Software Development Kit* (SDK), you can use objects of these C++ classes wherever you can use these C-language structures. The classes provide useful interfaces through their member functions. **CString** provides very flexible dynamic character strings. **CTime** and **CTimeSpan** represent time and date values. For more information about these classes, see "Other Support Classes" on page 126.

CPoint
Holds coordinate (x, y) pairs.

CSize
Holds distance, relative positions, or paired values.

CRect
Holds rectangular areas.

CString
Holds character strings.

CTime
Holds absolute time and date values.

CTimeSpan
Holds relative time and date values.

Object Linking and Embedding (OLE) Classes

The class library supplies four categories of classes to support Object Linking and Embedding: OLE base classes, OLE client classes, OLE server classes, and an OLE exception class. For more about using the OLE classes, see Chapter 18 in the *Class Library User's Guide*.

OLE Base Classes

The classes listed in this category serve as base classes for more specialized OLE classes in the other categories. These classes are listed here for completeness; you will not use them directly.

COleDocument
The abstract base class of the **COleClientDoc** and **COleServerDoc** classes. A **COleDocument** is the container for items of type **CDocItem**. A **COleClientDoc** contains items of type **COleClientItem** while a **COleServerDoc** contains items of type **COleServerItem**.

CDocItem
An item that is part of a document. Abstract base class of **COleClientItem** and **COleServerItem**.

OLE Client Classes

The class library supplies two classes for use in OLE client applications. **COleClientDoc** represents client documents, which maintain a collection of items of type **COleClientItem**. A **COleClientItem** represents the client view of an embedded or linked OLE item. These classes are derived from abstract base classes, as shown.

COleClientDoc
A client document class that manages client items. You must derive your documents from this class instead of **CDocument** to implement OLE client functionality.

COleClientItem
A client item class that represents the client's side of the connection to an embedded or linked OLE item. You must derive your client items from this class.

OLE Server Classes

An OLE server application has server objects for each of the document types it supports. A server creates and maintains server documents in much the same way that **CDocTemplate** objects create and maintain documents. For OLE objects embedded in a client application, the OLE server maintains one server document and one server item for each active item embedded in a client. For OLE objects linked to this server application, the OLE server maintains an OLE server document for each document that contains links. Each of these documents can be linked to multiple server items.

COleServer
A server application class that creates and manages server documents. You must derive a class from this class for each server type your application supports.

COleServerDoc
A server document class that creates and manages server items. You must derive your server documents from this class instead of **CDocument**.

COleServerItem
A server item class that represents the server's side of the connection to an embedded or linked OLE item. You must derive your server items from this class.

COleTemplateServer
An OLE server implementation class that manages server documents using a document template. This class can be used directly as an alternative to deriving from **COleServer**.

OLE Exception Class

The class library provides an exception class, derived from **CException**, for exceptional conditions that occur during OLE processing. For more information, see Chapter 3. For details about exception handling, see Chapter 16, "Exceptions," in the *Class Library User's Guide.*

COleException
An exception resulting from a failure in OLE processing. This class is used by both clients and servers.

Macros and Globals

The "Macros and Globals" section in Part 2 of this manual documents the elements of the Microsoft Foundation Class Library that are not defined as members of specific classes. These include macros and global functions and variables in the following general categories:

- Data types
- Run-time object model services
- Diagnostic services
- Exception processing
- **CString** formatting and message-box display
- Message maps
- Dialog data exchange and validation
- Application information and management
- OLE support
- Standard commands and window IDs

General Class Design Philosophy

Microsoft Windows was designed long before the C++ language became popular. Because thousands of applications use the C-language Windows application programming interface (API), that interface will be maintained for the foreseeable future. Any C++ Windows interface must therefore be built on top of the procedural C-language API. This guarantees that C++ applications will be able to coexist with C applications.

Design Goals

The Microsoft Foundation Class Library is truly an object-oriented interface to Windows that meets the following design goals:

- Significantly reduce the effort of programming an application for Windows
- Execution speed comparable to that of the C-language API
- Minimum code size overhead
- The ability to call any Windows C function directly
- Easier conversion of existing C applications to C++
- The ability to leverage from the existing base of C-language Windows programming experience
- Easier use of the Windows API with C++ than with C
- True Windows API for C++ that effectively uses C++ language features

The Application Framework

The core of the Microsoft Foundation Class Library is an encapsulation of a large portion of the Windows API in C++ form. Library classes represent windows, dialog boxes, device contexts, common GDI objects such as brushes and pens, controls, and other standard Windows items. These classes provide a convenient C++ member function interface to the structures in Windows that they encapsulate. For more information about these core classes, see "Window Objects" in Chapter 2.

But the Microsoft Foundation Class Library also supplies a layer of additional application functionality built on the C++ encapsulation of the Windows API. This layer is a working application framework for Windows that provides most of the common user interface expected of programs for Windows. Chapter 2 explains the framework in detail, and the *Class Library User's Guide* provides a tutorial that teaches application-framework programming.

Relationship to the C-Language API

The single characteristic that sets the Microsoft Foundation classes for Windows apart from other class libraries for Windows is the very close mapping to the Windows API written in the C language. Further, you can generally freely mix calls to the class library with direct calls to the Windows API. This direct access does not, however, imply that the classes are a complete replacement for that API. Developers must still occasionally make direct calls to some Windows functions—**GetSystemMetrics**, for example. A Windows function is wrapped by a class member function only if there is a clear advantage to doing so.

Because you sometimes need to make native Windows function calls, you should have access to the C-language Windows API documentation. This is included with Microsoft Visual C++ as Help. If you require printed documentation, refer to the

Microsoft Windows 3.1 Programmer's Reference and the *Microsoft Windows 3.1 Guide to Programming* from Microsoft Press. Another useful book is *Programming Windows* by Charles Petzold, also from Microsoft Press. Many of that book's examples can be easily converted to the Microsoft Foundation classes.

For examples and additional information about programming with the Microsoft Foundation Class Library version 2.0, see *Microsoft Visual C/C++ Programming for Windows* by David J. Kruglinski from Microsoft Press.

In Chapters to Come

Chapters 2 through 6 provide an overview of the framework and how it functions. Table 1.1 shows the topics covered by each chapter.

Table 1.1 Reference Overview Chapters

Chapter	Contents
2	The application object; creation of document templates, documents, views, and frame windows. How to initialize these objects.
3	Messages and commands; command routing; updating user-interface objects such as menus and toolbar buttons.
4	Documents and views; drawing in a view; working with multiple views; printing and print preview.
5	Dialog boxes and controls; control bars, including toolbars and status bars; using context-sensitive help.
6	Diagnostics; exception handling; files and serialization; collection classes.

The alphabetical reference for the classes in the Microsoft Foundation Class Library begins on page 131.

CHAPTER 2

Using the Classes to Write Applications for Windows

Taken together, the classes in the Microsoft Foundation Class Library make up an "application framework"—the framework on which you build an application for Windows. At a very general level, the framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton. Your job as programmer is to fill in the rest of the skeleton—those things that are specific to your application. You can get a head start by using AppWizard to create the files for a very thorough starter application. You use App Studio to design your user-interface elements visually, ClassWizard to connect those elements to code, and the class library to implement your application-specific logic.

This chapter presents a broad overview of the application framework. It also explores the major objects that make up your application and how they are created. Among the topics covered in this chapter are the following:

- The major objects in a running application
- Division of labor between the framework and your code
- The application class, which encapsulates application-level functionality
- How document templates create and manage documents and their associated views and frame windows
- Class **CWnd**, the root base class of all windows
- Graphic objects, such as pens and brushes
- The Windows Clipboard

Subsequent chapters continue the framework story, covering:

- Messages and commands (Chapter 3)
- Documents, views, and frame windows (Chapter 4)
- Dialog boxes, controls, control bars, and context-sensitive help (Chapter 5)

For a step-by-step tutorial in which you build an application with the framework, read the *Class Library User's Guide*, Chapters 1 through 10. Table 2.1 directs you to other documents:

Table 2.1 Where to Find More Information

Topic	Manual	Chapters
Classes mentioned in this chapter	*Class Library Reference*	Alphabetic reference
App Studio	*App Studio User's Guide*	
ClassWizard	*App Studio User's Guide*	9
	Class Library User's Guide	6, 7
	Visual Workbench User's Guide	13
AppWizard	*Visual Workbench User's Guide*	13
	Class Library User's Guide	2
Visual Workbench	*Visual Workbench User's Guide*	
Diagnostics, exceptions	*Class Library User's Guide*	15–16
Macros and globals	*Class Library Reference*	Alphabetic reference
Resources	*App Studio User's Guide*	

The Framework

This section introduces the major classes of the framework and three tools that simplify your work with the framework. Some of the classes encapsulate a large portion of the Microsoft Windows application programming interface (API). Other classes encapsulate application concepts such as documents, views, and the application itself.

SDI and MDI

The Microsoft Foundation Class Library makes it easy to work with both single document interface (SDI) and multiple document interface (MDI) applications.

SDI applications allow only one open document frame window at a time. MDI applications allow multiple document frame windows to be open in the same instance of an application. An MDI application has a window within which multiple MDI child windows, which are frame windows themselves, can be opened, each containing a separate document. In some applications, the child windows may be of different types, such as chart windows and spreadsheet windows. In that case, the menu bar may change as MDI child windows of different types are activated.

Documents, Views, and the Framework

At the heart of the framework are the concepts of document and view. A document is a data object with which the user interacts in an editing session. It is created by the New or Open commands on the File menu and is typically saved in a file. A view is a window object through which the user interacts with a document.

The key objects in a running application are:

- The document(s)

  Your document class (derived from **CDocument**) specifies your application's data.

- The view(s)

  Your view class (derived from **CView**) is the user's "window on the data." The view class specifies how the user sees your document's data and interacts with it. In some cases, you may want a document to have multiple views of the data.

  If you need scrolling, derive from **CScrollView**. If your view has a user interface that is laid out in a dialog-template resource, derive from **CFormView**. For simple text data, use or derive from **CEditView**.

- The frame windows

  Views are displayed inside "document frame windows." In an SDI application, the document frame window is also the "main frame window" for the application. In an MDI application, document windows are child windows displayed inside a main frame window. Your derived main frame-window class specifies the styles and other characteristics of the frame windows that contain your views. Derive from **CFrameWnd** to customize the document frame window for SDI applications. Derive from **CMDIFrameWnd** to customize the main frame window for MDI applications. Also derive a class from **CMDIChildWnd** to customize each of the distinct kinds of MDI document frame windows that your application supports.

- The document template(s)

  A document template orchestrates the creation of documents, views, and frame windows. A particular document-template class creates and manages all open documents of one type. Applications that support more than one type of document have multiple document templates. Use class **CSingleDocTemplate** for SDI applications, or use class **CMultiDocTemplate** for MDI applications.

- The application object

  Your application class (derived from **CWinApp**) controls all of the objects above and specifies application behavior such as initialization and cleanup. The application's one and only application object creates and manages the document templates for any document types the application supports.

In a running application, these objects cooperatively respond to user actions, bound together by commands and other messages. A single application object manages one or more document templates. Each document template creates and manages one or more documents (depending on whether the application is SDI or MDI). The user views and manipulates a document through a view contained inside a frame window. Figure 2.1 shows the relationships among these objects for an SDI application.

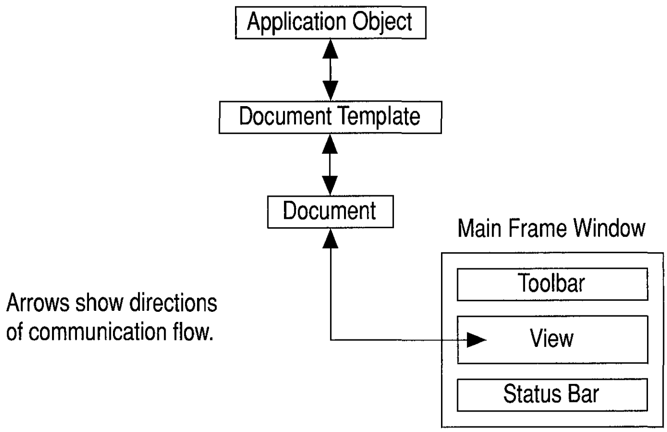


Figure 2.1 Objects in a Running SDI Application

The rest of this chapter explains how the framework creates these objects, how they work together, and how you use them in your programming. Documents, views, and frame windows are discussed in more detail in Chapter 4.

AppWizard

AppWizard creates a skeleton application upon which you can build your application-specific code.

You begin your application by invoking AppWizard from Visual Workbench. By default, AppWizard creates an MDI application, but you can change this through the Options dialog box. AppWizard then creates all of the necessary files and classes for the application type you have chosen.

An MDI application created by AppWizard already supports creating new MDI child windows when the user opens a document with the New or Open commands on the File menu. It handles changing the menu bar when an MDI child window of a different type receives the focus. It manages tiling or cascading open MDI child windows in response to the Tile and Cascade commands on the Window menu.

AppWizard also offers numerous options that let you incorporate support for toolbars, printing and print preview, VBX controls, context-sensitive help, and Object Linking and Embedding (OLE) in the files that AppWizard creates.

For more information about AppWizard, see Chapter 13 in the *Visual Workbench User's Guide* and Chapter 2 in the *Class Library User's Guide*.

App Studio

Use App Studio to design your application's user interface and create the application's resources: menus, dialog boxes, custom controls, accelerator keys, bitmaps, icons, cursors, and strings.

After creating a skeletal application with AppWizard, run App Studio from Visual Workbench. Select the type of resource you want to create or edit and open an editor for that type. App Studio lets you work easily and intuitively, operating visually upon visual objects. For example, to add controls to a dialog box, simply select a control icon on the Control Palette, drag it into the dialog box, and drop it in place. Editor functions make it easy to align and organize controls in a dialog box.

To help you even more, the Microsoft Foundation Class Library provides a file called COMMON.RC, which contains "clip art" resources that you can copy from COMMON.RC and paste into your own resource file. COMMON.RC includes toolbar buttons, common cursors, icons, and more. You can use, modify, and redistribute these resources in your application.

For more information about App Studio and COMMON.RC, see the *App Studio User's Guide*.

ClassWizard

Applications running under the Windows operating system are "message driven." User actions and other events that occur in the running program cause Windows to send messages to the windows in the program. For example, if the user clicks the mouse in a window, Windows sends a **WM_LBUTTONDOWN** message when the left mouse button is pressed and a **WM_LBUTTONUP** message when the button is released. Windows also sends **WM_COMMAND** messages when the user selects commands from the menu bar.

In the framework, various objects—documents, views, frame windows, document templates, the application object—can "handle" messages. Such an object provides a "handler function" as one of its member functions, and the framework maps the incoming message to its handler.

A large part of your programming task is choosing which messages to map to which objects and then implementing that mapping. To do so, you use the ClassWizard tool.

You can invoke ClassWizard from App Studio or from Visual Workbench. ClassWizard will create empty message-handler member functions and you use the Visual Workbench editor to implement the body of the handler.

For more information about messages, see Chapter 3, "Working with Messages and Commands." For more information about ClassWizard, see Chapter 9 in the *App Studio User's Guide*.

Building on the Framework

Your role in configuring an application with the framework is to supply the application-specific source code and to connect the components by defining what messages and commands they respond to. You use the C++ language and standard C++ techniques to derive your own application-specific classes from those supplied by the class library and to override and augment the base class's behavior.

Table 2.2 shows what you do in relation to what the framework does.

Table 2.2 Sequence in Building an Application with the Framework

Task	You Do	The Framework Does
Create a skeleton application.	Run AppWizard. Specify the options you want in the Options dialog box.	AppWizard creates the files for a skeleton application, including source files for your application, document, view, and frame windows; a resource file; a project file (.MAK); and others—all tailored to your specifications.
See what it offers without adding a line of your own code.	Build the skeleton application and run it in Visual Workbench.	The running skeleton application derives many standard File, Edit, View, and Help menu commands from the framework. For MDI applications, you also get a fully functional Window menu, and the framework manages creation, arrangement, and destruction of MDI child windows.

Table 2.2 Sequence in Building an Application with the Framework *(continued)*

Task	You Do	The Framework Does
Construct your application's user interface.	Use App Studio to visually edit the application's user interface: ■ Create menus. ■ Define accelerators. ■ Create dialog boxes. ■ Create and edit bitmaps, icons, and cursors. ■ Edit the toolbar bitmap created for you by AppWizard. ■ Create and edit other resources. You can also test the dialog boxes in App Studio.	The default resource file created by AppWizard supplies many of the resources you need. App Studio lets you edit existing resources and add new resources, easily and visually.
Map menus to handler functions.	Use ClassWizard to connect menus and accelerators to handler functions in your code.	ClassWizard inserts message-map entries and empty function templates in the source files you specify and manages many manual coding tasks.
Write your handler code.	Use ClassWizard to jump directly to the code in the Visual Workbench editor. Fill in the code for your handler functions.	ClassWizard brings up the editor, scrolls to the empty function template, and positions the cursor for you.
Map toolbar buttons to commands.	Map each button on your toolbar to a menu or accelerator command by assigning the button the appropriate command ID.	The framework controls the drawing, enabling, disabling, checking, and other visual aspects of the toolbar buttons.
Test your handler functions.	Rebuild the program and use Visual Workbench's built-in debugging tools to test that your handlers work correctly.	You can step or trace through the code to see how your handlers are called. If you've filled out the handler code, the handlers carry out commands. The framework will automatically disable menu items and toolbar buttons that are not handled.

Table 2.2 Sequence in Building an Application with the Framework *(continued)*

Task	You Do	The Framework Does
Create additional classes.	Use ClassWizard to create additional document, view, and frame-window classes beyond those created automatically by AppWizard.	ClassWizard adds these classes to your source files and helps you define their connections to any commands they handle.
Implement your document class.	Implement your application-specific document class(es). Add member variables to hold data structures. Add member functions to provide an interface to the data.	The framework already knows how to interact with document data files. It can open and close document files, read and write the document's data, and handle other user interfaces. You can focus on how the document's data is manipulated.
Implement Open, Save, and Save As commands.	Write code for the document's `Serialize` member function.	The framework displays dialog boxes for the Open, Save, and Save As commands on the File menu. It writes and reads back a document using the data format specified in your `Serialize` member function.
Implement your view class.	Implement one or more view classes corresponding to your documents. Implement the view's member functions that you mapped to the user interface with ClassWizard.	The framework manages most of the relationship between a document and its view. The view's member functions access the view's document to render its image on the screen or printed page and to update the document's data structures in response to user editing commands.
Enhance default printing.	If you need to support multipage printing, override view member functions.	The framework supports the Print, Print Setup, and Print Preview commands on the File menu. You must tell it how to break your document into multiple pages.

Table 2.2 Sequence in Building an Application with the Framework *(continued)*

Task	You Do	The Framework Does
Add scrolling.	If you need to support scrolling, derive your view class(es) from **CScrollView**.	The view automatically adds scroll bars when the view window becomes too small.
Create form views.	If you want to base your views on dialog-template resources, derive your view class(es) from **CFormView**.	The view uses the dialog-template resource to display controls. The user can tab from control to control in the view.
Create a simple text editor.	If you want your view to be a simple text editor, derive your view class(es) from **CEditView**.	The view provides editing functions, Clipboard support, and file input/output.
Add splitter windows.	If you want to support window splitting, add a **CSplitterWnd** object to your SDI frame window or MDI child window and hook it up in the window's **OnCreateClient** member function.	The framework supplies splitter-box controls next to the scroll bars and manages splitting your view into multiple panes. If the user splits a window, the framework creates and attaches additional view objects to the document.
Add dialog boxes.	Design dialog-template resources with App Studio. Then use ClassWizard to create a dialog class and the code that handles the dialog box.	The framework manages the dialog box and facilitates retrieving information entered by the user.
Initialize, validate, and retrieve dialog-box data.	You can also define how the dialog box's controls are to be initialized and validated. Use ClassWizard to add member variables to the dialog class and map them to dialog controls. Specify validation rules to be applied to each control as the user enters data. Provide your own custom validations if you wish.	The framework manages dialog-box initialization and validation. If the user enters invalid information, the framework puts up a message box and lets the user reenter the data.
Build, test, and debug your application.	Use the facilities of Visual Workbench to build, test, and debug your application.	Visual Workbench is closely coupled with AppWizard, App Studio, and ClassWizard. It lets you adjust compile, link, and other options. And it lets you browse your source code and class structure.

As you can see, AppWizard, App Studio, and ClassWizard do a lot of work for you and make managing your code much easier. The bulk of your application-specific code is in your document and view classes. For a tour of this process with a real application, see Chapters 1 through 10 in the *Class Library User's Guide*.

While it is possible to do these tasks by hand or using other tools, your savings in time, energy, and errors suggest that using the tools is greatly to your benefit.

You will learn more about these tools in the rest of this chapter. For more information about AppWizard, see Chapter 13 in the *Visual Workbench User's Guide* and Chapter 3 in the *Class Library User's Guide*. For more information about App Studio, see the *App Studio User's Guide*. For more information about ClassWizard, see Chapter 9 in the *App Studio User's Guide*, Chapter 13 in the *Visual Workbench User's Guide*, and Chapters 6 and 7 in the *Class Library User's Guide*. For information about resources and resource files, see the *App Studio User's Guide*.

How the Framework Calls Your Code

It is crucial to understand the relationship between your source code and the code in the framework. When your application runs, most of the flow of control resides in the framework's code. The framework manages the message loop that gets messages from Windows as the user chooses commands and edits data in a view. Events that the framework can handle by itself don't rely on your code at all. For example, the framework knows how to close windows and how to exit the application in response to user commands. As it handles these tasks, the framework uses message handlers and C++ virtual functions to give you opportunities to respond to these events as well. But your code is not in the driver's seat.

Your code is called by the framework for application-specific events. For example, when the user chooses a menu command, the framework routes the command along a sequence of C++ objects: the current view and frame window, the document associated with the view, the document's document template, and the application object. If one of these objects can handle the command, it does so, calling the appropriate message-handler function. For any given command, the code called may be yours or it may be the framework's.

This arrangement is somewhat familiar to programmers experienced with traditional programming for Windows or event-driven programming.

In the next several sections, you'll see what the framework does as it initializes and runs the application and then cleans up as the application terminates. You'll also get a clearer picture of where the code you write fits in.

CWinApp: The Application Class

The main application class encapsulates the initialization, running, and termination of an application for Windows. An application built on the framework must have one (and only one) object of a class derived from **CWinApp**. This object is constructed before windows are created.

Like any program for Windows, your framework application has a **WinMain** function. In a framework application, however, you don't write **WinMain**. It is supplied by the class library and is called when the application starts up. **WinMain** performs standard services such as registering window classes. Then it calls member functions of the application object to initialize and run the application.

To initialize the application, **WinMain** calls your application object's `InitApplication` and `InitInstance` member functions. To run the application's message loop, **WinMain** calls the **Run** member function. On termination, **WinMain** calls the application object's `ExitInstance` member function. Figure 2.2 shows the sequence of execution in a framework application.

WinMain	Standard function supplied by framework
calls	
└─► `InitInstance`	Initializes current instance of the application
calls	
└─► **Run**	Runs the message loop and **OnIdle**
calls	
└─► `ExitInstance`	Cleans up after the application

Figure 2.2 Sequence of Execution

Note Names shown in bold type indicate elements supplied by the Microsoft Foundation Class Library. Names shown in monospaced type indicate elements that you create or override.

CWinApp and AppWizard

When it creates a skeleton application, AppWizard declares an application class derived from **CWinApp**. AppWizard also generates an implementation file that contains the following items:

- A message map for the application class
- An empty class constructor

- A variable that declares the one and only object of the class
- A standard implementation of your `InitInstance` member function

The application class is placed in the project header and main source files. The names of the class and files created are based on the project name you supply in the AppWizard dialog box.

The standard implementations and message map supplied are adequate for many purposes, but you can modify them as needed. The most interesting of these implementations is the `InitInstance` member function. Typically you will add code to the skeletal implementation of `InitInstance`.

Overridable CWinApp Member Functions

CWinApp provides several key overridable member functions. The only **CWinApp** member function that you must override is **InitInstance**.

InitInstance

Windows allows you to run more than one copy, or "instance," of the same application. **WinMain** calls **InitInstance** every time a new instance of the application starts.

The standard `InitInstance` implementation created by AppWizard performs the following tasks:

- Loads standard file options from an .INI file, including the names of the most recently used files.
- Registers one or more document templates.
- For an MDI application, creates a main frame window.
- Processes the command line to open a document specified on the command line or to open a new, empty document.

The central action of `InitInstance` is to create the document templates that, in turn, create documents, views, and frame windows. For a description of this process, see "Document Templates" on page 33.

ExitInstance

The **ExitInstance** member function of class **CWinApp** is called each time a copy of your application terminates, usually as a result of the user quitting the application. Override **ExitInstance** if you need special cleanup processing, such as freeing graphics device interface (GDI) resources or deallocating memory used during program execution. Cleanup of standard items such as documents and views, however, is provided by the framework, with other overridable functions for doing special cleanup specific to those objects.

OnIdle

When no Windows messages are being processed, the framework calls the **CWinApp** member function **OnIdle**. Override **OnIdle** to perform background tasks. The default version updates the state of user-interface objects such as toolbar buttons and performs cleanup of temporary objects created by the framework in the course of its operations. Figure 2.3 illustrates how the message loop calls **OnIdle** when there are no messages in the queue.

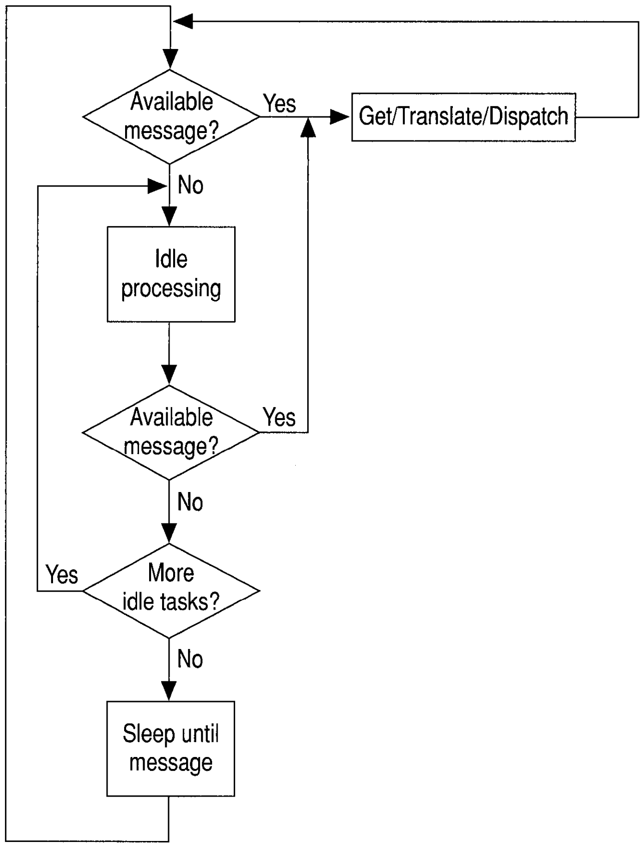


Figure 2.3 The Message Loop

The Run Function

A framework application spends most of its time in the **Run** member function of class **CWinApp**. After initialization, **WinMain** calls **Run** to process the message loop.

Run cycles through a message loop, checking the message queue for available messages. If a message is available, **Run** dispatches it for action. If no messages are available—often the case—**Run** calls **OnIdle** to do any idle-time processing that you or the framework may need done. If there are no messages and no idle processing to do, the application waits until something happens. When the application terminates, **Run** calls **ExitInstance**. Figure 2.3 above shows the sequence of actions in the message loop.

Message dispatching depends on the kind of message. For more information, see Chapter 3, "Working with Messages and Commands."

Other CWinApp Services

Besides running the message loop and giving you an opportunity to initialize the application and clean up after it, **CWinApp** provides several other services.

Shell Registration

By default, AppWizard makes it possible for the user to open data files that your application has created by double-clicking them in the Windows File Manager. If your application is an MDI application and you specify an extension for the files your application creates, AppWizard adds calls to the **EnableShellOpen** and **RegisterShellFileTypes** member functions of **CWinApp** to the `InitInstance` override that it writes for you.

RegisterShellFileTypes registers your application's document types with File Manager. The function adds entries to the registration database that Windows maintains. The entries register each document type, associate a file extension with the file type, specify a command line to open the application, and specify a dynamic data exchange (DDE) command to open a document of that type.

EnableShellOpen completes the process by allowing your application to receive DDE commands from File Manager to open the file chosen by the user.

This automatic registration support in **CWinApp** eliminates the need to ship an .REG file with your application or to do special installation work.

File Manager Drag and Drop

Windows versions 3.1 and later allow the user to drag filenames from the file view window in the File Manager and drop them into a window in your application. You might, for example, allow the user to drag one or more filenames into an MDI application's main window, where the application could retrieve the filenames and open MDI child windows for those files.

To enable file drag and drop in your application, AppWizard writes a call to the **CWnd** member function **DragAcceptFiles** for your main frame window in your `InitInstance`. You can remove that call if you do not want to implement the drag-and-drop feature.

Keeping Track of the Most Recently Used Documents

As the user opens and closes files, the application object keeps track of the four most recently used files. The names of these files are added to the File menu and updated when they change. The framework stores these filenames in an .INI file with the same name as your project and reads them from the file when your application starts up. The `InitInstance` override that AppWizard creates for you includes a call to the **CWinApp** member function **LoadStdProfileSettings**, which loads information from the .INI file, including the most recently used filenames.

Document Templates

To manage the complex process of creating documents with their associated views and frame windows, the framework uses two document template classes: **CSingleDocTemplate** for SDI applications and **CMultiDocTemplate** for MDI applications. A **CSingleDocTemplate** can create and store one document of one type at a time. A **CMultiDocTemplate** keeps a list of many open documents of one type.

Some applications support multiple document types. For example, an application might support text documents and graphics documents. In such an application, when the user chooses the New command on the File menu, a dialog box shows a list of possible new document types to open. For each supported document type, the application uses a distinct document template object. Figure 2.4 illustrates the configuration of an MDI application that supports two document types. The figure shows several open documents.

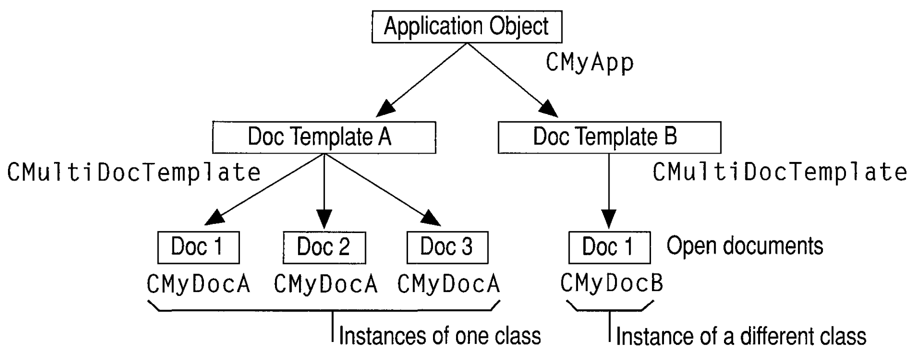


Figure 2.4 An MDI Application with Two Document Types

Document templates are created and maintained by the application object. One of the key tasks performed during your application's `InitInstance` function is to construct one or more document templates of the appropriate kind. This feature is described in "Document Template Creation" below. The application object stores a pointer to each document template in its template list and provides an interface for adding and removing document templates.

If you need to support two or more document types, you must add an extra call to **AddDocTemplate** for each document type.

Document Template Creation

While creating a new document in response to a New or Open command from the File menu, the document template also creates a new frame window through which to view the document.

The document-template constructor specifies what types of documents, windows, and views the template will be able to create. This is determined by the arguments you pass to the document-template constructor. The following code illustrates creation of a **CMultiDocTemplate** for a sample application:

```
AddDocTemplate( new CMultiDocTemplate( IDR_SCRIBTYPE,
          RUNTIME_CLASS( CScribDoc ),
          RUNTIME_CLASS( CMDIChildWnd ),
          RUNTIME_CLASS( CScribView ) ) );
```

The pointer to a new **CMultiDocTemplate** object is used as an argument to **AddDocTemplate**. Arguments to the **CMultiDocTemplate** constructor include the resource ID associated with the document type's menus and accelerators, and three uses of the **RUNTIME_CLASS** macro. **RUNTIME_CLASS** returns the **CRuntimeClass** object for the C++ class named as its argument. The three **CRuntimeClass** objects passed to the document-template constructor supply the information needed to create new objects of the specified classes during the document creation process. The example shows creation of a document template that creates `CScribDoc` objects with `CScribView` objects attached. The views are framed by standard MDI child frame windows.

Document/View Creation

The framework supplies implementations of the New and Open commands (among others) on the File menu. Creation of a new document and its associated view and frame window is a cooperative effort among the application object, a document template, the newly created document, and the newly created frame window. Table 2.3 summarizes which objects create what.

Table 2.3 Object Creators

Creator	Creates
Application object	Document template
Document template	Document
Document template	Frame window
Frame window	View

Relationships Among Documents, Views, Frame Windows, Templates, and the Application

To help put the document/view creation process in perspective, first consider a running program: a document, the frame window used to contain the view, and the view associated with the document.

- A document keeps a list of the views of that document and a pointer to the document template that created the document.
- A view keeps a pointer to its document and is a child of its parent frame window.
- A document frame window keeps a pointer to its current active view.
- A document template keeps a list of its open documents.
- The application keeps a list of its document templates.
- Windows keeps track of all open windows so it can send messages to them.

These relationships are established during document/view creation. Table 2.4 shows how objects in a running program can access other objects. Any object can obtain a pointer to the application object by calling the global function **AfxGetApp**.

Table 2.4 How to Access Other Objects

From Object	How to Access Other Objects
Document	Use **GetFirstViewPosition** and **GetNextView** to access the document's view list.
	Call **GetDocTemplate** to get the document template.
View	Call **GetDocument** to get the document.
	Call **GetParentFrame** to get the frame window.
Document frame window	Call **GetActiveView** to get the current view.
MDI frame window	Call **MDIGetActive** to get the currently active **CMDIChildWnd**.

Typically, a frame window has one view, but sometimes, as in splitter windows, the same frame window contains multiple views. The frame window keeps a pointer to the currently active view; the pointer is updated any time another view is activated.

Note A pointer to the main frame window is stored in the **m_pMainWnd** member variable of the application object. You must set the value of this variable in your override of **CWinApp**'s **InitInstance** member function.

Creating New Documents, Windows, and Views

Figures 2.5, 2.6, and 2.7 give an overview of the creation process for documents, views, and frame windows. Later chapters that focus on the participating objects provide further details.

Upon completion of this process, the cooperating objects exist and store pointers to each other. These figures show the sequence in which objects are created. You can follow the sequence from figure to figure.

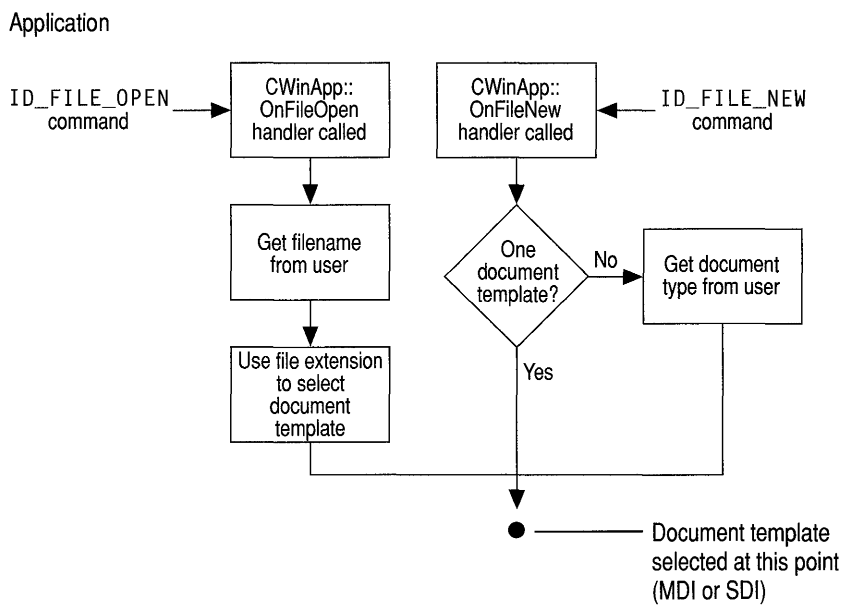


Figure 2.5 Sequence in Creating a Document

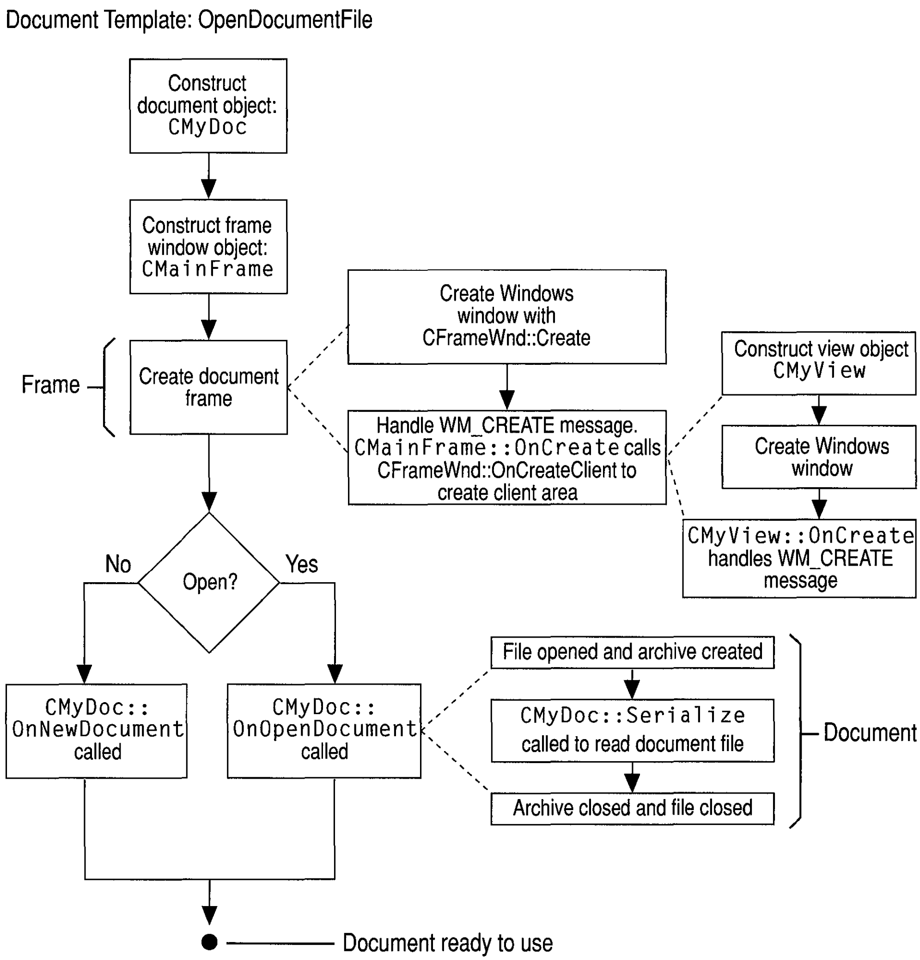


Figure 2.6 Sequence in Creating a Frame Window

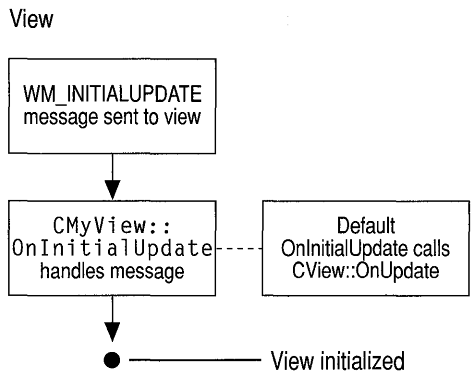


Figure 2.7 Sequence in Creating a View

Initializing the New Objects

For information about how the framework initializes the new document, view, and frame window objects, see classes **CDocument**, **CView**, **CFrameWnd**, **CMDIFrameWnd**, and **CMDIChildWnd** in the alphabetic reference. Also see Technical Note 22 in MSVC\HELP\MFCNOTES.HLP, which explains the creation and initialization processes further under its discussion of the framework's standard commands for the New and Open items on the File menu.

Initializing Your Own Additions to These Classes

Figures 2.5, 2.6, and 2.7 also suggest the points at which you can override member functions to initialize your application's objects. An override of **OnInitialUpdate** in your view class is the best place to initialize the view. The **OnInitialUpdate** call occurs immediately after the frame window is created and the view within the frame window is attached to its document. For example, if your view is a scroll view (derived from **CScrollView** rather than **CView**), you should set the view size based on the document size in your `OnInitialUpdate` override. (This process is described in the description of class **CScrollView**.) You can override the **CDocument** member functions **OnNewDocument** and **OnOpenDocument** to provide application-specific initialization of the document. Typically, you must override both since a document can be created in two ways.

In most cases, your override should call the base class version. For more information, see the named member functions of classes **CDocument**, **CView**, **CFrameWnd**, and **CWinApp**.

Windows of Your Own

Although the framework provides windows on your documents, you may at times want to create your own windows, particularly child windows. Keeping in mind how much the framework does for you, this section discusses windows in a more general way, with particular emphasis on creating windows of your own. For more information about the frame windows that the framework creates, see Chapter 4.

Class CWnd

In the Microsoft Foundation Class Library, all windows are ultimately derived from class **CWnd**. This includes dialog boxes, controls, control bars, and views as well as frame windows and your own child windows, as shown in the Microsoft Foundation Class Library hierarchy diagram on page xvi.

Window Objects

A C++ window object (whether for a frame window or some other kind of window) is distinct from its corresponding Windows window (the **HWND**), but the two are tightly linked. A good understanding of this relationship is crucial for effective programming with the Microsoft Foundation Class Library.

The window *object* is an object of the C++ **CWnd** class (or a derived class) that your program creates directly. It comes and goes in response to your program's constructor and destructor calls. The Windows *window*, on the other hand, is an opaque handle to an internal Windows data structure that corresponds to a window and consumes system resources when present. A Windows window is identified by a "window handle" (**HWND**) and is created after the **CWnd** object is created by a call to the **Create** member function of class **CWnd**. The window may be destroyed either by a program call or by a user's action. The window handle is stored in the window object's **m_hWnd** member variable. Figure 2.8 shows the relationship between the C++ window object and the Windows window. Creating windows is discussed in "Creating Windows" on page 42. Destroying windows is discussed in "Destroying Windows" on page 43.

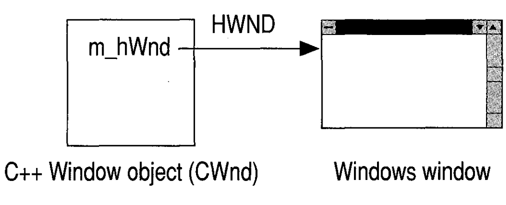


Figure 2.8 Window Object and Windows Window

CWnd Member Functions

CWnd and its derived classes provide constructors, destructors, and member functions to initialize the object, create the underlying Windows structures, and access the encapsulated **HWND**. **CWnd** also provides member functions that encapsulate Windows APIs for sending messages, accessing the window's state, converting coordinates, updating, scrolling, accessing the Clipboard, and many other tasks. Most Windows window-management APIs that take an **HWND** argument are encapsulated as member functions of **CWnd**. The names of the functions and their parameters are preserved in the **CWnd** member function. For details about the Windows APIs encapsulated by **CWnd**, see class **CWnd** in the alphabetic reference.

The general literature on programming for Windows is a good resource for learning how to use the **CWnd** member functions, which typically encapsulate the **HWND** APIs. For example, see Charles Petzold's *Programming Windows 3.1*, third edition.

Message Handling

One of the primary purposes of **CWnd** is to provide an interface for handling Windows messages, such as **WM_PAINT** or **WM_MOUSEMOVE**. Many of the member functions of **CWnd** are handlers for standard messages—those beginning with the identifier **afx_msg** and the prefix "On," such as **OnPaint** and **OnMouseMove**. Chapter 3 covers messages and message handling in detail. The information there applies equally to the framework's windows and those that you create yourself for special purposes.

Derived Window Classes

Although you can create windows directly from **CWnd**, or derive new window classes from **CWnd**, most windows used in a framework program are instead created from one of the **CWnd**-derived frame-window classes supplied by the Microsoft Foundation Class Library:

CFrameWnd
: Used for SDI frame windows that frame a single document and its view. The frame window is both the main frame window for the application and the frame window for the current document.

CMDIFrameWnd
: Used as the main frame window for MDI applications. The main frame window is a container for all MDI document windows and shares its menu bar with them. An MDI frame window is a top-level window that appears on the desktop.

CMDIChildWnd

Used for individual documents opened in an MDI main frame window. Each document and its view are framed by an MDI child frame window contained by the MDI main frame window. An MDI child window looks much like a typical frame window but is contained inside an MDI frame window instead of sitting on the desktop. However, the MDI child window lacks a menu bar of its own and must share the menu bar of the MDI frame window that contains it. Figure 2.9 shows an MDI application whose main frame window contains two MDI document windows. Each document window contains a document and its view.

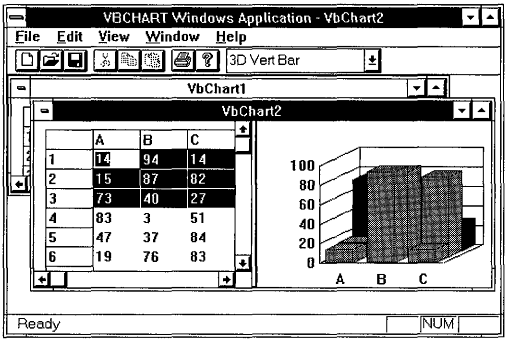


Figure 2.9 An MDI Frame Window with Children

In addition to frame windows, several other major categories of windows are derived from **CWnd**:

Views

Views are created using the **CWnd**-derived class **CView** (or one of its derived classes). A view is attached to a document and acts as an intermediary between the document and the user. A view is a child window (not an MDI child) that typically fills the client area of an SDI frame window or an MDI child frame window.

Dialog Boxes

Dialog boxes are created using the **CWnd**-derived class **CDialog**.

Controls

Controls such as buttons, list boxes, and combo boxes are created using other classes derived from **CWnd**.

Control Bars

Child windows that contain controls. Examples include toolbars and status bars.

Refer again to the Microsoft Foundation Class Library hierarchy diagram on page xvi. Views are explained in Chapter 4. Dialog boxes, controls, and control bars are explained in Chapter 5.

In addition to the window classes provided by the class library, you may need special-purpose child windows. To create such a window, write your own **CWnd**-derived class and make it a child window of a frame window or view.

Bear in mind that the framework manages the client area of a document frame window. Most of the client area is managed by a view, but other windows, such as control bars or your own custom windows, may share the space with the view. You may need to interact with the mechanisms in classes **CView** and **CControlBar** for positioning child windows in a frame window's client area.

The next section discusses creation of window objects and the Windows windows they manage.

Creating Windows

Most of the windows you need in a framework program are created automatically by the framework. You have already seen, in this chapter, how the framework creates the frame windows associated with documents and views. This section discusses window creation at a more general level. The material presented here is especially useful if you need to create your own windows—in addition to the windows supplied by the framework—for special purposes.

Registering Window "Classes"

In a traditional Windows program, you process all messages to a window in its "window procedure" or "**WndProc**." A **WndProc** is associated with a window by means of a "window class registration" process. The main window is registered in the **WinMain** function, but other classes of windows can be registered anywhere in the application. Registration depends on a structure that contains a pointer to the **WndProc** function together with specifications for the cursor, background brush, and so forth. The structure is passed as a parameter, along with the string name of the class, in a prior call to the **RegisterClass** function. Thus a registration class can be shared by multiple windows.

In contrast, most window class registration activity is done automatically in a framework program. If you are using the Microsoft Foundation Class Library, you typically derive a C++ window class from an existing library class using the normal C++ syntax for class inheritance. The framework still uses traditional "registration classes," and it provides several standard ones, registered for you in the standard application initialization function. You can register additional registration classes by calling the **AfxRegisterWndClass** global function and then pass the registered class to the **Create** member function of **CWnd**. As described here, the traditional Windows "registration class" is not to be confused with a C++ class.

For more information, see Technical Note 1 in MFCNOTES.HLP.

General Creation Sequence

If you are creating a window of your own, such as a child window, this section describes what you need to know. The framework uses much the same process to create windows for your documents as that described earlier in the chapter.

All the window classes provided by the Microsoft Foundation Class Library employ two-phase construction. That is, during an invocation of the C++ **new** operator, the constructor allocates and initializes a C++ object but does not create a corresponding Windows window. That is done afterwards by calling the **Create** member function of the window object.

The **Create** member function makes the Windows window and stores its **HWND** in the C++ object's public data member **m_hWnd**. **Create** gives complete flexibility over the creation parameters. Before calling **Create**, you may want to register a window class with **AfxRegisterWndClass** in order to set the icon and class styles for the frame.

For frame windows, the **LoadFrame** member function can be used instead of **Create**. **LoadFrame** makes the Windows window using fewer parameters. It gets many default values from resources, including the frame's caption, icon, accelerator table, and menu.

Note Your icon, accelerator table, and menu resources must have a common resource ID, such as **IDR_MAINFRAME**.

Destroying Windows

Care must be taken with your own child windows to destroy the C++ window object when the user is finished with the window. If these objects are not destroyed, your application will not recover their memory. Fortunately, the framework manages window destruction as well as creation for frame windows, views, and dialog boxes. If you create additional windows, you are responsible for destroying them.

In the framework, when the user closes the frame window, the window's default **OnClose** handler calls **DestroyWindow**. The last member function called when the Windows window is destroyed is **OnNcDestroy**, which does some cleanup, calls the **Default** member function to perform Windows cleanup, and lastly calls the virtual member function **PostNcDestroy**. The **CFrameWnd** implementation of **PostNcDestroy** deletes the C++ window object.

Do not use the C++ **delete** operator to destroy a frame window or view. Instead, call the **CWnd** member function **DestroyWindow**. Frame windows, therefore, should be allocated on the heap with operator **new**. Care must be taken when allocating frame windows on the stack frame or globally. Other windows should be allocated on the stack frame whenever possible.

If you need to circumvent the object-**HWND** relationship, the Microsoft Foundation Class Library provides another **CWnd** member function, **Detach**, which disconnects the C++ window object from the Windows window. This prevents the destructor from destroying the Windows window when the object is destroyed.

Working With Windows

Working with windows calls for two kinds of activity:

- Handling Windows messages
- Drawing in the window

To handle Windows messages in any window, including your own child windows, use ClassWizard to map the messages to your window class. Then write message-handler member functions in your class. Chapter 3 details message handling.

Most drawing in a framework application occurs in the view, whose `OnDraw` member function is called whenever the window's contents must be drawn. If your window is a child of the view, you might delegate some of the view's drawing to your child window by having `OnDraw` call one of your window's member functions.

In any case, you will need a device context for drawing.

Device Contexts

A device context is a Windows data structure that contains information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object, which encapsulates the Windows APIs for drawing lines, shapes, and text. Device contexts allow device-independent Windows drawing. Device contexts can be used to draw to the screen, to the printer, or to a metafile.

Special Device-Context Classes

CPaintDC objects encapsulate the common Windows idiom of calling the **BeginPaint** function, then drawing in the device context, then calling the **EndPaint** function. The **CPaintDC** constructor calls **BeginPaint** for you, and the destructor calls **EndPaint**. The simplified process is to create the **CDC** object, draw, and destroy the **CDC** object. In the framework, much of even this process is automated. In particular, your `OnDraw` function is passed a **CPaintDC** already prepared (via **OnPrepareDC**), and you simply draw into it. It is destroyed by the framework and the underlying Windows device context is released to Windows upon return from the call to your `OnDraw` function.

CClientDC objects encapsulate working with a device context that represents only the client area of a window. The **CClientDC** constructor calls the **GetDC** function, and the destructor calls the **ReleaseDC** function. **CWindowDC** objects encapsulate a device context that represents the whole window, including its frame.

CMetaFileDC objects encapsulate drawing into a Windows metafile. In contrast to the **CPaintDC** passed to `OnDraw`, you must in this case call **OnPrepareDC** yourself. For more information about these classes, see the alphabetic reference.

Drawing is discussed in greater detail in Chapter 4.

Other Device-Context Uses

Although most drawing—and thus most device-context work—in a framework program is done in the view's `OnDraw` member function, as described in Chapter 4, you can still use device-context objects for other purposes. For example, to provide tracking feedback for mouse movement in a view, you need to draw directly into the view without waiting for `OnDraw` to be called.

In such a case, you can use a **CClientDC** device-context object to draw directly into the view. For more information about mouse drawing, see "Interpreting User Input Through a View" in Chapter 4.

Graphic Objects

Windows provides a variety of drawing tools to use in device contexts. It provides pens to draw lines, brushes to fill interiors, and fonts to draw text. The Microsoft Foundation Class Library provides graphic-object classes equivalent to the drawing tools in Windows. Table 2.5 shows the available classes and the equivalent Windows GDI handle types.

The general literature on programming for the Windows GDI applies to the Microsoft Foundation classes that encapsulate GDI graphic objects. This section explains the use of the graphic-object classes.

Table 2.5 Graphic Objects

Classes	Windows Handle Types
CPen	**HPEN**
CBrush	**HBRUSH**
CFont	**HFONT**
CBitmap	**HBITMAP**
CPalette	**HPALETTE**
CRgn	**HRGN**

Each of the graphic-object classes in the class library has a constructor that allows you to create graphic objects of that class, which you must then initialize with the appropriate create function, such as **CreatePen**.

The following four steps are typically used when you need a graphic object for a drawing operation:

1. Define a graphic object on the stack frame. Initialize the object with the type-specific create function, such as **CreatePen**. Alternatively, initialize the object in the constructor. See the discussion of one-stage and two-stage creation below.
2. Select the object into the current device context, saving the old graphic object that was selected before.
3. When done with the current graphic object, select the old graphic object back into the device context to restore its state.
4. Allow the frame-allocated graphic object to be deleted automatically when the scope is exited.

Note If you will be using a graphic object repeatedly, you can allocate it once and select it into a device context each time it is needed. Be sure to delete such an object when you no longer need it.

You have a choice between two techniques for creating graphic objects:

- One-stage construction: Construct and initialize the object in one stage, all with the constructor.
- Two-stage construction: Construct and initialize the object in two separate stages. The constructor creates the object and an initialization function initializes it.

Two-stage construction is always safer. In one-stage construction, the constructor could throw an exception if you provide incorrect arguments or memory allocation fails. That problem is avoided by two-stage construction, although you do have to check for failure. In either case, destroying the object is the same process.

The following brief example shows both methods of constructing a pen object:

```
void CMyView::OnDraw( CDC* pDC )
{
    CPen myPen1( PS_DOT, 5, RGB(0,0,0) );    // One-stage
    // Two-stage: first construct the pen
    CPen myPen2;
    // Then initialize it
    if( myPen2.CreatePen( PS_DOT, 5, RGB(0,0,0) ) )
        // Use the pen
}
```

After you create a drawing object, you must select it into the device context in place of the default pen stored there:

```
void CMyView::OnDraw( CDC* pDC )
{
    CPen penBlack;  // Construct it, then initialize
    if( newPen.CreatePen( PS_SOLID, 2, RGB(0,0,0) ) )
    {
        // Select it into the device context
        // Save the old pen at the same time
        CPen* pOldPen = pDC->SelectObject( &penBlack );

        // Draw with the pen
        pDC->MoveTo(...);
        pDC->LineTo(...);

        // Restore the old pen to the device context
        pDC->SelectObject( pOldPen );
    }
    else
    {
        // Alert the user that resources are low
    }
}
```

The graphic object returned by **SelectObject** is a "temporary" object. That is, it will be deleted by the **OnIdle** member function of class **CWinApp** the next time the program gets idle time. As long as you use the object returned by **SelectObject** in a single function without returning control to the main message loop, you will have no problem.

How to Use the Clipboard

Most applications for Windows support cutting or copying data to the Windows Clipboard and pasting data from the Clipboard. The Clipboard data formats vary among applications. The framework supports only a limited number of Clipboard formats for a limited number of classes. You will normally implement the Clipboard-related commands—Cut, Copy, and Paste—on the Edit menu for your view. The class library defines the command IDs for these commands: **ID_EDIT_CUT**, **ID_EDIT_COPY**, and **ID_EDIT_PASTE**. Their message-line prompts are also defined.

The Clipboard is a system service shared by the entire Windows session, so it does not have a handle or class of its own. You manage the Clipboard through member functions of class **CWnd**.

Chapter 3 explains how to handle menu commands in your application by mapping the menu command to a handler function. As long as your application does not define handler functions for the Clipboard commands on the Edit menu, they remain disabled. To write handler functions for the Cut and Copy commands, implement selection in your application. To write a handler function for the Paste command, query the Clipboard to see whether it contains data in a format your application can accept. For example, to enable the Copy command, you might write a handler something like the following:

```
void CMyView::OnEditCopy()
{
   if(!OpenClipboard())
   {
      AfxMessageBox("Cannot open the Clipboard");
      return;
   }
   // ...
   // Get the currently selected data
   // ...
   // For the appropriate data formats...
   SetClipboardData(CF_??, hData);
   // ...
   CloseClipboard();
}
```

The Cut, Copy, and Paste commands are only meaningful in certain contexts. The Cut and Copy commands should be enabled only when something is selected, and the Paste command only when something is in the Clipboard. You can provide this behavior by defining update handler functions that enable or disable these commands depending on the context. For more information, see "How to Update User-Interface Objects" on page 67 in Chapter 3.

The Microsoft Foundation Class Library does provide Clipboard support for text editing with the **CEdit** and **CEditView** classes. The Object Linking and Embedding (OLE) classes also simplify implementing Clipboard operations that involve OLE items. For more information on the OLE classes, see Chapter 18 in the *Class Library User's Guide*.

Implementing other Edit menu commands, such as Undo (**ID_EDIT_UNDO**) and Redo (**ID_EDIT_REDO**), is also left to you. If your application does not support these commands, you can easily delete them from your resource file using App Studio.

In the Next Chapter

So far you have seen how the framework creates its major component objects. In Chapter 3, you will see how the framework dispatches Windows messages—including "commands," a new category of messages introduced by the Microsoft Foundation Class Library—to those objects and how the objects "handle" the messages and commands to do the application's work.

CHAPTER 3

Working with Messages and Commands

Chapter 2 introduced the major objects in a running framework application written with the Microsoft Foundation Class Library. This chapter describes how messages and commands are processed by the framework and how you connect them to their handler functions using the ClassWizard tool. Topics covered include:

- Messages and commands
- Message categories
- How the framework calls a message handler
- Message maps
- Managing messages and commands with ClassWizard
- Dynamic update of user-interface objects
- Dynamic display of command information in the status bar

Messages and Commands in the Framework

Applications written for Microsoft Windows are "message driven." In response to events such as mouse clicks, keystrokes, window movements, and so on, Windows sends messages to the proper window. Framework applications process Windows messages like any other application for Windows. But the framework also provides some enhancements that make processing messages easier, more maintainable, and better encapsulated.

The following sections introduce the key terms used in the rest of the chapter to discuss messages and commands.

Messages

The message loop in the **Run** member function of class **CWinApp** retrieves queued messages generated by various events. For example, when the user clicks the mouse, Windows sends several mouse-related messages, such as

WM_LBUTTONDOWN when the left mouse button is pressed and **WM_LBUTTONUP** when the left mouse button is released. The framework's implementation of the application message loop dispatches the message to the appropriate window.

The important categories of messages are described in "Message Categories" later on this page.

Message Handlers

In the Microsoft Foundation Class Library, a dedicated "handler" function processes each separate message. Message-handler functions are member functions of a class. This manual uses the terms "message-handler member function," "message-handler function," "message handler," and "handler" interchangeably.

Writing message handlers accounts for a large proportion of your work in writing a framework application. This chapter describes how the message-processing mechanism works.

What does the handler for a message do? The answer is that it does whatever you want done in response to that message. ClassWizard will create the handlers for you and allow you to implement them. You can jump directly from the ClassWizard dialog box to the handler function's definition in your source files and fill in the handler's code using the Visual Workbench editor. Or you can create all of your handlers with ClassWizard, then move to the editor to fill in all functions at once. You will learn more about using ClassWizard in "How to Manage Commands and Messages with ClassWizard" on page 65.

You can use all of the facilities of Microsoft Visual C++ and the Microsoft Foundation Class Library to write your handlers. For a list of all classes, see Chapter 1.

Message Categories

What kinds of messages do you write handlers for? There are three main categories:

1. Windows messages

   This includes primarily those messages beginning with the **WM_** prefix, except for **WM_COMMAND**. Windows messages are handled by windows and views. These messages often have parameters that are used in determining how to handle the message.

2. Control notifications

This includes **WM_COMMAND** notification messages from controls, including VBX control events from Microsoft Visual Basic™–compatible controls, and other child windows to their parent windows. For example, an edit control sends its parent a **WM_COMMAND** message containing the **EN_CHANGE** control-notification code when the user has taken an action that may have altered text in the edit control. The window's handler for the message responds to the notification message in some appropriate way, such as retrieving the text in the control. VBX notification messages are identified by **VBN_** identifiers.

The framework routes control-notification messages like other **WM_** messages. One exception, however, is the **BN_CLICKED** control-notification message sent by buttons when the user clicks them. This message is treated specially as a command message and routed like other commands.

3. Command messages

   This includes **WM_COMMAND** notification messages from user-interface objects: menus, toolbar buttons, and accelerator keys. The framework processes commands differently from other messages, and they can be handled by more kinds of objects, as explained below.

Windows Messages and Control-Notification Messages

Messages in categories 1 and 2 are handled by windows: objects of classes derived from class **CWnd**. This includes **CFrameWnd**, **CMDIFrameWnd**, **CMDIChildWnd**, **CView**, **CDialog**, and your own classes derived from these base classes. Such objects encapsulate an **HWND**, a handle to a Windows window.

Command Messages

Messages in category 3—commands—can be handled by a wider variety of objects: documents, document templates, and the application object itself in addition to windows and views. When a command directly affects some particular object, it makes sense to have that object handle the command. For example, the Open command on the File menu is logically associated with the application: the application opens a specified document upon receiving the command. So the handler for the Open command is a member function of the application class. You will learn more about commands and how they are routed to objects in "How the Framework Calls a Handler" on page 56.

Message Maps

Each framework class that can receive messages or commands has its own "message map." The framework uses message maps to connect messages and commands to their handler functions. Any class derived from class **CCmdTarget** can have a message map. Later sections of this chapter explain message maps in detail and describe how to use them.

In spite of the name "message map," message maps handle both messages and commands—all three categories of messages listed in "Message Categories" on page 52.

User-Interface Objects and Command IDs

Menu items, toolbar buttons, and accelerator keys are "user-interface objects" capable of generating commands. Each such user-interface object has an ID. You associate a user-interface object with a command by assigning the same ID to the object and the command. As you have seen, commands are implemented as special messages. Figure 3.1 shows how the framework manages commands.

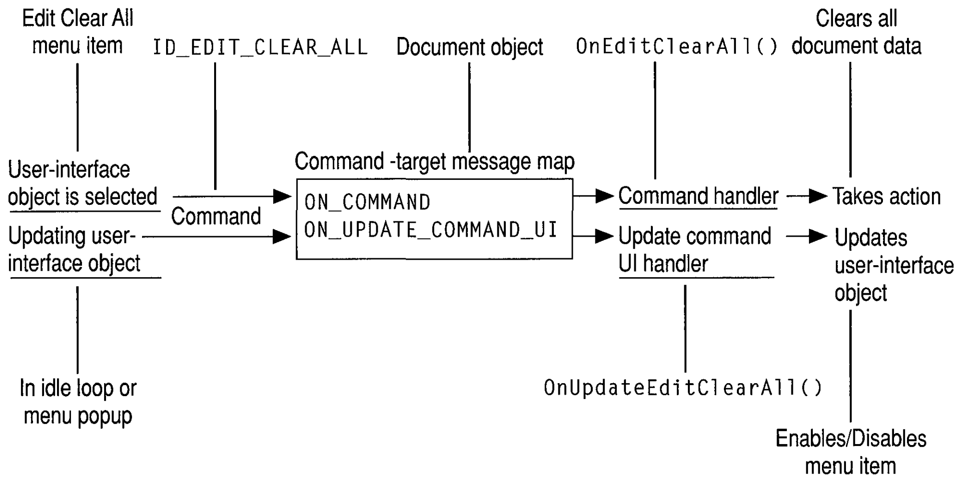


Figure 3.1 Commands in the Framework

Command IDs

A command is fully described by its command ID alone (encoded in the **WM_COMMAND** message). This ID is assigned to the user-interface object that generates the command. Typically, IDs are named for the functionality of the user-interface object they are assigned to.

For example, a Clear All item in the Edit menu might be assigned an ID such as **ID_EDIT_CLEAR_ALL**. The class library predefines some IDs, particularly for commands that the framework handles itself, such as **ID_EDIT_CLEAR_ALL** or **ID_FILE_OPEN**. You will create other command IDs yourself.

When you create your own menus in App Studio, it is a good idea to follow the class library's naming convention as illustrated by **ID_FILE_OPEN**. The next section explains the standard commands defined by the class library.

Standard Commands

The framework defines many standard command messages. The IDs for these commands typically take the form:

ID_*Source_Item*

where *Source* is usually a menu name and *Item* is a menu item. For example, the command ID for the New command on the File menu is **ID_FILE_NEW**. Standard command IDs are shown in bold type in the documentation. Programmer-defined IDs are shown in monotype.

The following is a list of some of the most important commands supported:

File Menu Commands
New, Open, Close, Save, Save As, Page Setup, Print Setup, Print, Print Preview, Exit, and most-recently-used files.

Edit Menu Commands
Clear, Clear All, Copy, Cut, Find, Paste, Repeat, Replace, Select All, Undo, and Redo.

View Menu Commands
Toolbar and Status Bar.

Window Menu Commands
New, Arrange, Cascade, Tile Horizontal, Tile Vertical, and Split.

Help Menu Commands
Index, Using Help, and About.

Object Linking and Embedding (OLE) Commands (Edit Menu)
Insert New Object, Edit Links, Paste Link, Paste Special, and *typename* Object (verb commands).

The framework provides varying levels of support for these commands. Some commands are supported only as defined command IDs, while others are supported with thorough implementations. For example, the framework implements the Open command on the File menu by creating a new document object, displaying an Open dialog box, and opening and reading the file. In contrast, you must implement commands on the Edit menu yourself, since commands like **ID_EDIT_COPY** depend on the nature of the data you are copying.

For more information about the commands supported and the level of implementation provided, see Technical Note 22 in MSVC\HELP\MFCNOTES.HLP. The standard commands are defined in file AFXRES.H.

Command Targets

Figure 3.1 shows the connection between a user-interface object, such as a menu item, and the handler function that the framework calls to carry out the resulting command when the object is clicked.

Windows sends messages that are not command messages directly to a window whose handler for the message is then called. However, the framework routes commands to a number of candidate objects—called "command targets"—one of which normally invokes a handler for the command. The handler functions work the same way for both commands and standard Windows messages, but the mechanism by which they are called is different, as explained in "How the Framework Calls a Handler" below.

How the Framework Calls a Handler

This section first examines how the framework routes commands, then examines how other messages and control notifications are sent to windows.

Message Sending and Receiving

Consider the sending part of the process and how the framework responds.

Most messages result from user interaction with the program. Commands are generated by mouse clicks in menu items or toolbar buttons or by accelerator keystrokes. The user also generates Windows messages by, for example, moving or resizing a window. Other Windows messages are sent when events such as program startup or termination occur, as windows get or lose the focus, and so on. Control-notification messages are generated by mouse clicks or other user interactions with a control, such as a button or list-box control in a dialog box. VBX events are generated by user interactions with VBX controls.

The **Run** member function of class **CWinApp** retrieves messages and dispatches them to the appropriate window. Most command messages are sent to the main frame window of the application. The **WindowProc** predefined by the class library gets the messages and routes them differently, depending on the category of message received.

Now consider the receiving part of the process.

The initial receiver of a message must be a window object. Windows messages are usually handled directly by that window object. Command messages, usually originating in the application's main frame window, get routed to the command-target chain described in "Command Routing" on page 57.

Each object capable of receiving messages or commands has its own message map that pairs a message or command with the name of its handler.

When a command-target object receives a message or command, it searches its message map for a match. If it finds a handler for the message, it calls the handler. For more information about how message maps are searched, see "How the Framework Searches Message Maps" on page 60. Refer again to Figure 3.1 on page 54.

How Noncommand Messages Reach Their Handlers

Unlike commands, standard Windows messages do not get routed through a chain of command targets but are usually handled by the window to which Windows sends the message. The window might be a main frame window, an MDI child window, a standard control, a dialog box, a view, or some other kind of child window.

At run time, each Windows window is attached to a window object (derived from **CWnd**) that has its own associated message map and handler functions. The framework uses the message map—as for a command—to map incoming messages to handlers.

Command Routing

Your responsibility in working with commands is limited to making message-map connections between commands and their handler functions, a task for which you use ClassWizard. You must also write most command handlers.

All messages are usually sent to the main frame window, but command messages are then routed on to other objects. The framework routes commands through a standard sequence of command-target objects, one of which is expected to have a handler for the command. Each command-target object checks its message map to see if it can handle the incoming message.

Different command-target classes check their own message maps at different times. Typically, a class routes the command to certain other objects to give them first chance at the command. If none of those objects handles the command, the original class checks its own message map. Then, if it can't supply a handler itself, it may route the command to yet more command targets. Table 3.1, on the next page, shows how each of the classes structures this sequence. The general order in which a command target routes a command is:

1. To its currently active child command-target object
2. To itself
3. To other command targets

How expensive is this routing mechanism? Compared to what your handler does in response to a command, the cost of the routing is low. Bear in mind that the framework generates commands only when the user interacts with a user-interface object.

Table 3.1 Standard Command Route

When an object of this type receives a command . . .	. . . it gives itself and other command-target objects a chance to handle the command in this order:
MDI frame window (**CMDIFrameWnd**)	1. Active **CMDIChildWnd**
	2. This frame window
	3. Application (**CWinApp** object)
Document frame window (**CFrameWnd**, **CMDIChildWnd**)	1. Active view
	2. This frame window
	3. Application (**CWinApp** object)
View	1. This view
	2. Document attached to the view
Document	1. This document
	2. Document template attached to the document
Dialog box	1. This dialog box
	2. Window that owns the dialog box
	3. Application (**CWinApp** object)

Where numbered entries in the second column of Table 3.1 mention other objects, such as a document, see the corresponding item in the first column. For instance, when you read in the second column that the view forwards a command to its document, see the "Document" entry in the first column to follow the routing further.

An Example

To illustrate, consider a command message from a Clear All menu item in an MDI application's Edit menu. Suppose the handler function for this command happens to be a member function of the application's document class. Here's how that command reaches its handler after the user chooses the menu item:

1. The main frame window receives the command message first.
2. The main MDI frame window gives the currently active MDI child window a chance to handle the command.
3. The standard routing of an MDI child frame window gives its view a chance at the command before checking its own message map.

4. The view checks its own message map first, but, finding no handler, the view next routes the command to its associated document.
5. The document checks its message map and finds a handler. This document member function is called and the routing stops.

If the document did not have a handler, it would next route the command to its document template. Then the command would return to the view and then the frame window. Finally, the frame window would check its message map. If that check failed as well, the command would be routed back to the main MDI frame window and then to the application object—the ultimate destination of unhandled commands.

OnCmdMsg

To accomplish this routing of commands, each command target calls the **OnCmdMsg** member function of the next command target in the sequence. Command targets use **OnCmdMsg** to determine whether they can handle a command and to route it to another command target if they cannot handle it.

Each command-target class may override the **OnCmdMsg** member function. The overrides let each class route commands to a particular next target. A frame window, for example, always routes commands to its current child window or view, as shown in Table 3.1 on page 58.

The default **CCmdTarget** implementation of **OnCmdMsg** uses the message map of the command-target class to search for a handler function for each command message it receives—in the same way that standard messages are searched. If it finds a match, it calls the handler. Message-map searching is explained in the section "How the Framework Searches Message Maps" on page 60.

Overriding the Standard Routing

In rare cases when you must implement some variation of the standard framework routing, you can override it. The idea is to change the routing in one or more classes by overriding **OnCmdMsg** in those classes. Do so:

- In the class that breaks the order to pass to a nondefault object.
- In the new nondefault object or in command targets it might in turn pass commands to.

If you insert some new object into the routing, its class must be a command-target class. In your overriding versions of **OnCmdMsg**, be sure to call the version that you're overriding. See the **OnCmdMsg** member function of class **CCmdTarget** and the versions in such classes as **CView** and **CDocument** in the supplied source code for examples.

How the Framework Searches Message Maps

The framework searches the message-map table for matches with incoming messages. Once you use ClassWizard to write a message-map entry for each message you want a class to handle and to write the corresponding handlers, the framework calls your handlers automatically.

Where to Find Message Maps

When you create a new skeleton application with AppWizard, AppWizard writes a message map for each command-target class it creates for you. This includes your derived application, document, view, and frame-window classes. Some of these message maps already have AppWizard-supplied entries for certain messages and predefined commands, and some are just placeholders for handlers that you will add.

A class's message map is located in the .CPP file for the class. Working with the basic message maps that AppWizard creates, you use ClassWizard to add entries for the messages and commands that each class will handle. A typical message map might look like the following after you add some entries:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
   //{{AFX_MSG_MAP(CMyView)
   ON_WM_MOUSEACTIVATE()
   ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
   ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
   ON_BN_CLICKED(ID_MY_BUTTON, OnMyButton)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

The message map consists of a collection of macros. Two macros, **BEGIN_MESSAGE_MAP** and **END_MESSAGE_MAP**, bracket the message map. Other macros, such as **ON_COMMAND**, fill in the message map's contents. You will learn more about these macros in the sections to come.

Note The message-map macros are not followed by semicolons.

The message map also includes comments of the form

```
//{{AFX_MSG_MAP(CMyView)
//}}AFX_MSG_MAP
```

that bracket many of the entries (not necessarily all). ClassWizard uses these special comments when it writes entries for you. All ClassWizard entries go between the comment lines.

When you use ClassWizard to create a new class, it provides a message map for the class. Alternatively, you can create a message map manually using the Visual Workbench editor.

Derived Message Maps

During message handling, checking a class's own message map is not the end of the message-map story. What happens if class `CMyView` (derived from **CView**) has no matching entry for a message?

Keep in mind that **CView**, the base class of `CMyView`, is derived in turn from **CWnd**. Thus `CMyView` *is* a **CView** and *is* a **CWnd**. Each of those classes has its own message map. Figure 3.2 shows the hierarchical relationship of the classes, but keep in mind that a `CMyView` object is a single object that has the characteristics of all three classes.

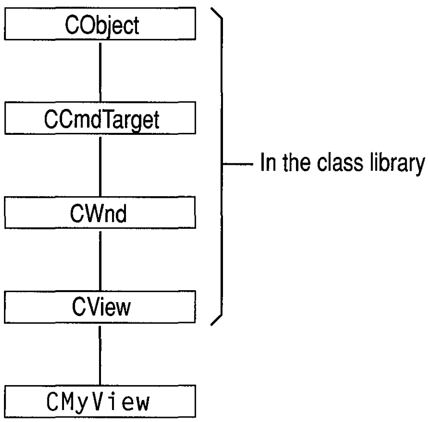


Figure 3.2 A View Hierarchy

So if a message can't be matched in class `CMyView`'s message map, the framework also searches the message map of its immediate base class. The **BEGIN_MESSAGE_MAP** macro at the start of the message map specifies two class names as its arguments:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
```

The first argument names the class to which the message map belongs. The second argument provides a connection with the immediate base class—**CView** here—so the framework can search its message map too.

The message handlers provided in a base class are thus inherited by the derived class. This is very similar to normal virtual member functions without needing to make all handler member functions virtual.

If no handler is found in any of the base-class message maps, default processing of the message is performed. If the message is a command, the framework routes it to the next command target. If it is a standard Windows message, the message is passed to the appropriate default window procedure.

To speed message-map matching, the framework caches recent matches on the likelihood that it will receive the same message again. One consequence of this is that the framework processes unhandled messages quite efficiently. Message maps are also more space-efficient than implementations that use virtual functions.

Message-Map Entries

In your source files, a message map consists of a sequence of predefined macros. The macros inside the message map are called "entry macros." The entry macros used in a message map depend upon the category of the message to be handled. The following sample shows a message map with several common entries (given in the same order as the items in Table 3.2):

```
BEGIN_MESSAGE_MAP(CMyView, CView)
   //{{AFX_MSG_MAP(CMyView)
   ON_WM_MOUSEACTIVATE()
   ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
   ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
   ON_BN_CLICKED(ID_MY_BUTTON, OnMyButton)
   ON_MESSAGE(WM_MYMESSAGE, OnMyMessage)
   ON_REGISTERED_MESSAGE(WM_FIND, OnFind)
   ON_VBXEVENT(VBN_CLICK, IDC_MYBUTTON, OnClickedMyButton)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Table 3.2 summarizes the various kinds of entries. Each entry consists of a macro with zero or more arguments. The macros are predefined by the class library. For examples of the macros, see the message map above.

Table 3.2 Message-Map Entry Macros

Message Type	Macro Form	Arguments
Predefined Windows messages	**ON_WM_XXXX**	None
Commands	**ON_COMMAND**	Command ID, Handler name
Update commands	**ON_UPDATE_COMMAND_UI**	Command ID, Handler name
Control notifications	**ON_XXXX**	Control ID, Handler name

Table 3.2 Message-Map Entry Macros *(continued)*

Message Type	Macro Form	Arguments
User-defined message	**ON_MESSAGE**	User-defined message ID, Handler name (see Technical Note 6 in MFCNOTES.HLP)
Registered Windows message	**ON_REGISTERED_MESSAGE**	Registered message ID variable, Handler name (see Technical Note 6 in MFCNOTES.HLP)
VBX control event	**ON_VBXEVENT**	Event-registration variable (VBN_XXX), Control ID, Handler name (see Technical Note 27 in MFCNOTES.HLP)

Names in the table with the notation **_XXX** represent groups of messages whose names are based on standard message names or control-notification codes in Windows. For example: **ON_WM_PAINT**, **ON_WM_LBUTTONDOWN**, **ON_EN_CHANGE**, **ON_LB_GETSEL**. Even though the **ON_WM_XXX** macros take no arguments, the corresponding handler functions often do take arguments, passed to them by the framework.

Declaring Handler Functions

Certain rules and conventions govern the names of your message-handler functions. These depend on the message category.

Standard Windows Messages

Default handlers for standard Windows messages (**WM_**) are predefined in class **CWnd**. The class library bases names for these handlers on the message name. For example, the handler for the **WM_PAINT** message is declared in **CWnd** as:

```
afx_msg void OnPaint();
```

The **afx_msg** keyword suggests the effect of the C++ **virtual** keyword by distinguishing the handlers from other **CWnd** member functions. Note, however, that these functions are not actually virtual; they are instead implemented through message maps. Message maps depend solely on standard preprocessor macros, not on any extensions to the C++ language. The **afx_msg** keyword resolves to white space after preprocessing.

To override a handler defined in a base class, simply use ClassWizard to define a function with the same prototype in your derived class and to make a message-map entry for the handler. Your handler "overrides" any handler of the same name in any of your class's base classes.

In some cases, your handler should call the overridden handler in the base class so the base class(es) and Windows can operate on the message. Where you call the base-class handler in your override depends on the circumstances. Sometimes you must call the base-class handler first and sometimes last. Sometimes you call the base-class handler conditionally, if you choose not to handle the message yourself. Sometimes you should call the base-class handler, then conditionally execute your own handler code, depending on the value or state returned by the base-class handler.

Important It is not safe to modify the arguments passed into a handler if you intend to pass them to a base-class handler. For example, you might be tempted to modify the *nChar* argument of the `OnChar` handler (to convert to uppercase, for example). This behavior is fairly obscure, but if you need to accomplish this effect, use the **CWnd** member function **SendMessage** instead.

How do you determine the proper way to override a given message? ClassWizard helps with this decision. When ClassWizard writes the skeleton of the handler function for a given message—an `OnCreate` handler for **WM_CREATE**, for example—it sketches in the form of the recommended overridden member function. The following example recommends that the handler first call the base-class handler and proceed only on condition that it does not return –1.

```
int CMyView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    // TODO: Add your specialized creation code here
    return 0;
}
```

By convention the names of these handlers begin with the prefix "On." Some of these handlers take no arguments, while others take several. Some also have a return type other than **void**. The default handlers for all **WM_** messages are documented in the reference as member functions of class **CWnd** whose names begin with "On." The member function declarations in **CWnd** are prefixed with **afx_msg**.

Commands and Control Notifications

There are no default handlers for commands or control-notification messages. Therefore, you are bound only by convention in naming your handlers for these categories of messages. When you map the command or control notification to a handler, ClassWizard proposes a name based on the command ID or control-notification code. You can accept the proposed name, change it, or replace it.

Convention suggests that you name handlers in both categories for the user-interface object they represent. Thus a handler for the Cut command on the Edit menu might be named

```
afx_msg void OnEditCut();
```

Because the Cut command is so commonly implemented in applications, the framework predefines the command ID for the Cut command as **ID_EDIT_CUT**. For a list of all predefined command IDs, see the file AFXRES.H. For more information, see "Standard Commands" on page 54.

In addition, convention suggests a handler for the **BN_CLICKED** notification message from a button labeled "Use As Default" might be named

```
afx_msg void OnClickedUseAsDefault();
```

You might assign this command an ID of `IDC_USE_AS_DEFAULT` since it is equivalent to an application-specific user-interface object.

Both categories of messages take no arguments and return no value.

How to Manage Commands and Messages with ClassWizard

Now that you have seen how messages and commands work, it is time to see how easy it is to manage them with ClassWizard. This section briefly describes the process.

Since a framework application must handle many messages—with the handlers distributed among numerous windows and views, and even documents and other objects—the job of making and maintaining all the connections is demanding.

For that reason, Visual C++ provides ClassWizard, a tool designed specifically to connect Windows messages and user-interface objects such as menus to their handlers. Figure 3.3 shows ClassWizard being used to map a message to a handler.

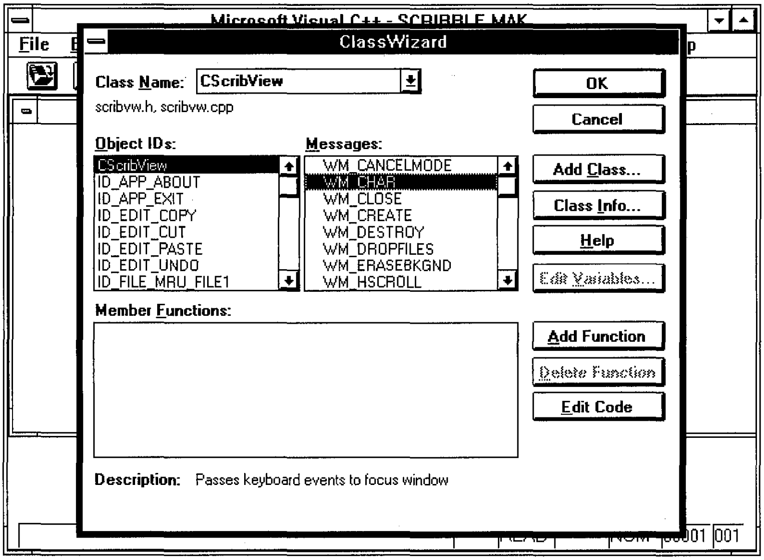


Figure 3.3 ClassWizard

The typical development scenarios are as follows:

- You determine that one of your classes must handle a certain Windows message, so you invoke ClassWizard and make the connection.
- You create a menu or accelerator resource in App Studio, then invoke ClassWizard to connect the command associated with that object to a handler.

As you work with the framework, you'll find that ClassWizard greatly simplifies your message-management tasks.

ClassWizard writes the following information to your source files:

- The appropriate message-map entry for the connection
- A declaration of the handler as a member function of the class
- An empty function template for you to fill in with the handler's code

You can invoke ClassWizard from App Studio while you're editing menus, accelerators, toolbars, or dialog boxes. Or you can invoke it from Visual Workbench while you're working on source code files with the editor. For detailed information about using ClassWizard to connect messages to handlers, see Chapter 13 in the *Visual Workbench User's Guide* and Chapter 9 in the *App Studio User's Guide*. For examples, see Chapters 6 and 7 in the *Class Library User's Guide*.

Important Use ClassWizard to create and edit all message-map entries. If you add them manually, you may not be able to edit them with ClassWizard later. If you add them outside the bracketing comments, `//{{AFX_MSG_MAP(classname)` and `//}}AFX_MSG_MAP`, ClassWizard cannot edit them at all. Note that by the same token ClassWizard will not touch any entries you add outside the comments, so feel free to add messages outside the comments if you do not want them to be modified.

How to Update User-Interface Objects

Typically, menu items and toolbar buttons have more than one state. For example, a menu item is grayed (dimmed) if it is unavailable in the present context. Menu items can also be checked or unchecked. A toolbar button can also be disabled if unavailable, or be checked.

Who updates the state of these items as program conditions change? Logically, if a menu item generates a command that is handled by, say, a document, it makes sense to have the document update the menu item. The document probably contains the information on which the update is based.

If a command has multiple user-interface objects (perhaps a menu item and a toolbar button), both are routed to the same handler function. This encapsulates your user-interface update code for all of the equivalent user-interface objects in a single place.

The framework provides a convenient interface for automatically updating user-interface objects. You can choose to do the updating in some other way, but the interface provided is efficient and easy to use.

When Update Handlers are Called

Suppose the user clicks the mouse in the File menu, which generates a **WM_INITMENUPOPUP** message. The framework's update mechanism collectively updates all items on the File menu before the menu drops down so the user can see it.

To do this, the framework routes update commands for all menu items in the pop-up menu along the standard command routing. Command targets on the routing have an opportunity to update any menu items by matching the update command with an appropriate message-map entry (of the form **ON_UPDATE_COMMAND_UI**) and calling an "update handler" function. Thus, for a menu with six menu items, six update commands are sent out. If an update handler exists for the command ID of the menu item, it is called to do the updating. If not, the framework checks for the existence of a handler for that command ID and enables or disables the menu item as appropriate.

If the framework does not find an **ON_UPDATE_COMMAND_UI** entry during command routing, it automatically enables the user-interface object if there is an **ON_COMMAND** entry somewhere with the same command ID. Otherwise, it disables the user-interface object. Therefore, to ensure that a user-interface object is enabled, supply a handler for the command the object generates or supply an update handler for it. See Figure 3.1 on page 54.

It is possible to disable the default disabling of user-interface objects. For more information, see the **m_bAutoMenuEnable** member of class **CFrameWnd**.

Menu initialization is automatic in the framework, occurring when the application receives a **WM_INITMENUPOPUP** message. During the idle loop, the framework searches the command routing for button update handlers in much the same way as it does for menus.

The ON_UPDATE_COMMAND_UI Macro

Use ClassWizard to connect a user-interface object to a command-update handler in a command-target object. It will automatically connect the user-interface object's ID to the **ON_UPDATE_COMMAND_UI** macro and create a handler in the object that will handle the update.

For example, the Scribble tutorial in the *Class Library User's Guide* updates a Clear All command in its Edit menu. In the tutorial, ClassWizard adds a message-map entry in the chosen class, a function declaration for a command-update handler called `OnUpdateEditClearAll` in the class declaration, and an empty function template in the class's implementation file. The function prototype looks like this:

```
afx_msg void OnUpdateEditClearAll( CCmdUI* pCmdUI );
```

Like all handlers, the function shows the **afx_msg** keyword. Like all update handlers, it takes one argument, a pointer to a **CCmdUI** object.

The CCmdUI Class

When it routes the update command to the handler, the framework passes the handler a pointer to a **CCmdUI** object (or to an object of a **CCmdUI**-derived class). This object represents the menu item or toolbar button or other user-interface object that generated the command. The update handler calls member functions of the **CCmdUI** structure through the pointer to update the user-interface object. For example, here is an update handler for the Clear All menu item:

```
void CMyClass::OnUpdateToolsMyTool( CCmdUI* pCmdUI )
{
    if( ToolAvailable() )
        pCmdUI->Enable( TRUE );
}
```

This handler calls the **Enable** member function of an object with access to the menu item. **Enable** makes the item available for use.

How to Display Command Information In the Status Bar

When you run AppWizard to create the skeleton of your application, you can easily support a toolbar and a status bar. A single option in AppWizard supports both together. When a status bar is present, the framework automatically gives helpful feedback as the user of your application moves the mouse through items in the menus. The framework automatically displays a prompt string in the status bar when the menu item is being selected. For example, when the user drags the mouse over the Cut item in the Edit menu, the framework might display "Cut the selection and put it on the Clipboard" in the message area of the status bar. The prompt helps the user grasp the menu item's purpose. This also works when the user clicks on a toolbar button. Figure 3.4 shows a status bar displaying a command prompt string.

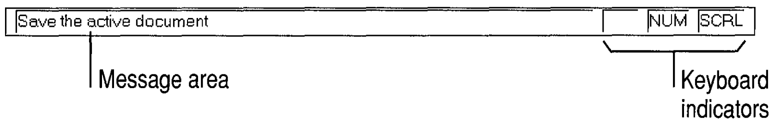


Figure 3.4 A Command Prompt in the Status Bar

You can easily add to this status-bar help by defining prompt strings for the menu items that you add to the program. To do so, provide the prompt strings when you edit the properties of the menu item in App Studio. The strings you define this way are stored in your application's resource file; they have the same IDs as the commands they explain.

By default, AppWizard adds the ID for a standard prompt, "Ready," which is displayed when the program is waiting for new messages. If you specify the Context-Sensitive Help option in AppWizard, the ID for a help prompt, "For Help, press F1," is added to your application. This ID is **AFX_IDS_IDLEMESSAGE**.

In the Next Chapter

So far you have seen how the framework creates its major component objects and how those objects communicate through Windows messages and user-initiated commands. In Chapter 4, you will learn more about documents, views, frame windows, drawing, and printing.

CHAPTER 4

Working with Frame Windows, Documents, and Views

Previous chapters introduced the primary objects in an application built upon the framework of the Microsoft Foundation Class Library and showed how these objects communicate via messages and commands.

This chapter takes you deeper into three of the most important objects in a framework application:

- Frame windows, which contain and manage your views
- Documents, which define your application's data
- Views, which display your documents and manage user interaction with them

The chapter also explains how the framework manages printing and print preview since printing functionality is intimately tied to the view.

One of the most important features of the framework is the division of labor among frame windows, documents, and views. The document manages your data. The view displays it and takes user input. And the frame window puts a frame around the view. Code that defines and manipulates data resides in the document class. Code that displays the data and interprets user input resides in the view class.

Frame Windows

When an application runs under Microsoft Windows, the user interacts with documents displayed in frame windows. A document frame window has two major components: the frame and the contents that it frames. A document frame window can be a single document interface (SDI) frame window or a multiple document interface (MDI) child window. The Windows operating system manages most of the user's interaction with the frame window: moving and resizing the window, closing it, minimizing and maximizing it. You manage the contents inside the frame.

The framework uses frame windows to contain views. The two components—frame and contents—are represented and managed by two different classes in the Microsoft Foundation Class Library. A frame window class manages the frame, and a view class manages the contents. The view window is a child of the frame window. Drawing and other user interaction with the document take place in the view's client area, not the frame window's client area. The frame window provides a visible frame around a view, complete with a caption bar and standard window controls such as a control menu, buttons to minimize and maximize the window, and controls for resizing the window. The "contents" consist of the window's client area, which is fully occupied by a child window—the view. Figure 4.1 shows the relationship between a frame window and a view.

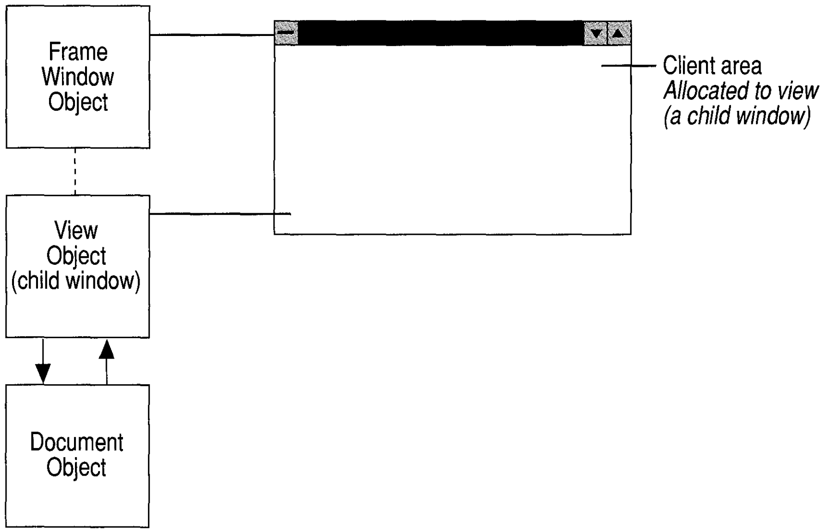


Figure 4.1 Frame Window and View

Later, the chapter discusses splitter windows. In a splitter window, the frame window's client area is occupied by a splitter window, which in turn has multiple child windows, called panes, which are views.

This section explains what you need to know about frame windows. Topics covered include:

- The frame window classes created by AppWizard
- Managing child windows
- Managing the current view
- Managing menus, control bars, and accelerators
- Working with the File Manager
- Orchestrating other window actions

Window Classes

Each application has one "main frame window," a desktop window that usually has the application name in its caption. Each document usually has one "document frame window." A document frame window contains at least one view, which presents the document's data. For an SDI application, there is one frame window derived from class **CFrameWnd**. This window is both the main frame window and the document frame window. For an MDI application, the main frame window is derived from class **CMDIFrameWnd**, and the document frame windows, which are MDI child windows, are derived from class **CMDIChildWnd**.

These classes provide most of the frame window functionality you will need for your applications. Under normal circumstances, the default behavior and appearance they provide will suit your needs. If you need additional functionality, derive from these classes.

The Frame Window Classes Created by AppWizard

When you use AppWizard to create a skeleton application, in addition to application, document, and view classes, AppWizard creates a derived frame-window class for your application's main frame window. The class is called `CMainFrame` by default, and the files that contain it are named MAINFRM.H and MAINFRM.CPP.

If your application is SDI, your `CMainFrame` class is derived from class **CFrameWnd**. If your application is MDI, `CMainFrame` is derived from class **CMDIFrameWnd**. If you choose to support a toolbar, the class also has member variables of type **CToolBar** and **CStatusBar** and an `OnCreate` message-handler function to initialize the two control bars.

If your application is MDI, AppWizard does not derive a new document frame window class for you. Instead, it uses the default implementation in **CMDIChildWnd**. Later on, if you find you need to customize your document frame window, you can use ClassWizard to create a new document frame window class.

These frame window classes work as created, but to enhance their functionality, you must add member variables and member functions. You may also want to have your window classes handle other Windows messages.

Using Frame Windows

The framework creates document frame windows—and their views and documents—as part of its implementation of the New and Open commands on the File menu. Because the framework does most of the frame window work for you, you play only a small role in creating, using, and destroying those windows. You can, however, explicitly create your own frame windows and child windows for special purposes.

Creating Document Frame Windows

As you saw earlier, in "Document/View Creation" in Chapter 2, the **CDocTemplate** object orchestrates creating the frame window, document, and view and connecting them all together. Three **CRuntimeClass** arguments to the **CDocTemplate** constructor specify the frame window, document, and view classes that the document template creates dynamically in response to user commands such as the New command on the File menu or the New Window command on an MDI Window menu. The document template stores this information for later use when it creates a frame window for a view and document.

In order for the **RUNTIME_CLASS** mechanism to work correctly, your derived frame-window classes must be declared with the **DECLARE_DYNCREATE** macro. This is because the framework needs to create document frame windows using the dynamic construction mechanism of class **CObject**. For details about **DECLARE_DYNCREATE**, see the "Macros and Globals" section in Part 2 and Chapter 12 in the *Class Library User's Guide*.

When the user chooses a command that creates a document, the framework calls upon the document template to create the document object, its view, and the frame window that will display the view. Chapter 2 described this creation process. When it creates the document frame window, the document template creates an object of the appropriate class—a class derived from **CFrameWnd** for an SDI application or from **CMDIChildWnd** for an MDI application. The framework then calls the frame window object's **LoadFrame** member function to get creation information from resources and to create the Windows window. The framework attaches the window handle to the frame-window object. Then it creates the view as a child window of the document frame window.

Note You cannot create your own child windows or call any Windows application programming interface (API) functions in the constructor of a **CWnd**-derived object. This is because the **HWND** for the **CWnd** object has not been created yet. Most Windows-specific initialization, such as adding child windows, must be done in an **OnCreate** message handler.

Destroying Frame Windows

The framework manages window destruction as well as creation for those windows associated with framework documents and views. If you create additional windows, you are responsible for destroying them.

In the framework, when the user closes the frame window, the window's default **OnClose** handler calls **DestroyWindow**. The last member function called when the Windows window is destroyed is **OnNcDestroy**, which does some cleanup, calls the **Default** member function to perform Windows cleanup, and lastly calls the virtual member function **PostNcDestroy**. The **CFrameWnd** implementation of

PostNcDestroy deletes the C++ window object. You should never use the C++ **delete** operator on a frame window. Use **DestroyWindow** instead.

When the main window closes, the application closes. If there are modified unsaved documents, the framework puts up a message box to ask if the documents should be saved and ensures that the appropriate documents are saved if necessary.

What Frame Windows Do

Besides simply framing a view, frame windows are responsible for numerous tasks involved in coordinating the frame with its view and with the application. **CMDIFrameWnd** and **CMDIChildWnd** inherit from **CFrameWnd**, so they have **CFrameWnd** capabilities as well as new capabilities that they add. Examples of child windows include views, controls such as buttons and list boxes, and control bars, including toolbars, status bars, and dialog bars. The frame window is responsible for managing the layout of its child windows. In the framework, a frame window positions any control bars, views, and other child windows inside its client area. The frame window also forwards commands to its views and can respond to notification messages from control windows. Chapter 2 showed how commands are routed from the frame window to its view and other command targets.

Managing Child Windows

MDI main frame windows (one per application) contain a special child window called the **MDICLIENT** window. The **MDICLIENT** window manages the client area of the main frame window, and itself has child windows: the document windows, derived from **CMDIChildWnd**. Because the document windows are frame windows themselves (MDI child windows), they can also have their own children. In all of these cases, the parent window manages its child windows and forwards some commands to them.

In an MDI frame window, the frame window manages the **MDICLIENT** window, repositioning it in conjunction with control bars. The **MDICLIENT** window, in turn, manages all MDI child frame windows. Figure 4.2 shows the relationship between an MDI frame window, its **MDICLIENT** window, and its child document frame windows.

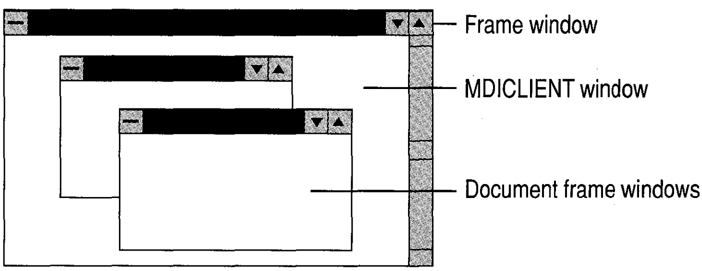


Figure 4.2 MDI Frame Windows and Children

An MDI frame window also works in conjunction with the current MDI child window, if there is one. The MDI frame window delegates command messages to the MDI child before it tries to handle them itself.

Managing the Current View

As part of the default implementation of frame windows, a frame window keeps track of a currently active view. If the frame window contains more than one view, as for example in a splitter window, the current view is the most recent view in use. The active view is independent of the active window in Windows or the current input focus.

When the active view changes, the framework notifies the current view by calling its **OnActivateView** member function. You can tell whether the view is being activated or deactivated by examining **OnActivateView**'s *bActivate* parameter. By default, **OnActivateView** sets the focus to the current view on activation. You can override **OnActivateView** to perform any special processing when the view is deactivated or reactivated. For example, you might want to provide special visual cues to distinguish the active view from other, inactive views. For more information, see the **OnActivateView** member function of class **CView**.

A frame window forwards commands to its current (active) view, as described in Chapter 2, as part of the standard command routing.

Managing Menus, Control Bars, and Accelerators

The frame window manages updating user-interface objects, including menus, toolbar buttons, and the status bar. It also manages sharing the menu bar in MDI applications.

The frame window participates in updating user-interface items using the **ON_UPDATE_COMMAND_UI** mechanism described in Chapter 3. Buttons on toolbars and other control bars are updated during the idle loop. Menu items in drop-down menus on the menu bar are updated just before the menu drops down.

The frame window also positions the status bar within its client area and manages the status bar's indicators. The frame window clears and updates the message area in the status bar as needed and displays prompt strings as the user selects menu items or toolbar buttons, as described in Chapter 3.

For MDI applications, the MDI frame window manages the menu bar and caption. An MDI frame window owns one default menu that is used as the menu bar when there are no active MDI child windows. When there are active children, the MDI frame window's menu bar is taken over by the menu for the active MDI child window. If an MDI application supports multiple document types, such as chart and worksheet documents, each type puts its own menus into the menu bar and changes the main frame window's caption.

CMDIFrameWnd provides default implementations for the standard commands on the Window menu that appears for MDI applications. In particular, the New Window command (**ID_WINDOW_NEW**) is implemented to create a new frame window and view on the current document. You need to override these implementations only if you need advanced customization.

Multiple MDI child windows of the same document type share menu resources. If several MDI child windows are created by the same document template, they can all use the same menu resource, saving on Windows system resources.

Each frame window maintains an optional accelerator table that does keyboard accelerator translation for you automatically. This mechanism makes it easy to define accelerator keys (also called shortcut keys) that invoke menu commands.

Frame Window Styles

The frame windows that you get with the framework are suitable for most programs, but you can gain additional flexibility by using the advanced functions **PreCreateWindow** and **AfxRegisterWindowClass. PreCreateWindow** is a member function of **CWnd. AfxRegisterWindowClass** is a global function documented in "Macros and Globals" in the alphabetic reference.

If you apply the **WS_HSCROLL** and **WS_VSCROLL** styles to the main frame window, they are instead applied to the **MDICLIENT** window so users can scroll the **MDICLIENT** area.

If the window's **FWS_ADDTOTITLE** style bit is set (which it is by default), the view tells the frame window what title to display in the window's title bar based on the view's document name.

Working with the File Manager

The frame window manages a relationship with the Windows File Manager.

By adding a few initializing calls in your override of the **CWinApp** member function **InitInstance**, as described in Chapter 2, you can have your frame window indirectly open files dragged from the Windows File Manager and dropped in the frame window. See "File Manager Drag and Drop" in Chapter 2, on page 32.

The frame window can also respond to dynamic data exchange (DDE) requests to open files from the File Manager (if the file extension is registered or associated with the application). See "Shell Registration" in Chapter 2, on page 32.

Orchestrating Other Window Actions

The frame window orchestrates semimodal states such as context-sensitive help and print preview. The framework's role in managing context-sensitive help is described in Chapter 5. For a description of the frame window's role in print preview, see "Printing and Print Preview" on page 91.

Documents and Views

The parts of the framework most visible both to the user and to you, the programmer, are the document and view. Most of your work in developing an application with the framework goes into writing your document and view classes. This section describes:

- The purposes of documents and views and how they interact in the framework.
- What you must do to implement them.

The **CDocument** class provides the basic functionality for programmer-defined document classes. A document represents the unit of data that the user typically opens with the File Open command and saves with the File Save command.

The **CView** class provides the basic functionality for programmer-defined view classes. A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen and interprets user input as operations upon the document. The view also renders the image for both printing and print preview.

Figure 4.3 shows the relationship between a document and its view.

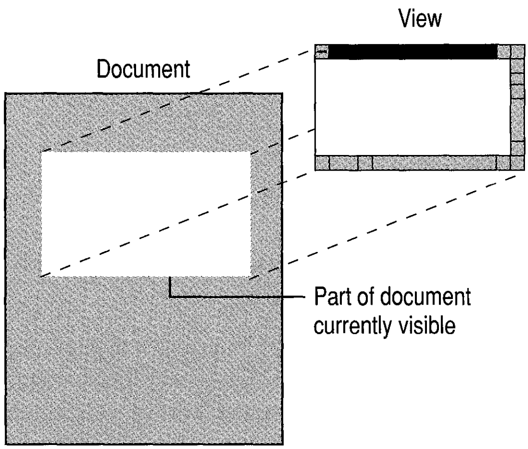


Figure 4.3 Document and View

The document/view implementation in the class library separates the data itself from its display and from user operations on the data. All changes to the data are managed through the document class. The view calls this interface to access and update the data.

Documents, their associated views, and the frame windows that frame the views are created by a document template, as described in "Document/View Creation" on page 34 in Chapter 2. The document template is responsible for creating and managing all documents of one document type.

Document and View Classes Created by AppWizard

AppWizard gives you a head start on your program development by creating skeletal document and view classes for you. You can then use ClassWizard to map commands and messages to these classes and the Visual Workbench editor to write their member functions.

The document class created by AppWizard is derived from class **CDocument**. The view class is derived from **CView**. The names that AppWizard gives these classes and the files that contain them are based on the project name you supply in the AppWizard dialog box. From AppWizard, you can use the Classes dialog box to alter the default names.

Some applications might need more than one document class, view class, or frame window class. For more information, see "Multiple Document Types, Views, and Frame Windows" on page 86.

Using Documents and Views

Working together, documents and views:

- Contain, manage, and display your application-specific data.
- Provide an interface for manipulating the data.
- Participate in writing and reading files.
- Participate in printing.
- Handle most of your application's commands and messages.

Managing Data

Documents contain and manage your application's data. To use the AppWizard-supplied document class, you must do the following:

- Derive a class from **CDocument** for each type of document.
- Add member variables to store each document's data.
- Override **CDocument**'s **Serialize** member function in your document class. **Serialize** writes and reads the document's data to and from disk.

You may also want to override other **CDocument** member functions. In particular, you will often need to override **OnNewDocument** and **OnOpenDocument** to initialize the document's data members and **DeleteContents** to destroy dynamically allocated data. For information about overridable members, see class **CDocument**.

Document Data Variables

Implement your document's data as member variables of your document class. For example, the Scribble tutorial program declares a data member of type **CObList**—a linked list that stores pointers to **CObject** objects. This list is used to store arrays of points that make up a freehand line drawing.

How you implement your document's member data depends on the nature of your application. To help you out, the Microsoft Foundation Class Library supplies a group of "collection classes"—arrays, lists, and maps (dictionaries)—along with classes that encapsulate a variety of common data types such as **CString**, **CRect**, **CPoint**, **CSize**, and **CTime**. For more information about these classes, see Chapter 1.

When you define your document's member data, you will usually add member functions to the document class to set and get data items and perform other useful operations on them.

Your views access the document object by using the view's pointer to the document, installed in the view at creation time. You can retrieve this pointer in a view's member functions by calling the **CView** member function **GetDocument**. Be sure to cast this pointer to your own document type. Then you can access public document members through the pointer.

If frequent data transfer requires direct access, or you wish to use the nonpublic members of the document class, you may want to make your view class a friend of the document class.

Serializing Data to and from Files

The basic idea of persistence is that an object should be able to write its current state, indicated by the values of its member variables, to persistent storage. Later, the object can be recreated by reading, or "deserializing," the object's state from persistent storage. A key point here is that the object itself is responsible for reading and writing its own state. Thus, for a class to be persistent, it must implement the basic serialization operations.

The framework provides a default implementation for saving documents to disk files in response to the Save and Save As commands on the File menu and for loading documents from disk files in response to the Open command. With very little work, you can implement a document's ability to write and read its data to and from a file. The main thing you must do is override **CDocument**'s **Serialize** member function in your document class.

AppWizard places a skeletal override of the **CDocument** member function **Serialize** in the document class it creates for you. After you have implemented your application's member variables, you can fill in your `Serialize` override with code that sends the data to an "archive object" connected to a file. A **CArchive** object is similar to the **cin** and **cout** input/output objects from the C++ iostream library. However, **CArchive** writes and reads binary format, not formatted text.

The Document's Role

The framework responds automatically to the File menu's Open, Save, and Save As commands by calling the document's `Serialize` member function if it is implemented. An **ID_FILE_OPEN** command, for example, calls a handler function in the application object. During this process, the user sees and responds to the File Open dialog box and the framework obtains the filename the user chooses. The framework creates a **CArchive** object set up for loading data into the document and passes the archive to `Serialize`. The framework has already opened the file. The code in your document's `Serialize` member function reads the data in through the archive, reconstructing data objects as needed. For more information about serialization, see Chapter 14 in the *Class Library User's Guide.*

The Data's Role

In general, class-type data should be able to serialize itself. That is, when you pass an object to an archive, the object should know how to write itself to the archive and how to read itself from the archive. The Microsoft Foundation Class Library provides support for making classes serializable in this way. If you design a class to define a data type and you intend to serialize data of that type, design for serialization.

Instructions for defining a serializable class are given in Chapter 14 of the *Class Library User's Guide.*

Bypassing the Archive Mechanism

As you have seen, the framework provides a default way to read and write data to and from files. Serializing through an archive object suits the needs of a great many applications. Such an application reads a file entirely into memory, lets the user update the file, and then writes the updated version to disk again.

However, some applications operate on data very differently, and for these applications serialization through an archive is not suitable. Examples include database programs, programs that edit only parts of large files, and programs that share data files.

In these cases, you can override the **Serialize** member function of **CDocument** in a different way to mediate file actions through a **CFile** object rather than a **CArchive** object.

You can use the **Open**, **Read**, **Write**, **Close**, and **Seek** member functions of class **CFile** to open a file, move the file pointer (seek) to a specific point in the file, read a record (a specified number of bytes) at that point, let the user update the record, then seek to the same point again and write the record back to the file. The framework will open the file for you, and you can use the **GetFile** member function of class **CArchive** to obtain a pointer to the **CFile** object. For even more sophisticated and flexible use, you can override the **OnOpenDocument** and **OnSaveDocument** member functions of class **CWinApp**. For more information, see class **CFile** in the alphabetic reference.

In this scenario, your `Serialize` override does nothing, unless, for example, you want to have it read and write a file header to keep it up to date when the document closes.

For an example of such nonarchived processing, see the CHKBOOK sample program.

Handling Commands in the Document

Your document class may also handle certain commands generated by menu items, toolbar buttons, or accelerator keys. By default, **CDocument** handles the File Save and Save As commands, using serialization. Other commands that affect the data may also be handled by member functions of your document. For example, in the Scribble tutorial program, class `CScribDoc` provides a handler for the Edit Clear All command, which deletes all of the data currently stored in the document. Unlike views, documents cannot handle standard Windows messages.

Displaying Data in a View and Interacting with the User

The view's responsibilities are to display the document's data graphically to the user and to accept and interpret user input as operations on the document. Your tasks in writing your view class are to:

- Write your view class's `OnDraw` member function, which renders the document's data.
- Connect appropriate Windows messages and user-interface objects such as menu items to message-handler member functions in the view class.
- Implement those handlers to interpret user input.

In addition, you may need to override other **CView** member functions in your derived view class. In particular, you may want to override **OnInitialUpdate** to perform special initialization for the view and **OnUpdate** to do any special processing needed just before the view redraws itself. For multipage documents, you also must override **OnPreparePrinting** to initialize the Print dialog box with the number of pages to print and other information. For more information on overriding **CView** member functions, see class **CView**.

The Microsoft Foundation Class Library also provides several derived view classes for special purposes:

- **CScrollView**, which provides automatic scrolling and view scaling.
- **CFormView**, which provides a scrollable view useful for displaying a form made up of dialog controls. A **CFormView** object is created from a dialog-template resource.
- **CEditView**, which provides a view with the characteristics of an editable-text control with enhanced editing features. You can use a **CEditView** object to implement a simple text editor.

To take advantage of these special classes, derive your view classes from them. For more information, see "Scrolling" on page 86 and "Special View Classes" on page 90.

Drawing in a View

Nearly all drawing in your application occurs in the view's `OnDraw` member function, which you must override in your view class. (The exception is mouse drawing, discussed in the next section.) Your `OnDraw` override:

1. Gets data by calling the document member functions you provide.
2. Displays the data by calling member functions of a device-context object that the framework passes to `OnDraw`.

When a document's data changes in some way, the view must be redrawn to reflect the changes. Typically, this happens when the user makes a change through a view on the document. In this case, the view calls the document's **UpdateAllViews** member function to notify all views on the same document to update themselves. **UpdateAllViews** calls each view's **OnUpdate** member function. The default implementation of **OnUpdate** invalidates the view's entire client area. You can override it to invalidate only those regions of the client area that map to the modified portions of the document.

The **UpdateAllViews** member function of class **CDocument** and the **OnUpdate** member function of class **CView** let you pass information describing what parts of the document were modified. This "hint" mechanism lets you limit the area that the view must redraw. **OnUpdate** takes two "hint" arguments. The first, *lHint*, of type **LPARAM**, lets you pass any data you like, while the second, *pHint*, of type **CObject***, lets you pass a pointer to any object derived from **CObject**.

When a view becomes invalid, Windows sends it a **WM_PAINT** message. The view's **OnPaint** handler function responds to the message by creating a device-context object of class **CPaintDC** and calls your view's `OnDraw` member function. You do not normally have to write an overriding `OnPaint` handler function.

Recall from Chapter 2 that a device context is a Windows data structure that contains information about the drawing attributes of a device such as a display or a printer. All drawing calls are made through a device-context object. For drawing on the screen, `OnDraw` is passed a **CPaintDC** object. For drawing on a printer, it is passed a **CDC** object set up for the current printer.

Your code for drawing in the view first retrieves a pointer to the document, then makes drawing calls through the device context. The following simple `OnDraw` example illustrates the process:

```
void CMyView::OnDraw( CDC* pDC )
{
    CMyDoc* pDoc = GetDocument();
    CString s = pDoc->GetData();    // Returns a CString
    CRect rect;

    GetClientRect( &rect );
    pDC->SetTextAlign( TA_BASELINE | TA_CENTER );
    pDC->TextOut( rect.right / 2, rect.bottom / 2,
                  s, s.GetLength() );
}
```

In this example, you would define the `GetData` function as a member of your derived document class.

The example prints whatever string it gets from the document, centered in the view. If the `OnDraw` call is for screen drawing, the **CDC** object passed in *pDC* is a **CPaintDC** whose constructor has already called **BeginPaint**. Calls to drawing functions are made through the device-context pointer. For information about device contexts and drawing calls, see class **CDC** and "Working with Windows" in Chapter 2.

For more examples of how to write `OnDraw`, see MFCSAMP.HLP in MFC.HLP.

Interpreting User Input Through a View

Other member functions of the view handle and interpret all user input. You will usually define message-handler member functions in your view class to:

- Process Windows messages generated by mouse and keyboard actions.
- Process commands from menus, toolbar buttons, and accelerator keys.

These message-handler member functions interpret mouse clicks, drags, double-clicks, and mouse movements; keystrokes; and menu commands as data input, selection, dragging, or other editing operations, including moving data to and from the Clipboard. Which Windows messages your view handles depends on your application's needs.

You saw earlier, in "Messages and Commands in the Framework" on page 51 in Chapter 3, how to assign menu items and other user-interface objects to commands and how to bind the commands to handler functions with ClassWizard. You have also seen how the framework routes such commands and sends standard Windows messages to the objects that contain handlers for them.

For example, your application might need to implement direct mouse drawing in the view. The Scribble tutorial example shows how to handle the **WM_LBUTTONDOWN**, **WM_MOUSEMOVE**, and **WM_LBUTTONUP** messages respectively to begin, continue, and end the drawing of a line segment. On the other hand, you might sometimes need to interpret a mouse click in your view as a selection. Your view's `OnLButtonDown` handler function would determine whether the user was drawing or selecting. If selecting, the handler would determine whether the click was within the bounds of some object in the view and, if so, alter the display to show the object as selected.

Your view might also handle certain menu commands, such as those from the Edit menu to cut, copy, paste, or delete selected data using the Clipboard. Such a handler would call some of the Clipboard-related member functions of class **CWnd** to transfer a selected data item to or from the Clipboard.

Printing and the View

Your view also plays two important roles in printing its associated document. The view:

- Uses the same `OnDraw` code to draw on the printer as to draw on the screen.
- Manages dividing the document into pages for printing.

For more information about printing and about the view's role in printing, see "Printing and Print Preview" on page 91.

Scrolling and Scaling Views

The Microsoft Foundation Class Library supports views that scroll and views that are automatically scaled to the size of the frame window that displays them. Class **CScrollView** supports both kinds of views.

For more information about scrolling and scaling, see class **CScrollView**. For a scrolling example, see Chapter 8, "Enhancing Views," in the *Class Library User's Guide*.

Scrolling

Frequently the size of a document is greater than the size that its view can display. This may occur because the document's data increases or the user shrinks the window that frames the view. In such cases, the view must support scrolling.

Any view can handle scroll-bar messages in its **OnHScroll** and **OnVScroll** member functions. You can either implement scroll-bar message handling in these functions, doing all the work yourself, or you can use the **CScrollView** class to handle scrolling for you.

CScrollView does the following:

- Manages window and viewport sizes and mapping modes
- Scrolls automatically in response to scroll-bar messages

You can specify how much to scroll for a "page" (when the user clicks in a scroll-bar shaft) and a "line" (when the user clicks in a scroll arrow). Plan these values to suit the nature of your view. For example, you might want to scroll in 1-pixel increments for a graphics view but in increments based on the line height in text documents.

Scaling

When you want the view to automatically fit the size of its frame window, you can use **CScrollView** for scaling instead of scrolling. The logical view is stretched or shrunk to fit the window's client area exactly. A scaled view has no scroll bars.

Multiple Document Types, Views, and Frame Windows

The standard relationship among a document, its view, and its frame window was described earlier in "Document/View Creation" on page 34 in Chapter 2. Many applications support a single document type (but possibly multiple open documents of that type) with a single view on the document and only one frame window per document. But some applications may need to alter one or more of those defaults.

Multiple Document Types

AppWizard creates a single document class for you. In some cases, though, you may need to support more than one document type. For example, your application may need worksheet and chart documents. Each document type is represented by its own document class and probably by its own view class as well. When the user chooses the File New command, the framework puts up a dialog box that lists the supported document types. Then it creates a document of the type that the user chooses. Each document type is managed by its own document-template object.

To create extra document classes, use the Add Class button in the ClassWizard dialog box. Choose **CDocument** as the Class Type to derive from and supply the requested document information. Then implement the new class's data.

To let the framework know about your extra document class, you must add a second call to **AddDocTemplate** in your application class's `InitInstance` override. For more information, see "Document Templates" in Chapter 2.

Multiple Views

Many documents require only a single view, but it is possible to support more than one view of a single document. To help you implement multiple views, a document object keeps a list of its views, provides member functions for adding and removing views, and supplies the **UpdateAllViews** member function for letting multiple views know when the document's data has changed.

The Microsoft Foundation Class Library supports three common user interfaces requiring multiple views on the same document. These models are:

- View objects of the same class, each in a separate MDI document frame window.

  You might want to support creating a second frame window on a document. The user could choose a New Window command to open a second frame with a view of the same document and then use the two frames to view different portions of the document simultaneously. The framework supports the New Window command on the Window menu for MDI applications by duplicating the initial frame window and view attached to the document.

- View objects of the same class in the same document frame window.

  Splitter windows split the view space of a single document window into multiple separate views of the document. The framework creates multiple view objects from the same view class. For more information, see the next section, "Splitter Windows."

- View objects of different classes in a single frame window.

  In this model, a variation of the splitter window, multiple views share a single frame window. The views are constructed from different classes, each view providing a different way to view the same document. For example, one view might show a word-processing document in normal mode while the other view shows it in outline mode. A splitter control allows the user to adjust the relative sizes of the views.

Figure 4.4, on the next page, shows the three user-interface models in the order presented above.

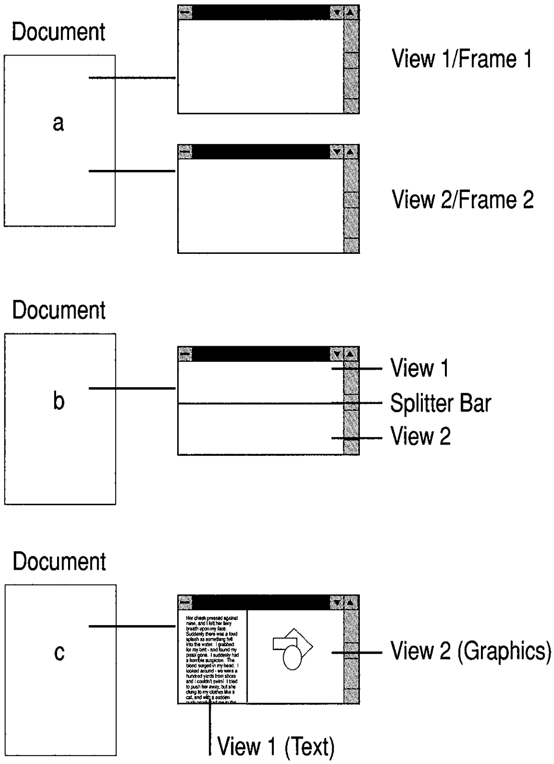


Figure 4.4 Multiple-View User Interfaces

The framework provides these models by implementing the New Window command and by providing class **CSplitterWnd**, as discussed in the next section. You can implement other models using these as your starting point. For sample programs that illustrate different configurations of views, frame windows, and splitters, see MFCSAMP.HLP in MFC.HLP.

For more information about **UpdateAllViews**, see class **CView** in this manual and Chapter 8 in the *Class Library User's Guide*.

Splitter Windows

In a splitter window, the window is, or can be, split into two or more scrollable panes. A splitter control (or "split box") in the window frame next to the scroll bars allows the user to adjust the relative sizes of the panes. Each pane is a view on the same document. In "dynamic" splitters, the views are of the same class, as shown in Figure 4.4(b). In "static" splitters, the views can be of different classes. Splitter windows of both kinds are supported by class **CSplitterWnd**.

Dynamic splitter windows, with views of the same class, allow the user to split a window into multiple panes at will and then scroll different panes to see different parts of the document. The user can also unsplit the window to remove the additional views. The splitter windows added to the Scribble application in Chapter 8 of the *Class Library User's Guide* are an example. That chapter describes the technique for creating dynamic splitter windows. A dynamic splitter window is shown in Figure 4.4(b).

Static splitter windows, with views of different classes, start with the window split into multiple panes, each with a different purpose. For example, in App Studio's bitmap editor, the image window shows two panes side by side. The left-hand pane displays a life-sized image of the bitmap. The right-hand pane displays a zoomed or magnified image of the same bitmap. The panes are separated by a "splitter bar" that the user can drag to change the relative sizes of the panes. A static splitter window is shown in Figure 4.4(c).

For more information, see class **CSplitterWnd** in the alphabetical reference and MFCSAMP.HLP in MFC.HLP.

Initializing and Cleaning Up Documents and Views

Use the following guidelines for initializing and cleaning up after your documents and views:

- The framework initializes documents and views; you initialize any data that you add to them.
- The framework cleans up as documents and views close; you must deallocate any memory that you allocated on the heap from within the member functions of those documents and views.

Note Recall that initialization for the whole application is best done in your override of the **InitInstance** member function of class **CWinApp**, and cleanup for the whole application is best done in your override of the **CWinApp** member function **ExitInstance**.

The life cycle of a document (and its frame window and view or views) in an MDI application is as follows:

1. During dynamic creation, the document constructor is called.
2. For each new document, the document's **OnNewDocument** or **OnOpenDocument** is called.
3. The user interacts with the document throughout its lifetime.
4. The framework calls **DeleteContents** to delete data specific to a document.
5. The document's destructor is called.

In an SDI application, step 1 is performed once, when the document is first created. Then steps 2 through 4 are performed repeatedly each time a new document is opened. The new document reuses the existing document object. Finally, step 5 is performed when the application ends.

Initializing

Documents are created in two different ways, so your document class must support both ways. First, the user can create a new, empty document with the File New command. In that case, initialize the document in your override of the **OnNewDocument** member function of class **CDocument**. Second, the user can use the File Open command to create a new document whose contents are read from a file. In that case, initialize the document in your override of the **OnOpenDocument** member function of class **CDocument**. If both initializations are the same, you can call a common member function from both overrides, or **OnOpenDocument** can call **OnNewDocument** to initialize a clean document and then finish the open operation.

Views are created after their documents are created. The best time to initialize a view is after the framework has finished creating the document, frame window, and view. You can initialize your view by overriding the **OnInitialUpdate** member function of **CView**. If you need to reinitialize or adjust anything each time the document changes, you can override **OnUpdate**.

Cleaning Up

When a document is closing, the framework first calls its **DeleteContents** member function. If you allocated any memory on the heap during the course of the document's operation, **DeleteContents** is the best place to deallocate it.

Note You should not deallocate document data in the document's destructor. In the case of an SDI application, the document object may be reused.

You can override a view's destructor to deallocate any memory you allocated on the heap.

Special View Classes

Besides **CScrollView**, the Microsoft Foundation Class Library provides two other classes derived from **CView**:

- **CFormView**, a view with attributes of a dialog box and a scrolling view. A **CFormView** is created from a dialog-template resource. You can create the dialog-template resource with App Studio.
- **CEditView**, a view that uses the Windows edit control as a simple multiline text editor. You can use a **CEditView** as the view on a document.

CFormView

CFormView provides a view based on a dialog-template resource. You can use it to create formlike views with edit boxes and other dialog controls. The user can scroll the form view and tab among its controls. Form views support scrolling using the **CScrollView** functionality. For more information, see class **CFormView** in the alphabetical reference.

CEditView

CEditView provides the functionality of a **CEdit** control with enhanced editing features: printing; find and replace; cut, copy, paste, clear, and undo commands; and File Save and File Open commands. You can use a **CEditView** to implement a simple text-editor view. See classes **CEditView** and **CEdit** in the alphabetical reference.

Printing and Print Preview

Microsoft Windows implements device-independent display. This means that the same drawing calls, made through a device context passed to your view's `OnDraw` member function, are used to draw on the screen and on other devices, such as printers. You use the device context to call graphics device interface (GDI) functions, and the device driver associated with the particular device translates the calls into calls that the device can understand.

When your framework document prints, `OnDraw` receives a different kind of device-context object as its argument; instead of a **CPaintDC** object, it gets a **CDC** object associated with the current printer. `OnDraw` makes exactly the same calls through the device context as it does for rendering your document on the screen.

The framework also provides an implementation of the File Print Preview command as described below.

Chapter 9 in the *Class Library User's Guide* describes the partnership between you and the framework during printing and print preview and provides an example. In particular, see Figure 9.1 in that chapter.

Printing the Document

To print, the framework calls member functions of the view object to set up the Print dialog box, allocate fonts and other resources needed, set the printer mode for a given page, print a given page, and deallocate resources. Once the document as a whole is set up, the process iteratively prints each page. When all pages have been printed, the framework cleans up and deallocates resources. You can, and sometimes must, override some view member functions to facilitate printing. For information, see class **CView**.

When the view's **OnPrint** member function is called, it must calculate what part of the document image to draw for the given page number. Typically, **OnPrint** adjusts the viewport origin or the clipping region of the device context to specify what should be drawn. Then **OnPrint** calls the view's **OnDraw** member function to draw that portion of the image.

Print Preview

The framework also implements print-preview functionality and makes it easy for you to use this functionality in your applications. Print preview shows a reduced image of either one or two pages of the document as it would appear when printed. The implementation also provides controls for printing the displayed page(s), moving to the next or the previous page, toggling the display between one and two pages, zooming the display in and out to view it at different sizes, and closing the display. If the framework knows how long the document is, it can also display a scroll bar for moving from page to page.

To implement print preview, instead of directly drawing an image on a device, the framework must simulate the printer using the screen. To do this, the Microsoft Foundation Class Library implements the **CPreviewDC** class, which is used in conjunction with the implementation class **CPreviewView**. All **CDC** objects contain two device contexts. In a **CPreviewDC** object, the first device context represents the printer being simulated; the second represents the screen on which output is actually displayed.

In response to a Print Preview command from the File menu, the framework creates a **CPreviewDC** object. Then when your application performs an operation that sets a characteristic of the printer device context, the framework performs a similar operation on the screen device context. For example, if your application selects a font for printing, the framework selects a font for screen display that simulates the printer font. When your application sends output that would go to the printer, the framework instead sends it to the screen.

The order and manner in which pages of a document are displayed are also different for print preview. Instead of printing a range of pages from start to finish, print preview displays one or two pages at a time and waits for a cue from the user before it displays different pages.

You are not required to do anything to provide print preview, other than to make sure the Print Preview command is in the File menu for your application. However, if you choose, you can modify the behavior of print preview in a number of ways. For more information about making such modifications to print preview in your application, see Technical Note 30 in MSVC\HELP\MFCNOTES.HLP.

In the Next Chapter

In this and previous chapters, you have seen how the framework's application, frame window, document, and view classes work, bound together by messages and commands mapped to handler functions in the program's run-time objects. In Chapter 5, you will learn about dialog boxes and the controls that appear in them and about control bars, such as toolbars, status bars, and dialog bars. You will also learn how to incorporate context-sensitive Windows help in your application.

CHAPTER 5

Working with Dialog Boxes, Controls, Control Bars, and Context-Sensitive Help

The previous chapter explained windows, particularly the frame windows used to display views of documents. As you saw briefly in that chapter, class **CWnd** is the base class of many other window classes besides the frame windows.

This chapter covers the following topics, including several additional categories of window classes:

- Dialog boxes
- Control windows
- Control bars
- Context-sensitive Windows Help

Dialog boxes are used to take user input. Inside a dialog box, the user interacts with controls, such as buttons, list boxes, combo boxes, and edit boxes. You can also place controls in a frame window, a view, or a control bar.

A toolbar is a control bar that contains bitmapped buttons; these buttons can be configured to appear and behave as pushbuttons, radio buttons, or check boxes. A status bar is a control bar that contains text-output panes, or "indicators." A dialog bar is a control bar based on a dialog-template resource; as in a dialog box, the user can tab among the controls.

This chapter also explains how to implement context-sensitive Windows Help in your application. The Microsoft Foundation Class Library simplifies the process. If you choose the Context-Sensitive Help option in AppWizard, AppWizard creates basic .RTF files and supplies other code needed to invoke Help.

Dialog Boxes

Applications for the Windows graphical user interface frequently communicate with the user through dialog boxes. Class **CDialog** provides an interface for managing dialog boxes, App Studio makes it easy to design dialog boxes and create their dialog-template resources, and ClassWizard simplifies the process of initializing and validating the controls in a dialog box and of gathering the values entered by the user.

This section explains:

- Modal and modeless dialog boxes.
- The roles of AppWizard, App Studio, and ClassWizard in creating dialog resources and dialog classes for dialog boxes.
- Controls in dialog boxes.
- How dialog boxes are invoked and displayed on the screen.
- Initializing and gathering data from the controls in a dialog box: dialog data exchange (DDX).
- Validating data entered in a dialog box: dialog data validation (DDV).
- Dialog classes supplied by the class library.

Dialog-Box Components in the Framework

In the framework, a dialog box has two components:

- A dialog-template resource that specifies the dialog box's controls and their placement.

  The dialog resource stores a dialog template from which Windows creates the dialog window and displays it. The template specifies the dialog box's characteristics, including its size, location, style, and the types and positions of the dialog box's controls. You will usually use a dialog template stored as a resource, but you can also create your own template in memory.
- A dialog class, derived from **CDialog**, to provide a programmatic interface for managing the dialog box.

  A dialog box is a window and will be attached to a Windows window when visible. When the dialog window is created, the dialog-template resource is used as a template for creating child window controls for the dialog box.

Modal and Modeless Dialog Boxes

You can use class **CDialog** to manage two kinds of dialog boxes:

- Modal dialog boxes, which require the user to respond before continuing the program
- Modeless dialog boxes, which stay on the screen and are available for use at any time but permit other user activities

The App Studio and ClassWizard procedures for creating a dialog template are the same for modal and modeless dialog boxes.

Creating a dialog box for your program requires the following steps:

1. Use App Studio to design the dialog box and create its dialog-template resource.
2. Use ClassWizard to create a dialog class.
3. Connect its controls to message handlers in the dialog class.
4. Use ClassWizard to add data members associated with the dialog box's controls and to specify dialog data exchange and dialog data validations for the controls.

Creating the Dialog Resource with App Studio

To design the dialog box and create the dialog resource, you use App Studio. In the App Studio dialog editor, you can:

- Adjust the size and location your dialog will have when it appears.
- Drag various kinds of controls—including VBX and other custom controls—from a controls palette and drop them where you want them in the dialog box.
- Position the controls with alignment buttons on the App Studio toolbar.
- Test your dialog box by simulating the appearance and behavior it will have in your program. In Test mode, you can manipulate the dialog box's controls by typing text in text boxes, clicking pushbuttons, and so on.

When you finish, your dialog-template resource is stored in your application's resource script file. You can edit it later if needed. For a full description of how to create and edit dialog resources in App Studio, see the *App Studio User's Guide*.

When the dialog box's appearance suits you, use ClassWizard to create a dialog class and map its messages, as discussed in the next section.

Creating a Dialog Class with ClassWizard

ClassWizard helps you manage the dialog-related tasks shown in Table 5.1.

Table 5.1 Dialog-Related Tasks

Task	Apply to . . .
Create a new **CDialog**-derived class to manage your dialog box.	Each dialog box.
Map Windows messages to your dialog class.	Each message you want handled.
Declare class member variables to represent the controls in the dialog box.	Each control that yields a text or numeric value you want to access from your program.

Table 5.1 Dialog-Related Tasks *(continued)*

Task	Apply to . . .
Specify how data is to be exchanged between the controls and the member variables.	Each control that you want to access from your program.
Specify validation rules for the member variables.	Each control that yields a text or numeric value, if desired.

Mapping Windows messages to your dialog class is explained in "Handling Windows Messages" on page 100. Mapping dialog class member variables to dialog-box controls and specifying data exchange and validation are explained in "Dialog Data Exchange and Validation" on page 101.

Creating Your Dialog Class

For each dialog box in your program, create a new dialog class to work with the dialog resource.

Chapter 9 in the *App Studio User's Guide* explains how to create a new dialog class. When you create a dialog class with ClassWizard, ClassWizard writes the following items in the .H and .CPP files you specify:

In the .H file:

- A class declaration for the dialog class. The class is derived from **CDialog**.

In the .CPP file:

- A message map for the class.
- A standard constructor for the dialog box.
- An override of the **DoDataExchange** member function. Edit this function with ClassWizard. It is used for dialog data exchange and validation capabilities as described later in this chapter.

Life Cycle of a Dialog Box

During the life cycle of a dialog box, the user invokes the dialog box, typically inside a command handler that creates and intializes the dialog object; the user interacts with the dialog box; and the dialog box closes.

For modal dialog boxes, your handler gathers any data the user entered once the dialog box closes. Since the dialog object exists after its dialog window has closed, you can simply use the member variables of your dialog class to extract the data.

For modeless dialog boxes, you may often extract data from the dialog object while the dialog box is still visible. At some point, the dialog object is destroyed; when this happens depends on your code.

Creating and Displaying Dialog Boxes

Creating a dialog object is a two-phase operation. First, construct the dialog object. Then create the dialog window. Modal and modeless dialog boxes differ somewhat in the process used to create and display them. Table 5.2 lists how modal and modeless dialog boxes are normally constructed and displayed.

Table 5.2 Dialog Creation

Dialog Type	How to Create It
Modeless	Construct **CDialog**, then call **Create** member function.
Modal	Construct **CDialog**, then call **DoModal** member function.

Creating Modal Dialog Boxes

To create a modal dialog box, you call either of the two public constructors declared in **CDialog** and then call the dialog object's **DoModal** member function to display the dialog box and manage interaction with it until the user chooses OK or Cancel. This management by **DoModal** is what makes the dialog box "modal." For modal dialog boxes, **DoModal** loads the dialog resource.

Creating Modeless Dialog Boxes

For a modeless dialog box, you must provide your own public constructor in your dialog class. To create a modeless dialog box, call your public constructor and then call the dialog object's **Create** member function to load the dialog resource. You can call **Create** either during or after the constructor call. If the dialog resource has the property **WS_VISIBLE**, the dialog box appears immediately. If not, you must call its **ShowWindow** member function.

Using a Dialog Template in Memory

Instead of using the methods given in Table 5.2, you can create either kind of dialog box indirectly from a dialog template in memory. For more information, see class **CDialog** in the alphabetic reference.

Setting the Dialog Box's Background Color

You can set the background color of your dialog boxes by calling the **CWinApp** member function **SetDialogBkColor** in your `InitInstance` override. The color you set is used for all dialog boxes and message boxes.

Initializing the Dialog Box

After the dialog box and all of its controls are created but just before the dialog box (of either type) appears on the screen, the dialog object's **OnInitDialog** member function is called. For a modal dialog box, this occurs during the **DoModal** call. You typically override this function to initialize the dialog box's controls, such as setting the initial text of an edit box. You must call the **OnInitDialog** member function of the base class, **CDialog**, from your `OnInitDialog` override.

Handling Windows Messages

Dialog boxes are Windows, so they can handle Windows messages if you supply the appropriate handler functions.

Exchanging Data Between Dialog Box and Dialog Object

The framework provides an easy way to initialize the values of controls in a dialog box and to retrieve values from the controls. The more laborious manual approach is to call functions such as the **SetDlgItemText** and **GetDlgItemText** member functions of class **CWnd**, which apply to control windows. With these functions, you access each control individually to set or get its value, calling functions such as **SetWindowText** and **GetWindowText**. The framework's approach automates both initialization and retrieval.

Dialog data exchange (DDX) lets you automatically exchange data between the dialog box and member variables in the dialog object. This exchange works both ways. To initialize the controls in the dialog box, you can set the values of data members in the dialog object, and the values will be transferred automatically to the controls before the dialog box is displayed. Then you can at any time update the dialog data members with data entered by the user. At that point, you can use the data by referring to the data member variables.

You can also arrange for the values of dialog controls to be validated automatically with dialog data validation (DDV).

Use ClassWizard to add DDX and DDV capabilities to a dialog class. DDX and DDV are explained in more detail in "Dialog Data Exchange and Validation" on page 101.

Retrieving Data from the Dialog Object

DDX exchanges data between the dialog box and a dialog object. Once the dialog object's data members have been updated from the dialog box's controls, other objects in your program, such as a view, can access the data through those data members.

For a modal dialog box, you can retrieve any data the user entered when **DoModal** returns **IDOK** but before the dialog object is destroyed. For a modeless dialog box, you can retrieve data from the dialog object at any time by calling **UpdateData** with the argument **TRUE** and then accessing dialog class member variables. This subject is discussed in more detail in "Dialog Data Exchange and Validation" on this page.

Closing the Dialog Box

A modal dialog box closes when the user chooses one of its buttons, typically the OK button or the Cancel button. Choosing the OK or Cancel button causes Windows to send the dialog object a **BN_CLICKED** control-notification message with the button's ID, either **IDOK** or **IDCANCEL. CDialog** provides default handler functions for these messages: **OnOK** and **OnCancel**. The default handlers call the **EndDialog** member function to close the dialog window. You can also call **EndDialog** from your own code. For more information, see the **EndDialog** member function of class **CDialog**.

To arrange for closing and deleting a modeless dialog box, override **PostNcDestroy** and invoke the **delete** operator on the **this** pointer. The next section explains what happens next.

Destroying the Dialog Box

Modal dialog boxes are normally created on the stack frame and destroyed when the function that created them ends. The dialog object's destructor is called when the object goes out of scope.

Modeless dialog boxes are normally created and "owned" by a parent view or frame window—the application's main frame window or a document frame window. The default **OnClose** handler calls **DestroyWindow**, which destroys the dialog-box window. The **PostNcDestroy** handler destroys the C++ dialog object. You should also override **OnCancel** and call **DestroyWindow** from within it.

Dialog Data Exchange and Validation

Dialog data exchange (DDX) is an easy way to initialize the controls in your dialog box and to gather data input by the user. Dialog data validation (DDV) is an easy way to validate data entry in a dialog box. To take advantage of DDX and DDV in your dialog boxes, use ClassWizard to create the data members and set their data types and specify validation rules. For additional information about DDX/DDV and for examples, see Chapter 9 in the *App Studio User's Guide* and Chapter 7 in the *Class Library User's Guide*.

Data Exchange

If you use the DDX mechanism, you set the initial values of the dialog object's member variables, typically in your **OnInitDialog** handler or the dialog constructor. The framework's DDX mechanism then transfers the values of the member variables to the controls in the dialog box, where they appear when the dialog box itself appears. The default implementation of **OnInitDialog** in **CDialog** calls the **UpdateData** member function of class **CWnd** to initialize the controls in the dialog box.

The same mechanism transfers values from the controls to the member variables when the user clicks the OK button (or whenever you call the **UpdateData** member function with the argument **TRUE**). The dialog data validation mechanism validates any data items for which you specified validation rules.

Figure 5.1 illustrates dialog data exchange.

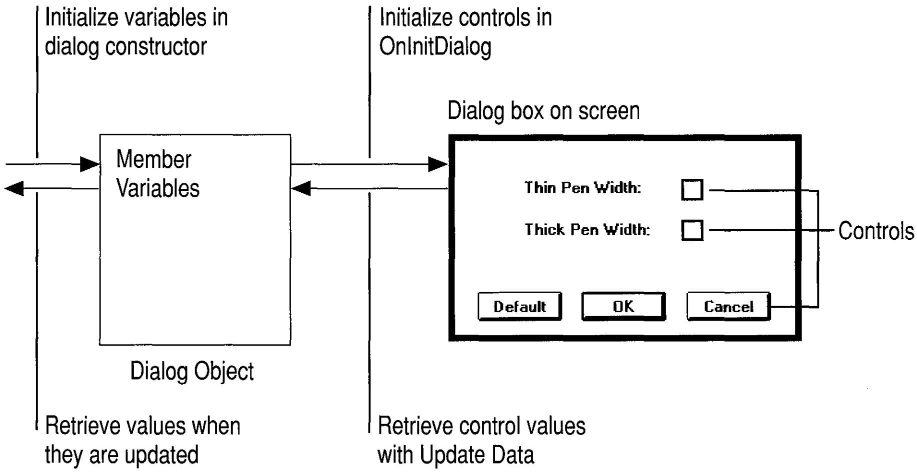


Figure 5.1 Dialog Data Exchange

UpdateData works in both directions, as specified by the **BOOL** parameter passed to it. To carry out the exchange, **UpdateData** sets up a **CDataExchange** object and calls your dialog class's override of **CDialog**'s **DoDataExchange** member function. **DoDataExchange** takes an argument of type **CDataExchange**. The **CDataExchange** object passed to **UpdateData** represents the context of the exchange, defining such information as the direction of the exchange.

When you (or ClassWizard) override **DoDataExchange**, you specify a call to one DDX function per data member (control). Each DDX function knows how to exchange data in both directions based on the context supplied by the **CDataExchange** argument passed to your `DoDataExchange` by **UpdateData**.

The Microsoft Foundation Class Library provides many DDX functions for different kinds of exchange. The following example shows a DoDataExchange override in which two DDX functions and one DDV function are called:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);    // Call base class version
    //{{AFX_DATA_MAP(CMyDialog)
    DDX_Check(pDX, IDC_MY_CHECKBOX, m_bVar);
    DDX_Text(pDX, IDC_MY_TEXTBOX, m_strName);
    DDV_MaxChars(pDX, IDC_MY_TEXTBOX, m_strName, 20);
    //}}AFX_DATA_MAP
}
```

The DDX_ and DDV_ lines between the //{{AFX_DATA_MAP and //}}AFX_DATA_MAP delimiters are a "data map." The sample DDX and DDV functions shown are for a check-box control and an edit-box control, respectively.

If the user cancels a modal dialog box, the **OnCancel** member function terminates the dialog box and **DoModal** returns the value **IDCANCEL**. In that case, no data is exchanged between the dialog box and the dialog object.

Data Validation

You can specify validation in addition to data exchange by calling DDV functions, as shown in the example above. The **DDV_MaxChars** call in the example above validates that the string entered in the text-box control is not longer than 20 characters. The DDV function typically alerts the user with a message box if the validation fails and puts the focus on the offending control so the user can reenter the data. A DDV function for a given control must be called immediately after the DDX function for the same control.

You can also define your own custom DDX and DDV routines. For details on this and other aspects of DDX and DDV, see Technical Note 26 in MSVC\HELP\MFCNOTES.HLP.

ClassWizard will write all of the DDX and DDV calls in the data map for you. Do not manually edit the lines in the data map between the delimiting comments.

Type-Safe Access to Controls in a Dialog Box

The controls in a dialog box can use the interfaces of the Microsoft Foundation Class Library control classes such as **CListBox** and **CEdit**. You can create a control object and attach it to a dialog control. Then you can access the control through its class interface, calling member functions to operate on the control, as shown below. The methods described here are designed to give you type-safe access to a control. This is especially useful for controls such as edit boxes and list boxes.

The connection between a control in a dialog box and a C++ control member variable in a **CDialog**-derived class can be done in two different ways.

Without ClassWizard

The first approach uses an inline member function to cast the return type of class **CWnd**'s **GetDlgItem** member function to the appropriate C++ control type, as in this example:

```
// Declared inline in class CMyDialog
CButton* GetMyCheckbox()
{
    return (CButton*)GetDlgItem(ID_MYCHECKBOX);
}
```

You can then use this member function to access the control in a type-safe manner with code similar to the following:

```
GetMyCheckbox()->SetState(TRUE);
```

With ClassWizard

However, there is a much easier way to accomplish the same effect if you are familiar with the DDX features, using the Control property in ClassWizard.

If you simply want access to a control's value, DDX provides it. If you want to do more than access a control's value, use ClassWizard to add a member variable of the appropriate class to your dialog class. Attach this member variable to the Control property.

Member variables can have a Control property instead of a Value property. The Value property refers to the type of data returned from the control, such as **CString** or **int**. The Control property enables direct access to the control through a data member whose type is one of the control classes in the Microsoft Foundation Class Library, such as **CButton** or **CEdit**.

You can use this object to call any member functions for the control object. Such calls affect the control in the dialog box. For example, for a check-box control represented by a variable `m_checkboxDefault`, of type **CButton**, you could call:

```
m_checkboxDefault.SetState(TRUE);
```

Here the member variable `m_checkboxDefault` serves the same purpose as the member function `GetMyCheckbox` shown above. If the check box is not an auto check box, you would still need a handler in your dialog class for the **BN_CLICKED** control-notification message when the button is clicked.

For more information about controls, see "Controls" on page 106.

Mapping Windows Messages to Your Class

If you need your dialog box to handle Windows messages, override the appropriate handler functions. To do so, use ClassWizard to map the messages to the dialog class. This writes a message-map entry for each message and adds the message-handler member functions to the class. Use the Visual Workbench editor to write code in the message handlers. Chapter 3 describes message maps and message-handler functions in detail.

Commonly Overridden Member Functions

The most likely member functions to override in your **CDialog**-derived class are listed in Table 5.3.

Table 5.3 Commonly Overridden Member Functions of Class CDialog

Member Function	Message It Responds To	Purpose of the Override
OnInitDialog	**WM_INITDIALOG**	Initialize the dialog box's controls
OnOK	**BN_CLICKED** for button **IDOK**	Respond when the user clicks the OK button
OnCancel	**BN_CLICKED** for button **IDCANCEL**	Respond when the user clicks the Cancel button

OnInitDialog, **OnOK**, and **OnCancel** are virtual functions. To override them, you declare an overriding function in your derived dialog class using ClassWizard; in these cases, ClassWizard will not add any message-map entries because they are not necessary.

OnInitDialog is called just before the dialog box is displayed. You must call the default **OnInitDialog** handler from your override—usually as the first action in the handler. By default, **OnInitDialog** returns **TRUE** to indicate that the focus should be set to the first control in the dialog box.

OnOK is typically overridden for modeless but not modal dialog boxes. If you override this handler for a modal dialog box, call the base class version from your override—to ensure that **EndDialog** is called—or call **EndDialog** yourself.

OnCancel is usually overridden for modeless dialog boxes.

For more information about these member functions, see class **CDialog** and the discussion on "Life Cycle of a Dialog Box" on page 98.

Commonly Added Member Functions

If your dialog box contains pushbuttons other than OK or Cancel, you need to write message-handler member functions in your dialog class to respond to the control-notification messages they generate. For an example, see Chapter 7, "Adding A Dialog Box," in the *Class Library User's Guide*. You can also handle control-notification messages from other controls in your dialog box.

Common Dialog Classes

In addition to class **CDialog**, the Microsoft Foundation Class Library supplies several classes derived from **CDialog** that encapsulate commonly used dialog boxes, as shown in Table 5.4. The dialog boxes encapsulated are called the "common dialog boxes" and are part of the Windows common dialog library. The dialog-template resources and code for these classes is provided in the Windows common dialog boxes that are part of Windows version 3.1.

Table 5.4 Common Dialog Classes

Derived Dialog Class	Purpose
CColorDialog	Lets user select colors
CFileDialog	Lets user select a filename to open or to save
CFindReplaceDialog	Lets user initiate a find or replace operation in a text file
CFontDialog	Lets user specify a font
CPrintDialog	Lets user specify information for a print job

For more information about the common dialog classes, see the individual class names in the alphabetic reference.

Two other classes in the Microsoft Foundation Class Library have dialog-like characteristics. For information about class **CFormView**, see "CFormView" on page 91 in Chapter 4. For information about class **CDialogBar**, see "Control Bars" on page 111.

Controls

The Microsoft Foundation Class Library supplies a set of classes that correspond to the standard control windows provided by Microsoft Windows. These include buttons of several kinds, static- and editable-text controls, scroll bars, list boxes, and combo boxes. Table 5.5 lists the classes and the corresponding standard controls. The next section describes new kinds of controls.

Table 5.5 Standard Control Window Classes

Class	Windows Control
CStatic	Static-text control
CButton	Button control: pushbutton, check box, radio button, or group-box control
CListBox	List-box control
CComboBox	Combo-box control
CEdit	Edit control
CScrollBar	Scroll-bar control

Each control class encapsulates a Windows control and provides a member function user interface to the underlying control. Using a control object's member functions, you can get and set the value or state of the control and respond to various standard messages sent by the control to its parent window (usually a dialog box). For additional control classes, see "New Controls," which follows.

You can create control objects in a window or dialog box. You can also use a control class as an interface to a control created in a dialog box from a dialog-template resource.

New Controls

In addition to the standard Windows controls discussed above, the Microsoft Foundation Class Library provides several new control classes. These provide buttons labeled with bitmaps instead of text, control bars, VBX controls, controls that support Microsoft Windows for Pen Computing operations, and splitter-window controls. Splitter windows were discussed in Chapter 4.

Table 5.6 shows the new classes and their purposes.

Table 5.6 New Control Classes

Class	Purpose
CBitmapButton	Button labeled with a bitmap instead of text
CToolBar	Toolbar arranged along a border of a frame window and containing other controls
CStatusBar	Status bar arranged along a border of a frame window and containing panes, or indicators
CDialogBar	Control bar created from a dialog-template resource and arranged along a border of a frame window
CVBControl	Custom control compatible with Visual C++ and Visual Basic

Table 5.6 New Control Classes *(continued)*

Class	Purpose
CHEdit	Text box in which the user can enter and edit text using standard pen editing gestures
CBEdit	Like a **CHEdit**, but with boxes to guide text entry

Control bars, including toolbars, status bars, and dialog bars, are discussed in "Control Bars" on page 111.

Bitmap Buttons

Class **CBitmapButton** allows you to have button controls labeled with bitmaps instead of text. An object of this class stores four **CBitmap** objects that represent various states of the button: up (active), down (pushed), focused, and disabled. Bitmap buttons can be used in dialog boxes. For more information, see class **CBitmapButton**. Figure 5.2 shows bitmap buttons in a dialog box.

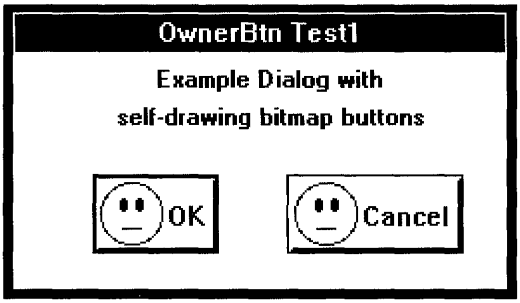


Figure 5.2 Bitmap Buttons

VBX Controls

Class **CVBControl** allows you to use VBX controls. You can use VBX controls in both Visual C++ and Microsoft Visual Basic. You can use the class to load controls, get their properties, set their properties, change their screen location, and perform many other operations. You can also import VBX controls into App Studio and place them in dialog boxes. For more information, see class **CVBControl**. For information about using VBX controls in App Studio, see the *App Studio User's Guide*.

Windows for Pen Controls

Classes **CHEdit** and **CBEdit** support programming Windows for Pen applications. These classes allow you to place controls in your dialog boxes that can be edited with a pen. For more information, see classes **CHEdit** and **CBEdit**.

Controls and Dialog Boxes

Normally the controls in a dialog box are created from the dialog template at the time the dialog box is created. Use ClassWizard to manage the controls in your dialog box. For details, see "Dialog Data Exchange and Validation" on page 101, "Type-Safe Access to Controls in a Dialog Box" on page 103, and "Mapping Windows Messages to Your Class" on page 105.

Making and Using Controls

You make most controls for dialog boxes in the App Studio dialog editor. But you can also create controls in any dialog box or window.

Using App Studio

When you create your dialog-template resource with App Studio, you drag controls from a controls palette and drop them into the dialog box. This adds the specifications for that control type to the dialog-template resource. When you construct a dialog object and call its **Create** or **DoModal** member function, the framework creates a Windows control and places it in the dialog window on screen.

Doing It By Hand

To create a control object yourself, you will usually embed the C++ control object in a C++ dialog or frame window object. Like many other objects in the framework, controls require two-stage construction. You should call the control's **Create** member function as part of the parent dialog box or frame window creation. For dialog boxes, this is usually done in **OnInitDialog**, and for frame windows, in **OnCreate**.

The following example shows how you might declare a **CEdit** object in the class declaration of a derived dialog class and then call the **Create** member function in **OnInitDialog**. Because the **CEdit** object is declared as an embedded object, it is automatically constructed when the dialog object is constructed, but it must still be initialized with its own **Create** member function.

```
class CMyDialog : public CDialog
{
protected:
    CEdit m_edit;   // Embedded edit object
public:
    virtual BOOL OnInitDialog();
};
```

The following OnInitDialog function sets up a rectangle, then calls **Create** to create the Windows edit control and attach it to the uninitialized **CEdit** object.

```
BOOL CMyDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    CRect rect(85, 110, 180, 210);

    m_edit.Create(WS_CHILD | WS_VISIBLE | WS_TABSTOP |
             ES_AUTOSCROLL | WS_BORDER, rect, this, ID_EXTRA_EDIT);
    m_edit.SetFocus();
    return FALSE;
}
```

After creating the edit object, you can also set the input focus to the control by calling the **SetFocus** member function. Finally, you return 0 from **OnInitDialog** to show that you set the focus. If you return nonzero, the dialog manager sets the focus to the first control item in the dialog item list.

Deriving Controls from a Standard Control

As with any **CWnd**-derived class, you can modify a control's behavior by deriving a new class from an existing control class.

To create a derived control class, follow these steps:

1. Derive your class from an existing control class and optionally override the **Create** member function so that it provides the necessary arguments to the base-class **Create** function.
2. Use ClassWizard to provide message-handler member functions and message-map entries to modify the control's behavior in response to specific Windows messages.
3. Provide new member functions to extend the functionality of the control (optional).

Using a derived control in a dialog box requires extra work. The types and positions of controls in a dialog box are normally specified in a dialog-template resource. If you create a derived control class, you cannot specify it in a dialog template since the resource compiler knows nothing about your derived class. To place your derived control in a dialog box, follow these steps:

1. Embed an object of the derived control class in the declaration of your derived dialog class.
2. Override the **OnInitDialog** member function in your dialog class to call the **SubclassDlgItem** member function for the derived control.

SubclassDlgItem "dynamically subclasses" a control created from a dialog template. When a control is dynamically subclassed, you hook into Windows, process some messages within your own application, then pass the remaining messages on to Windows. For more information, see the **SubclassDlgItem** member function of class **CWnd**. The following example shows how you might write an override of **OnInitDialog** to call **SubclassDlgItem**:

```
BOOL CMyDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_wndMyBtn.SubclassDlgItem(IDC_MYBTN, this);
    return TRUE;
}
```

Because the derived control is embedded in the dialog class, it will be constructed when the dialog box is constructed, and it will be destroyed when the dialog box is destroyed. Compare this code to the previous example on page 110.

Control Bars

Control bars greatly enhance a program's usability by providing quick, one-step command actions. Control bars include toolbars, status bars, and dialog bars. The base class of all control bars is **CControlBar**.

- A toolbar is a control bar that displays a row of bitmapped buttons that activate commands similarly to menu items. The buttons can act like pushbuttons, check boxes, or radio buttons. Toolbars are usually aligned to the top of a frame window.
- A status bar is a control bar with a row of text output panes, or "indicators." The output panes are commonly used as message lines and as status indicators. Examples include the command help-message lines that briefly explain the selected menu or toolbar command and the indicators that indicate the status of the SCROLL LOCK, NUM LOCK, and other keys. Status bars are usually aligned to the bottom of a frame window.
- A dialog bar is a control bar with the functionality of a modeless dialog box. Dialog bars are created from dialog templates and can contain any Windows control, including VBX controls. Dialog bars support tabbing among controls and can be aligned to the top, bottom, left, or right sides of a frame window.

This section explains how control bars of all three types work. The base class, **CControlBar**, provides the functionality for positioning the control bar in its parent frame window. Because a control bar is usually a child window of a parent frame window, it is a "sibling" to the client view or MDI client of the frame window. A control-bar object uses information about its parent window's client rectangle to position itself. Then it alters the parent's remaining client-window rectangle so that the client view or MDI client window will fill the rest of the client window.

Toolbars

The buttons in a toolbar are analogous to the items in a menu. Both kinds of user-interface objects generate commands, which your program handles by providing handler functions. Often toolbar buttons duplicate the functionality of menu commands, providing an alternative user interface to the same functionality. Such duplication is arranged by giving the button and the menu item the same ID.

Once constructed, a **CToolBar** object creates the toolbar image by loading a single bitmap that contains one image for each button. AppWizard creates a standard toolbar bitmap, in file TOOLBAR.BMP, that you can customize with App Studio. Figure 5.3 shows that bitmap as it appears in the App Studio bitmap editor.

Figure 5.3 The Standard Toolbar Bitmap

Figure 5.4 shows a toolbar as it appears in a running application, including separators between groups of buttons.

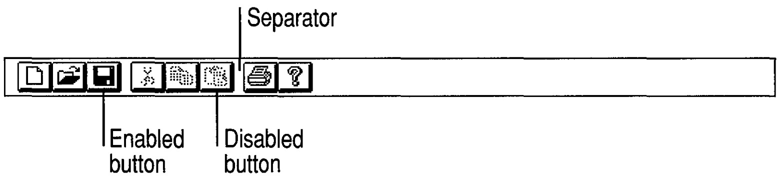


Figure 5.4 A Toolbar with Separators

The buttons in a toolbar are only bitmaps, but the toolbar object processes mouse clicks in the toolbar and generates the appropriate command based on the clicked button's position in the toolbar.

Buttons are correlated with the commands they generate by an array of command IDs, in which the position of an ID in the array is the same as the position of a button image in the toolbar bitmap. If you choose the Initial Toolbar option in AppWizard, AppWizard adds a "buttons" array to the source file for your main frame window class. The array also contains **ID_SEPARATOR** elements used to space the buttons into groups. The separators are ignored in determining button positions. For an example of using App Studio and the array to modify the default toolbar provided by AppWizard, see Chapter 5 in the *Class Library User's Guide*.

You can make the buttons in a toolbar appear and behave as pushbuttons, check boxes, or radio buttons.

For more information, see class **CToolBar** in the alphabetic reference.

Status Bars

As with toolbars, a **CStatusBar** object is based on an array of IDs for its indicator panes. If you select the Initial Toolbar option in AppWizard, AppWizard creates the array for a status bar as well as the array for a toolbar in the source file for your main frame window class. The array looks like this:

```
static UINT BASED_CODE indicators[] =
{
    ID_SEPARATOR,        // message line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
```

These indicators are arranged horizontally along the status bar from left to right. You can add more indicators by adding more IDs to the array. You can size these indicators as needed. You can also add separators by adding **ID_SEPARATOR** elements. The leftmost indicator, at position 0, takes up all space remaining after the other panes are placed. This indicator is most often used as a message area in which to display text strings such as command prompts. Figure 5.5 shows a status bar that displays several indicators.

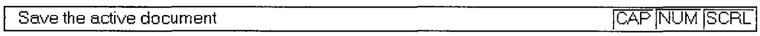


Figure 5.5 A Status Bar

Like the toolbar, the status-bar object is embedded in its parent frame window and is constructed automatically when the frame window is constructed. During creation, a call to the **SetIndicators** member function of class **CStatusBar** associates an ID from the array with each indicator. The status bar, like all control bars, is destroyed automatically as well.

For an example of using a status bar, see the Scribble tutorial program in the *Class Library User's Guide*. For more information, see class **CStatusBar**.

Dialog Bars

Because it has the characteristics of a modeless dialog box, a **CDialogBar** provides a more powerful toolbar. There are several key differences between a toolbar and a **CDialogBar**. A **CDialogBar** is created from a dialog-template resource, which you can create with App Studio and which can contain any kind of Windows control. The user can tab from control to control. And you can specify an alignment style to align the dialog bar with any part of the parent frame window or even to leave it in place if the parent is resized. Figure 5.6 shows a dialog bar with a variety of controls.

Figure 5.6 A Dialog Bar

In other respects, working with a **CDialogBar** is like working with a modeless dialog box. Use App Studio to design and create the dialog resource.

One of the virtues of dialog bars is that they can include controls other than buttons.

While it is normal to derive your own dialog classes from **CDialog**, you do not typically derive your own class for a dialog bar. Dialog bars are extensions to a main window and any dialog-bar control-notification messages, such as **BN_CLICKED** or **EN_CHANGE**, will be sent to the parent of the dialog bar—the main window.

For more information about dialog bars, see class **CDialogBar**.

Context-Sensitive Help

Applications written for Windows usually provide context-sensitive Help, allowing the user to get Help on a particular window, dialog box, command, or toolbar button. The Microsoft Foundation Class Library makes it simple to add context-sensitive Help to your application.

The user can access Help in three ways:

- Getting Help from the Help menu.
- Getting Help on the task at hand by pressing the F1 key. This kind of help is called "F1 Help."
- Getting Help by invoking a "help mode" with SHIFT+F1 and then selecting a user-interface object to get help about. This kind of help is called "SHIFT+F1 Help."

This section explains how the framework manages the three kinds of Help support. It also explains the tools you use to add Help support. For a detailed example, see Chapter 10 in the *Class Library User's Guide*. For additional technical information, see Technical Note 28 in MFCNOTES.HLP.

Components of Help

The Help subsystem in the framework has the following components, many of which are supplied by AppWizard when you choose its Context-Sensitive Help option:

- A Help drop-down menu with several commands. For a new MDI application, there are two copies of this drop-down menu: one for an application with no open documents and one for each type of document that uses its own menu structure. AppWizard supplies these menus.
- Several message-map entries in your **CWinApp**-derived application class. AppWizard supplies these entries.
- Message handlers corresponding to the message-map entries. Class **CWinApp** supplies these handlers and AppWizard supplies the message-map entries for them.
- The **CWinApp::WinHelp** member function, which calls WINHELP.EXE, the Windows Help program.
- Additional AppWizard support for Help, including several Help-related files. The files include skeleton .RTF files that contain Help entries for the common elements of the Windows user interface such as the File and Edit menus. You can edit these files to revise the supplied text and add your own application-specific Help information.
- A mechanism and tool for mapping resource and command IDs in your application to "help contexts" in Windows Help. The MAKEHM tool is described later.

Help-Menu Support

The framework implements two Help menu commands:

- Help Index launches Windows Help with the Help index. The user can browse Help topics or search for a specific topic. The command ID for Help Index is **ID_HELP_INDEX**.
- Using Help launches Windows Help with general information about using Windows Help. The command ID is **ID_HELP_USING**.

Each of these menu items is implemented with commands. The following partial message map for a main frame window class contains mappings for the Help commands:

```
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
   //{{AFX_MSG_MAP(CMyApp)
   // ...
   //}}AFX_MSG_MAP
   // Standard file based document commands
   // ...
   // Global help commands
   ON_COMMAND(ID_HELP_INDEX, CWinApp::OnHelpIndex)
   ON_COMMAND(ID_HELP_USING, CWinApp::OnHelpUsing)
   ON_COMMAND(ID_HELP, CWinApp::OnHelp)
   ON_COMMAND(ID_CONTEXT_HELP, CWinApp::OnContextHelp)
   ON_COMMAND(ID_DEFAULT_HELP, CWinApp::OnHelpIndex)
END_MESSAGE_MAP()
```

The first two entries under the `// Global help commands` comment specify handlers for the two menu commands. The remaining three entries are for F1 Help, Shift+F1 Help, and default Help, respectively. All you have to do to enable these menu items is choose the Context-Sensitive Help option in AppWizard. AppWizard writes the message-map entries.

When the user chooses a Help menu command (or uses one of the context-sensitive Help techniques described in the next two sections), the framework calls **CWinApp**'s **WinHelp** member function, which in turn starts the program WINHELP.EXE, passing context information to it.

F1 Help Support

The framework implements F1 Help for windows, dialog boxes, message boxes, menus, and toolbar buttons. If the cursor is over a window, dialog box, or message box when the user presses the F1 key, the framework opens Windows Help for that window. If a menu item is highlighted, the framework opens Windows Help for that menu item. And if a toolbar button has been pressed (but the mouse not released yet), the framework opens Windows Help for that toolbar button.

When the user presses the F1 key, the framework processes the keystroke as a Help request, as follows, using a variation on the normal command routing. Pressing F1 causes a **WM_COMMAND** message to be sent for the **ID_HELP** command. If the application supports Help, this command is mapped to the **OnHelp** message handler of class **CWinApp** and is routed directly there. **OnHelp** uses the ID of the current frame window or dialog box to determine the appropriate Help topic to display to the user. If no specific Help topic is found, **OnHelp** displays default Help, which is usually mapped to **CWinApp** member function **OnHelpIndex** in the application object's message map—the same handler as for the Help Index menu command.

SHIFT+F1 Help Support

If the user presses SHIFT+F1 at any time the application is active, the framework puts the application into Help mode and changes the cursor to a Help cursor. The next thing the user clicks determines what Help context the framework opens in Windows Help.

If the user presses SHIFT+F1, the framework routes the command **ID_CONTEXT_HELP** through the normal command routing. The command is mapped to the **CWinApp** member function **OnContextHelp**, which captures the mouse, changes the cursor to a Help cursor (arrow + question mark), and puts the application into Help mode. The Help cursor is maintained as long as the application is in Help mode but reverts to a normal arrow cursor if it is not over the application that is in Help mode. Activating a different application cancels Help mode in the original application. While in Help mode, the application determines what object the user clicks on and calls the **CWinApp** member function **WinHelp** with the appropriate context, determined from the object clicked upon. Once an object has been selected, Help mode ends and the cursor is restored to the normal arrow.

For more information, see Technical Note 28 in MFCNOTES.HLP.

More Precise Context-Sensitivity

The standard Help implementation in the framework can obtain a Help context from a window, dialog box, message box, menu item, or toolbar button. If you need more precise control over this mechanism, you can override parts of the mechanism.

For additional information, see Technical Note 28 in MFCNOTES.HLP.

Help Support Tools

You will use three main tools to develop your application's Help system: AppWizard, MAKEHM, and the Windows Help Compiler (the Help Compiler is included with the Microsoft Visual C++ Professional Edition). You also need an editor, such as Microsoft Word for Windows, that can edit .RTF files. You can use App Studio to create bitmaps to include in your Help files.

AppWizard

As you have seen, AppWizard is your first tool for implementing context-sensitive Help. Set the Context-Sensitive Help option in AppWizard's Options dialog box. AppWizard then provides the message-map entries in your **CWinApp**-derived class that connect up the whole help mechanism. AppWizard also creates a set of skeletal starter files, as shown in Table 5.7. The bitmap and .RTF files are in an HLP subdirectory that AppWizard creates in your project directory.

Table 5.7 AppWizard-Supplied Help Files

File	Description
[Yourproject].HPJ	A Windows Help project file that the Windows Help Compiler uses to compile your Help.
MAKEHELP.BAT	A batch file that manages Help ID mapping and calls the Help Compiler.
HLP*.BMP	Various bitmap files used with the supplied Help files.
HLP*.RTF	Skeleton Help files in .RTF format that contain starter Help for the application components supplied by the framework.

The help project file (.HPJ) and MAKEHELP.BAT are in your project directory. The other files are in an HLP subdirectory of your project directory.

You can edit these files as described in "Authoring and Compiling Help" on page 120 to fill in application-specific Help information.

MAKEHM and MAKEHELP.BAT

Once you've created the Help support files with AppWizard and are ready to prepare context-sensitive Help files, run the MAKEHELP.BAT tool from the MS-DOS command line to create a "Help mapping" file (.HM extension) and to compile your Help file. MAKEHELP.BAT calls the MAKEHM tool, which translates the contents of your RESOURCE.H file to a Help mapping file, which is then included in the [MAP] section of your .HPJ file. The [MAP] section associates context strings (or aliases) with context numbers used by the Help Compiler. Then MAKEHELP.BAT calls the Windows Help Compiler to compile your Help file.

When you create a new resource or object within a resource, App Studio assigns it an identifier, or symbol, consisting of a C preprocessor macro name mapped to an integer value. App Studio writes these symbols as **#define** statements in a file called RESOURCE.H.

MAKEHM reads your RESOURCE.H file, locates all applicable **#define** statements (defining various IDs, including those for dialog boxes, menus, and toolbar buttons), and adds an appropriate number to each ID number, using certain rules that depend on the kind of ID (dialog box, menu, etc.). The rules are defined by

MAKEHELP.BAT; the MAKEHM tool is actually more flexible than needed for MAKEHELP.BAT. The new "Help IDs" are written to an .HM file, which the Help Compiler uses to define contexts. For example, the following IDs defined in RESOURCE.H

```
#define IDD_MY_DIALOG   2000
#define ID_MY_COMMAND    150
```

would be translated by MAKEHM into

```
HIDD_MY_DIALOG     0x207d0
HID_MY_COMMAND     0x10096
```

Dialog-box IDs are translated to values beginning at 0x20000. Command and resource IDs are translated to values beginning at 0x10000. That is, the framework reserves specific ranges of values for different kinds of objects. For details, see the contents of MAKEHELP.BAT and Technical Note 28 in MFCNOTES.HLP.

This format is compatible with the Help Compiler, which maps context IDs (the numbers on the right side) to topic names (the symbols on the left). Use these topic names in the .RTF Help files to identify contexts.

Preferred Resource ID Prefixes

To facilitate using MAKEHELP.BAT and MAKEHM, observe the conventions in specifying IDs for your resource objects, as shown in Table 5.8. It is important that different kinds of resource objects have different ID prefixes.

Table 5.8 Preferred Resource ID Naming Conventions

Predefined ID	Object
IDP_	Message-box prompt
IDD_	Dialog-box ID
ID_	Toolbar or menu command (**IDM_** is okay too)
IDR_	Frame-related resources
IDW_	Control bar

Use the **IDS_** prefix for normal string resources, and do not write Help topics for them. For string resources used in message boxes, use the **IDP_** prefix and write Help topics for them so the user can get context-sensitive Help by pressing F1 while the message box is displayed.

Authoring and Compiling Help

For details about authoring and compiling Windows Help, see *Programming Tools for the Microsoft Windows Operating System.*

The preferred way to build Help for your framework application is to run MAKEHELP.BAT. You must have the Microsoft Windows 3.1 Help Compiler in your path.

Figure 5.7 shows the general process for creating a Help system for your application.

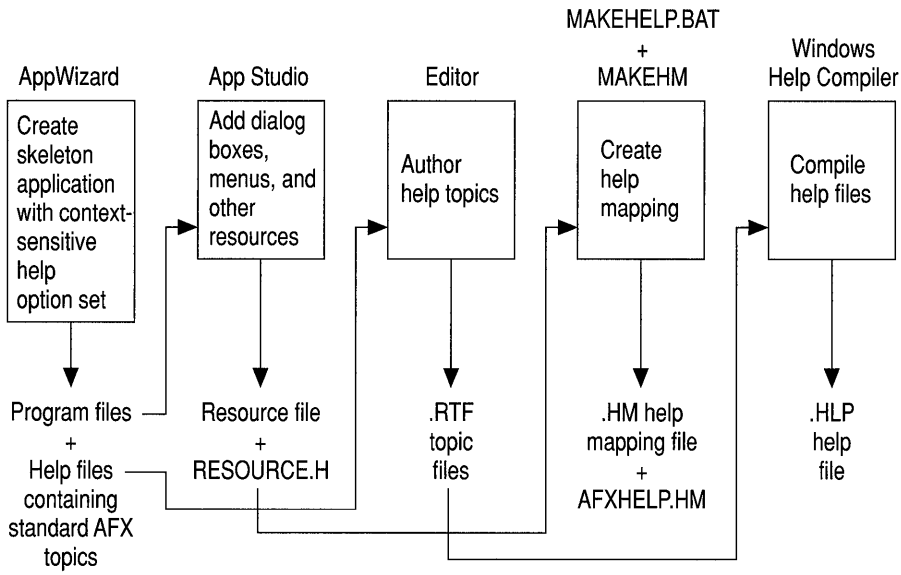


Figure 5.7 Preparing Help Files

For an example of preparing Help files, see Chapter 10 in the *Class Library User's Guide.*

In the Next Chapter

Chapters 2 through 5 have explained how the framework functions and given you some insight into its use. The next chapter explains the "general-purpose" classes and facilities of the Microsoft Foundation Class Library. These classes, global functions, and macros Help you diagnose problems with your application, manage file input/output, handle exceptional conditions, use collection classes, and more.

CHAPTER 6

Using the General-Purpose Classes

This chapter summarizes the use of the general-purpose classes in the Microsoft Foundation Class Library. These classes provide useful services such as diagnostics, exception handling, and collections.

CObject Services

The **CObject** base class provides the following services to objects of its derived classes:

- Object diagnostics
- Run-time class information
- Object persistence

Some of these services are available only if you use certain macros in derived class declarations and implementations. In order to make use of the services listed above, you should seriously consider deriving most of your nontrivial classes from **CObject**. Many of the Microsoft Foundation classes are so derived, including almost all of the application architecture classes that make up the framework.

Object Diagnostics

The Microsoft Foundation library provides many diagnostic features, including diagnostic dump context and object validity checking supplied by the **CObject** class. For global diagnostic features, see "Memory Diagnostics" later in this chapter, on page 127.

Diagnostic Dump Context

The **CDumpContext** class works in conjunction with the **Dump** member function of the **CObject** class to provide formatted diagnostic printing of internal object data. **CDumpContext** provides an insertion (<<) operator that accepts not only

CObject pointers; standard types, such as **BYTE** and **WORD**; and **CString** and **CTime** objects.

A predefined **CDumpContext** object, **afxDump**, is available in the Debug version of the Microsoft Foundation classes (**#define_DEBUG** is required in your source code). For more information about **afxDump**, see "Macros and Globals" on page 1046, and Technical Note 12, which can be found in MSVC\HELP\MFCNOTES.HLP.

Object Validity Checking

You override the base class **AssertValid** member function in your derived class to perform a specific test of your object's internal consistency. Call the **ASSERT_VALID** macro, passing it a pointer to any **CObject**, to call that object's `AssertValid` function. The implementation of an `AssertValid` function usually includes calls to the **ASSERT** macro. For more information about **AssertValid**, see Chapter 15, "Diagnostics," in the *Class Library User's Guide*.

Run-Time Class Information

The Microsoft Foundation classes offer the developer some optional features that make it possible to do run-time type checking. If you derive a class from **CObject** and implement one of three macros (**IMPLEMENT_DYNAMIC**, **IMPLEMENT_DYNCREATE**, or **IMPLEMENT_SERIAL**), you can use member functions to:

- Access the class name at run time.
- Safely cast a generic **CObject** pointer to a derived class pointer.

Run-time class information is particularly valuable in the Debug environment because it can be used to detect incorrect casts and to produce object dumps with class names included.

Note In order to access run-time type information, you must use the **DECLARE_DYNAMIC**, **DECLARE_DYNCREATE**, or **DECLARE_SERIAL** macro in your class declaration, and you must use the corresponding **IMPLEMENT_DYNAMIC**, **IMPLEMENT_DYNCREATE**, or **IMPLEMENT_SERIAL** macro in your class implementation.

Run-time class information is, of course, available in the Release environment. During serialization, the run-time class information is used to store the object's type with the object data.

Run-time class testing is not meant to be a substitute for using virtual functions added in a common base class. Use the run-time type information only when virtual functions are not appropriate.

Object Persistence

Class **CObject**, in conjunction with class **CArchive**, supports "object persistence" through a process called serialization. Object persistence allows you to save a complex network of objects in a permanent binary form (usually disk storage) that persists after those objects are deleted from memory. Later you can load the objects from persistent storage and reconstitute them in memory.

To create your own serializable **CObject**-derived class, you must use the **DECLARE_SERIAL** macro in the class declaration, and you must use the corresponding **IMPLEMENT_SERIAL** macro in the class implementation. If you have added new data members in your derived class, you must override the base class **Serialize** member function to store object data to the archive object and load object data from it. Once you have a serializable class, you can serialize objects of that class to and from a file via a **CArchive** object.

A **CArchive** object provides a type-safe buffering mechanism for writing or reading serializable objects to or from a **CFile** object. Usually the **CFile** object represents a disk file; however, it can be also be a memory file (**CMemFile** object), perhaps representing the Clipboard. A given **CArchive** object either stores (writes, serializes) data or loads (reads, deserializes) data, but never both. Thus two successively created **CArchive** objects are required to serialize data to a file and then deserialize it back from the file. The life of a **CArchive** object is limited to one pass—either writing an object to a file or reading an object from a file.

When storing an object to a file, an archive attaches the **CRuntimeClass** name to the object. Then, when another archive loads the object from a file, the archive uses the **CRuntimeClass** name of the object to dynamically reconstruct the object in memory. A given object may be referenced more than once as it is written to the file by the storing archive. The loading archive, however, will reconstruct the object only once. The details about how an archive attaches **CRuntimeClass** information to objects and reconstructs objects, taking into account possible multiple references, are described in Technical Note 2 in MFCNOTES.HLP.

As you serialize data to an archive, the archive accumulates the data until its buffer is full. When the buffer is full, the archive then writes its buffer to the **CFile** object pointed to by the **CArchive** object. Similarly, as you read data from an archive, the archive reads data from the file to its buffer, and then from the buffer to your deserialized object. This buffering reduces the number of times a hard disk is physically read, thus improving your application's performance.

There are two ways to create a **CArchive** object. The most common way, and the easiest way, is to let the framework create one for your document on behalf of the Save, Save As, and Open commands on the File menu. The other way is to explicitly create the **CArchive** object yourself.

To let the framework create the **CArchive** object for your document, simply implement the document's `Serialize` function, which writes and reads to and from the archive. You also have to implement `Serialize` for any **CObject**-derived objects that the document's `Serialize` function in turn serializes directly or indirectly.

There are other occasions besides serializing a document via the framework when you may need a **CArchive** object. For example, you might want to serialize data to and from the Clipboard, represented by a **CMemFile** object. Or, you might want to develop a user interface for saving files that is different from the one offered by the framework. In this case, you can explicitly create a **CArchive** object. You do this the same way the framework does. For more detailed information, see Chapter 14, "Files and Serialization" in the *Class Library User's Guide*.

The File Classes

The **CFile** family of classes provides a C++ programming interface to operating-system files. The **CFile** class itself gives access to low-level binary files, and the **CStdioFile** class gives access to buffered "standard I/O" files. **CStdioFile** files are often processed in "text mode," which means that newline characters are converted to carriage return–linefeed pairs on output.

CMemFile supports "in-memory files." The files behave like disk files except that bytes are stored in RAM. An in-memory file is a useful means of transferring raw bytes or serialized objects between independent processes.

Because **CFile** is the base class for all file classes, it provides a polymorphic programming interface. If a **CStdioFile** file is opened, for example, its object pointer can be used by the virtual **Read** and **Write** member functions defined for the **CFile** class. The **CDumpContext** and **CArchive** classes, described previously, depend on the **CFile** class for input and output.

The Collection Classes

The Microsoft Foundation Class Library contains a number of ready-to-use lists, arrays, and maps that are referred to as "collection classes." A collection is an extremely useful programming idiom for holding and processing groups of class objects or groups of standard types. A collection object appears as a single object. Class member functions can operate on all elements of the collection.

Most collections may be archived or sent to a dump context. The **Dump** and **Serialize** member functions for **CObject** pointer collections call the corresponding functions for each of their elements. Some collections may not be archived—for example, pointer collections.

If you need a list, array, or map that is not included among the standard collections provided with the Microsoft Foundation classes, you can use the Templdef template tool that is included in the \MSVC\MFC\SAMPLES directory. Technical Note 4, found in MSVC\HELP\MFCNOTES.HLP, describes how to the use this tool.

Note The collection classes **CObArray**, **CObList**, **CMapStringToOb**, and **CMapWordToOb** accept **CObject** pointer elements and thus are useful for storing collections of objects of **CObject**-derived classes. If such a collection is archived or sent to a diagnostic dump context, then the element objects are automatically archived or dumped as well. For more about collection classes, see Chapter 13, "Collections," in the *Class Library User's Guide*.

When you program with the application framework, the collection classes will be especially useful for implementing data structures in your document class. For an example, see the document implementation in the tutorial contained in the *Class Library User's Guide*.

Lists

There are "list" classes for **CString** objects, **CObject** pointers, and void pointers. A list is an ordered grouping of elements. New elements can be added at the head or tail of the list, or before or after a specified element. The list can be traversed in forward or reverse sequence, and elements may be retrieved or removed during the traversal.

Arrays

The Microsoft Foundation Class Library contains "array" classes for bytes, words, doublewords, **CString** objects, **CObject** pointers, and void pointers. An array implemented this way is a dynamically sized grouping of elements that is directly accessible through a zero-based integer subscript. The subscript ([]) operator can be used to set or retrieve array elements. If an element above the current array bound is to be set, then the programmer can specify whether the array is to grow automatically. When growing is not required, array collection access is as fast as standard C array access.

Maps

A "map" is a dictionary that maps keys to values. The map classes support **CString** objects, words, **CObject** pointers, and void pointers. Consider the **CMapWordToOb** class as an example. A **WORD** variable is used as a key to find the corresponding **CObject** pointer. Duplicate key values are not allowed. A key-pointer pair can be inserted only if the key is not already contained in the map. Key lookups are fast because they rely on a hashing technique.

Other Support Classes

The Microsoft Foundation **CString**, **CTime**, and **CTimeSpan** classes are not derived from **CObject**. They are discussed below.

The CString Class

The **CString** class supports dynamic character strings. **CString** objects can grow and shrink automatically, and they can be serialized. Member functions and overloaded operators add Basic-like string-processing capability. These features make **CString** objects easier to use than C-style fixed-length character arrays. Conversion functions allow **CString** objects to be used interchangeably with C-style strings. Thus a **CString** object can be passed to a function that expects a pointer to a constant string (**const char***) parameter.

Like other Microsoft Foundation classes, the **CString** class allocates memory on the heap. You must be sure that **CString** destructors are called at appropriate times to free unneeded memory. There is no automatic "garbage collection" as there is in Basic.

The CTime and CTimeSpan Classes

The **CTime** class encapsulates the run-time **time_t** data type. Thus it represents absolute time values in the range 1970 to 2038, approximately. There are member functions that convert a time value to years, months, days, hours, minutes, and seconds. The class has overloaded insertion and extraction operators for archiving and for diagnostic dumping.

The **CTimeSpan** class extends **time_t** by representing relative time values. When one **CTime** object is subtracted from another one, the result is a **CTimeSpan** object. A **CTimeSpan** object can be added to or subtracted from a **CTime** object. A **CTimeSpan** value is limited to the range of ± 68 years, approximately.

Diagnostic Services

The Microsoft Foundation Class Library provides diagnostic services that make it easier to debug your programs. These services include macros and global functions that allow you to trace your program's memory allocations, dump the contents of objects during run time, and print debugging messages during run time. Most of these services require the Debug version of the library and thus should not be used in released applications. For a detailed description of the functions and macros available, see Chapter 15, "Diagnostics," in the *Class Library User's Guide* and the overview of "Macros and Globals" in this book.

Memory Diagnostics

Many applications use the C++ **new** operator to allocate memory on the heap. The Microsoft Foundation classes provide a special Debug version of **new** that inserts extra control bytes in allocated memory blocks. These control bytes, together with the run-time class information that results from **CObject** derivation, allow you to analyze memory-allocation statistics and detect memory-block bounds violations. A memory dump can include the source filename and the line number of the allocated memory and, in the case of objects from **CObject**-derived classes, the name of the class and the output from its **Dump** function.

Diagnostic Output

Many programmers want diagnostic output statements in their programs, particularly during the early stages of development. The **TRACE** statement acts like **printf** except that the **TRACE** code is not generated by the compiler with the Release version of the library. In the Windows environment, debugging output goes to the debugger if it is present.

Important For important information on using **TRACE**, see the "Macros and Globals" section of this book and Technical Note 7 found in MFCNOTES.HLP.

You can use the **afxDump** dump context object for stream-style dumping of standard types as well as Microsoft Foundation class objects. If you use **afxDump**, be sure to bracket references with **#ifdef _DEBUG** and **#endif** statements.

Assertions

In the Debug environment, the **ASSERT** macro evaluates a specified condition. If the condition is false, the macro displays a message in a message box that gives the source filename and the line number and then terminates the program. In the Release environment, the **ASSERT** statement has no effect.

VERIFY, a companion macro, evaluates the condition in both the Debug and Release environments. It prints and terminates only in the Debug environment.

Classes derived from **CObject**, directly or indirectly, can also override the **AssertValid** member function to test the internal validity of objects of the class. For an example, see "Object Validity Checking" on page 122.

Exception Handling

The Microsoft Foundation Class Library includes an exception-handling mechanism, similar to, and upwardly compatible with, the one in the proposed ANSI C++ standard, for handling "abnormal conditions." An abnormal condition is defined as a condition outside the program's control that influences the outcome of a function. Abnormal conditions include low memory, I/O errors, and attempted use of an unsupported feature. They do not include programming errors or normally expected conditions such as an end-of-file condition. In general, you can consider an exception to be a bug that remains in your program after shipping.

Exception handling in the Microsoft Foundation classes relies on "exception objects" and a group of macros. The process starts with the interruption of normal program execution in response to a **THROW** statement (macro invocation). Execution resumes at the appropriate **CATCH** statement leading into code that presumably deals with the abnormal condition. The exception objects, which are instances of classes derived from **CException**, differentiate the various kinds of exceptions and are used for communication.

This exception-handling scheme eliminates the need for extensive error testing after every library function call. If, for example, you enclose your entire program in an exception-handling block, then you don't have to test for low memory after each statement that contains the **new** operator.

If you don't provide **THROW** and **CATCH** exception-processing code in your classes, then exceptions will be caught in the Microsoft Foundation code. This results in termination of the program through the global function **AfxTerminate**, which normally calls the run-time function **abort**. However, if you use the **AfxSetTerminate** function, the effect of **AfxTerminate** is changed. When programming for Windows, it is important to remember that exceptions cannot cross the boundary of a "callback." In other words, if an exception occurs within the scope of a message handler, it must be caught there, before the next message is processed. If you do not catch an exception, the **CWinApp** member function **ProcessWndProcException** is called as a last resort. This function displays an error message and then continues processing.

For exception-processing examples and a more detailed explanation of error categories, see Chapter 16, "Exceptions," in the *Class Library User's Guide*. For a detailed description of the functions and macros available, see the "Macros and Globals" section in Part 2 of this book.

PART 2

The Microsoft Foundation Class Library Reference

class CArchive

The **CArchive** class allows you to save a complex network of objects in a permanent binary form (usually disk storage) that persists after those objects are deleted. Later you can load the objects from persistent storage, reconstituting them in memory. This process of making data persistent is called "serialization."

You can think of an archive object as a kind of binary stream. Like an input/output stream, an archive is associated with a file and permits the buffered writing and reading of data to and from storage. An input/output stream processes sequences of ASCII characters, but an archive processes binary object data in an efficient, nonredundant format.

You must create a **CFile** object before you can create a **CArchive** object. In addition, you must ensure that the archive's load/store status is compatible with the file's open mode. You are limited to one active archive per file.

When you construct a **CArchive** object, you attach it to an object of class **CFile** (or a derived class) that represents an open file. You also specify whether the archive will be used for loading or storing. A **CArchive** object can process not only primitive types but also objects of **CObject**-derived classes designed for serialization. A serializable class must have a **Serialize** member function, and it must use the **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros, as described under class **CObject**.

The overloaded extraction (>>) and insertion (<<) operators are convenient archive programming interfaces that support both primitive types and **CObject**-derived classes.

#include <afx.h>

See Also **CFile, CObject**

Construction/Destruction—Public Members

CArchive	Creates a **CArchive** object.
~CArchive	Destroys a **CArchive** object and flushes unwritten data.
Close	Flushes unwritten data and disconnects from the **CFile**.

Basic Input/Output—Public Members

Flush	Flushes unwritten data from the archive buffer.
operator >>	Loads objects and primitive types from the archive.
operator <<	Stores objects and primitive types to the archive.

Read	Reads raw bytes.
Write	Writes raw bytes.

Status—Public Members

GetFile	Gets the **CFile** object pointer for this archive.
IsLoading	Determines if the archive is loading.
IsStoring	Determines if the archive is storing.

Object Input/Output—Public Members

ReadObject	Calls an object's **Serialize** function for loading.
WriteObject	Calls an object's **Serialize** function for storing.

Member Functions

CArchive::CArchive

CArchive(CFile* *pFile*, **UINT** *nMode*, **int** *nBufSize* = **512,**
void FAR* *lpBuf* = **NULL)**
throw(CMemoryException, CArchiveException, CFileException);

pFile A pointer to the **CFile** object that is the ultimate source or destination of the persistent data.

nMode A flag that specifies whether objects will be loaded from or stored to the archive. The *nMode* parameter must have one of the following values, with the meaning as given:

- **CArchive::load** Loads data from the archive. Requires only **CFile** read permission.
- **CArchive::store** Saves data to the archive. Requires **CFile** write permission.
- **CArchive::bNoFlushOnDelete** Prevents the archive from automatically calling **Flush** when the archive destructor is invoked. If you set this flag, you are responsible for explicitly calling **Close** before the destructor is invoked. If you do not, your data will be corrupted.

nBufSize An integer that specifies the size of the internal file buffer, in bytes. Note that the default buffer size is 512 bytes. If you routinely archive large objects, you will improve performance if you use a larger buffer size that is a multiple of the file buffer size.

lpBuf An optional **FAR** pointer to a user-supplied buffer of size *nBufSize*. If you do not specify this parameter, the archive allocates a buffer from the local heap and frees it when the object is destroyed. The archive does not free a user-supplied buffer.

Remarks

Constructs a **CArchive** object and specifies whether it will be used for loading or storing objects. You cannot change this specification after you have created the archive. You may not use **CFile** operations to alter the state of the file until you have closed the archive. Any such operation will damage the integrity of the archive. You may access the position of the file pointer at any time during serialization by (1) obtaining the archive's file object from the **GetFile** member function and then (2) using the **CFile::GetPosition** function. You should call **CArchive::Flush** before obtaining the position of the file pointer.

See Also

CArchive::Close, **CArchive::Flush**, **CFile::Close**

Example

```
extern char* pFileName;
CFile f;
char buf[512];
if( !f.Open( pFileName, CFile::modeCreate | CFile::modeWrite ) ) {
   #ifdef _DEBUG
      afxDump << "Unable to open file" << "\n";
      exit( 1 );
   #endif
}
CArchive ar( &f, CArchive::store, 512, buf );
```

CArchive::~CArchive

~CArchive( );

Remarks

The **CArchive** destructor closes the archive if it is not closed already. However, you should call the member function **Close** before calling the destructor. After you have used the **CFile** object for archiving, you must close and destroy it as you usually would.

See Also

CArchive::Flush, **CFile::Close**

CArchive::Close

void Close( )
throw(CArchiveException, CFileException);

Remarks

Flushes any data remaining in the buffer, closes the archive, and disconnects the archive from the file. No further operations on the archive are permitted. After you close an archive, you can create another archive for the same file or you can close the file. The member function **Close** ensures that all data is transferred from the archive to the file, and it makes the archive unavailable. To complete the transfer from the file to the storage medium, you must first use **CFile::Close** and then destroy the **CFile** object.

See Also

CArchive::Flush

CArchive::Flush

void Flush( )
throw(CFileException);

Remarks

Forces any data remaining in the archive buffer to be written to the file. Member function **Flush** ensures that all data is transferred from the archive to the file. You must call **CFile::Close** to complete the transfer from the file to the storage medium.

See Also

CArchive::Close, **CFile::Flush**, **CFile::Close**

CArchive::GetFile

CFile* GetFile( ) const;

Remarks

Gets the **CFile** object pointer for this archive. You must flush the archive before using **GetFile**.

Return Value

A constant pointer to the **CFile** object in use.

Example

```
extern CArchive ar;
const CFile* fp = ar.GetFile();
```

CArchive::IsLoading

BOOL IsLoading( ) const;

Remarks

Determines if the archive is loading data. This member function is called by the **Serialize** functions of the archived classes.

Return Value

TRUE if the archive is currently being used for loading; otherwise **FALSE**.

See Also

CArchive::IsStoring

Example

```
int i;
extern CArchive ar;
if( ar.IsLoading() )
  ar >> i;
else
  ar << i;
```

CArchive::IsStoring

BOOL IsStoring( ) const;

Remarks

Determines if the archive is storing data. This member function is called by the **Serialize** functions of the archived classes. If the **IsStoring** status of an archive is **TRUE**, then its **IsLoading** status is **FALSE**, and vice versa.

Return Value

TRUE if the archive is currently being used for storing; otherwise **FALSE**.

See Also

CArchive::IsLoading

Example

```
int i;
extern CArchive ar;
if( ar.IsStoring() )
  ar << i;
else
  ar >> i;
```

CArchive::Read

UINT Read(void FAR* *lpBuf*, **UINT** *nMax*)
throw(CFileException);

lpBuf A **FAR** pointer to a user-supplied buffer that is to receive the data read from the archive.

nMax An unsigned integer specifying the number of bytes to be read from the archive.

Remarks Reads a specified number of bytes from the archive. The archive does not interpret the bytes. You can use the **Read** member function within your **Serialize** function for reading ordinary structures that are contained in your objects.

Return Value An unsigned integer containing the number of bytes actually read. If the return value is less than the number requested, the end of file has been reached. No exception is thrown on the end-of-file condition.

Example

```
extern CArchive ar;
char pb[100];
UINT nr = ar.Read( pb, 100 );
```

CArchive::ReadObject

CObject* ReadObject(const CRuntimeClass* *pClass*)
throw(CFileException, CArchiveException, CMemoryException);

pClass A constant pointer to the **CRuntimeClass** structure that corresponds to the object you expect to read.

Remarks Reads object data from the archive and constructs an object of the appropriate type. If the object contains pointers to other objects, those objects are constructed automatically. This function is normally called by the **CArchive** extraction (>>) operator overloaded for a **CObject** pointer. **ReadObject**, in turn, calls the **Serialize** function of the archived class. If you supply a nonzero *pClass* parameter, which is obtained by the **RUNTIME_CLASS** macro, then the function verifies the run-time class of the archived object. This assumes you have used the **IMPLEMENT_SERIAL** macro in the implementation of the class.

Return Value A **CObject** pointer that must be safely cast to the correct derived class by using **CObject::IsKindOf**.

See Also **CArchive::WriteObject**, **CObject::IsKindOf**

CArchive::Write

void Write(const void FAR* *lpBuf*, **UINT** *nMax* **)**
throw(CFileException);

lpBuf A pointer to a user-supplied buffer that contains the data to be written to the archive.

nMax An integer that specifies the number of bytes to be written to the archive.

Remarks

Writes a specified number of bytes to the archive. The archive does not format the bytes. You can use the **Write** member function within your **Serialize** function to write ordinary structures that are contained in your objects.

See Also

CArchive::Read

Example

```
extern CArchive ar;
char pb[100];
ar.Write( pb, 100 );
```

CArchive::WriteObject

void WriteObject(const CObject* *pOb* **)**
throw(CFileException, CArchiveException);

pOb A constant pointer to the object being stored.

Remarks

Stores the specified **CObject** to the archive. If the object contains pointers to other objects, they are serialized in turn. This function is normally called by the **CArchive** insertion (<<) operator overloaded for **CObject**. **WriteObject**, in turn, calls the **Serialize** function of the archived class. To enable archiving you must use the **IMPLEMENT_SERIAL** macro. **WriteObject** writes the ASCII class name to the archive. This class name is validated later during the load process. A special encoding scheme prevents unnecessary duplication of the class name for multiple objects of the class. This scheme also prevents redundant storage of objects that are targets of more than one pointer. The exact object encoding method (including the presence of the ASCII class name) could change in future versions of the library.

Note Finish creating, deleting, and updating all your objects before you begin to archive them. Your archive will be corrupted if you mix archiving with object modification.

See Also

CArchive::ReadObject

Operators

CArchive::operator <<

friend CArchive& operator <<(CArchive& *ar*, **const CObject*** *pOb* **)**
throw(CArchiveException, CFileException);

CArchive& operator <<(BYTE *by* **)**
throw(CArchiveException, CFileException);

CArchive& operator <<(WORD *w* **)**
throw(CArchiveException, CFileException);

CArchive& operator <<(LONG *l* **)**
throw(CArchiveException, CFileException);

CArchive& operator <<(DWORD *dw* **)**
throw(CArchiveException, CFileException);

CArchive& operator <<(float *f* **)**
throw(CArchiveException, CFileException);

CArchive& operator <<(double *d* **)**
throw(CArchiveException, CFileException);

Remarks

Stores the indicated object or primitive type to the archive. If you used the **IMPLEMENT_SERIAL** macro in your class implementation, then the insertion operator overloaded for **CObject** calls the protected **WriteObject**. This function, in turn, calls the **Serialize** function of the class.

Return Value

A **CArchive** reference that enables multiple insertion operators on a single line.

See Also

CArchive::WriteObject, **CObject::Serialize**

Example

```
long l;
int i;
extern CArchive ar;
if( ar.IsStoring() )
```

CArchive::operator >>

friend CArchive& operator >>(CArchive& *ar*, **CObject *&** *pOb*)
throw(CArchiveException, CFileException, CMemoryException);

friend CArchive& operator >>(CArchive& *ar*, **const CObject *&** *pOb*)
throw(CArchiveException, CFileException, CMemoryException);

CArchive& operator >>(BYTE& *by*)
throw(CArchiveException, CFileException);

CArchive& operator >>(WORD& *w*)
throw(CArchiveException, CFileException);

CArchive& operator >>(LONG& *l*)
throw(CArchiveException, CFileException);

CArchive& operator >>(DWORD& *dw*)
throw(CArchiveException, CFileException);

CArchive& operator >>(float& *f*)
throw(CArchiveException, CFileException);

CArchive& operator >>(double& *d*)
throw(CArchiveException, CFileException);

Remarks

Loads the indicated object or primitive type from the archive. If you used the **IMPLEMENT_SERIAL** macro in your class implementation, then the extraction operators overloaded for **CObject** call the protected **ReadObject** function (with a nonzero run-time class pointer). This function, in turn, calls the **Serialize** function of the class.

Return Value

A **CArchive** reference that enables multiple insertion operators on a single line.

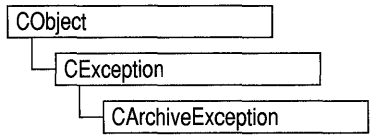
See Also

CArchive::ReadObject, **CObject::Serialize**

Example

```
int i;
extern CArchive ar;
if( ar.IsLoading() )
  ar >> i;
  ar >> l >> i;
```

class CArchiveException : public CException



A **CArchiveException** object represents a serialization exception condition. The **CArchiveException** class includes a public data member that indicates the cause of the exception. **CArchiveException** objects are constructed and thrown inside **CArchive** member functions. You can access these objects within the scope of a **CATCH** expression. The cause code is independent of the operating system. For more information about exception processing, see Chapter 16, "Exceptions," in the *Class Library User's Guide*.

#include <afx.h>

See Also **CArchive**, **AfxThrowArchiveException**

Data Members—Public Members

m_cause	Indicates the exception cause.

Construction/Destruction—Public Members

CArchiveException	Constructs a **CArchiveException** object.

Member Functions

CArchiveException::CArchiveException

CArchiveException(int *cause* **= CArchiveException::none);**

cause An enumerated type variable that indicates the reason for the exception. For a list of the enumerators, see the **m_cause** data member.

Remarks Constructs a **CArchiveException** object, storing the value of *cause* in the object. You can create a **CArchiveException** object on the heap and throw it yourself or let the global function **AfxThrowArchiveException** handle it for you. Do not use this constructor directly; instead, call the global function **AfxThrowArchiveException**.

Data Members

CArchiveException::m_cause

Remarks

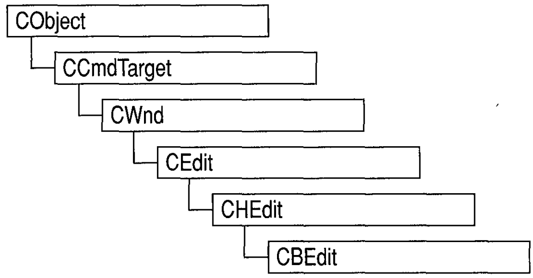
Specifies the cause of the exception. This data member is a public variable of type **int**. Its values are defined by a **CArchiveException** enumerated type. The enumerators and their meanings are as follows:

- **CArchiveException::none** No error occurred.
- **CArchiveException::generic** Unspecified error.
- **CArchiveException::readOnly** Tried to write into an archive opened for loading.
- **CArchiveException::endOfFile** Reached end of file while reading an object.
- **CArchiveException::writeOnly** Tried to read from an archive opened for storing.
- **CArchiveException::badIndex** Invalid file format.
- **CArchiveException::badClass** Tried to read an object into an object of the wrong type.
- **CArchiveException::badSchema** Tried to read an object with a different version of the class.

Note These **CArchiveException** cause enumerators are distinct from the **CFileException** cause enumerators.

---

class CBEdit : public CHEdit



The **CBEdit** class encapsulates the boxed handwriting edit, or "bedit," functionality of Microsoft Windows for Pen Computing. **CBEdit** controls allow the user of your application to enter and modify text using standard pen editing gestures. They differ from handwriting edit, or "hedit," controls, which are created using **CHEdit**-derived classes, in that they display a "comb" that shows the user where each character must be entered. The comb improves recognition accuracy because it gives the recognizer information about the location of input characters.

Text in a boxed edit control is considered a single stream of text that is arranged in rows of cells for convenience. Text always wraps at the end of a row, not necessarily at word boundaries or carriage returns.

You can set the layout of a bedit control by using the **SetBoxLayout** member function. Defaults are used if you do not set the box layout. For information about the default box layout, see *Microsoft Windows for Pen Computing: Programmer's Reference.*

See class **CHEdit** for information about:

- Creating a boxed-edit control using App Studio.
- Setting the alphabet code (ALC) styles for **CBEdit** controls.
- Setting control styles for **CBEdit** controls.
- Notification messages.

If you want to handle Windows notification messages sent by a **CBEdit** control to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler function to the parent class for each message.

#include <afxpen.h>

Construction/Destruction—Public Members

CBEdit	Constructs a **CBEdit** object.
Create	Creates and displays a **CBEdit** control.

Operations

CharOffset	Converts the logical character position of a character in the bedit control to a byte offset to that character.
CharPosition	Converts the byte offset in the text buffer to the logical character position in the bedit control.
DefaultFont	Changes the font of the bedit control to the default font.
GetBoxLayout	Gets the box layout.
SetBoxLayout	Sets the box layout.

Member Functions

CBEdit::CBEdit

CBEdit( );

Remarks Constructs a **CBEdit** object.

See Also **CBEdit::Create**

CBEdit::CharOffset

DWORD CharOffset(UINT *nCharPosition* **);**

nCharPosition The logical position in the bedit control to map to a character position. The first position is 0.

Remarks There is not always a one-to-one correspondence between characters and cells in the bedit control. To find the offset in the text buffer of a given cell position (or "logical" character position), use **CharOffset**.

Return Value If the logical position specified by *nCharPosition* is less than the total number of logical characters in the control, the low word of the return value is the byte offset and the high word is 0. If *nCharPosition* is greater than or equal to the total number of logical characters in the control, the low word contains the length of text in bytes and the high word contains 0xFFFF.

You can use the **LOWORD** and **HIWORD** macros to examine the two parts of the return value.

See Also **CBEdit::CharPosition**, **LOWORD**, **HIWORD**, **WM_HEDITCTL**

CBEdit::CharPosition

DWORD CharPosition(UINT *nCharOffset* **);**

nCharOffset A byte offset into the text buffer. The first offset is 0.

Remarks There is not always a one-to-one correspondence between characters and cells in the bedit control. To find the cell or "logical" character position that corresponds to a given byte offset into the text buffer, use **CharPosition**.

Return Value If the position specified by *nCharOffset* is less than the length of the text in bytes, the low word contains the logical character position and the high word is 0. If the position specified by *nCharOffset* is greater than or equal to the length of the text in bytes, the total number of logical characters in the control is returned in the low word and the high word contains 0xFFFF.

You can use the **LOWORD** and **HIWORD** macros to examine the two parts of the return value.

See Also **CBEdit::CharOffset**, **LOWORD**, **HIWORD**, **WM_HEDITCTL**

CBEdit::Create

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

dwStyle Specifies the bedit control's style. See **CEdit::Create** for a list of these styles.

rect Specifies the bedit control's boxed rectangle. Note that the area sensitive to pen gestures and inking can be modified using member function **SetInflate** of class **CHEdit**.

pParentWnd Specifies the bedit control's parent window (usually a **CDialog**). It must not be **NULL**.

nID Specifies the edit control ID.

Remarks

You construct a **CBEdit** object in two steps. First, construct the **CBEdit** object, then call **Create**, which creates the bedit control and attaches it to the **CBEdit** object. To extend the default message handling, derive a class from **CBEdit**, add a message map to the new class, and override the appropriate message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for the new class.

Return Value

Nonzero if initialization is successful; otherwise 0.

See Also

CEdit::Create, **CBEdit::CBEdit**, **CHEdit::SetInflate**, **WM_HEDITCTL**

CBEdit::DefaultFont

void DefaultFont(BOOL *bRepaint* **);**

bRepaint If **TRUE**, the control is repainted; otherwise, repainting is deferred until forced by some other event.

Comments

If you have made a **SetFont** call, you may want to force the bedit control to display using the font with which it was originally created. **DefaultFont** causes the bedit control to select this default font, and optionally forces repaint of the control.

See Also

CWnd::SetFont, **WM_HEDITCTL**

CBEdit::GetBoxLayout

void GetBoxLayout(LPBOXLAYOUT *lpBoxLayout* **);**

lpBoxLayout A far pointer to a **BOXLAYOUT** structure. See the structure description below.

Remarks

Use **GetBoxLayout** to retrieve a **BOXLAYOUT** structure that describes the way the bedit's boxes are arranged in the control. You can use **GetBoxLayout** in conjunction with **SetBoxLayout** to modify certain aspects of the box layout.

BOXLAYOUT Structure

A **BOXLAYOUT** structure has this form:

```
typedef struct
{
    int cyCusp;
    int cyEndCusp;
    UINT style;
    DWORD rgbText;
    DWORD rgbBox;
    DWORD rgbSelect;
} BOXLAYOUT;
```

A **BOXLAYOUT** structure specifies some of the characteristics of a bedit control.

Members

cyCusp Height (in pixels) of the box when the **BXS_RECT** style is specified, otherwise the height of the comb. This is the equivalent in pixels of **BXD_CUSPHEIGHT** in dialog units.

cyEndCusp Height (in pixels) of the cusps at the ends of the box. This is the equivalent in pixels of **BXD_ENDCUSPHEIGHT** in dialog units.

style 0 for a single-line boxed edit control, **BXS_ENDTEXTMARK** for a multiline boxed edit control, or **BXS_RECT** for a boxed-edit control that uses rectangular boxes instead of a comb.

rgbText If –1, the color of the window text is used; otherwise, this member specifies the RGB color to use for text.

rgbBox If –1, the color of the window frame is used; otherwise, this member specifies the RGB color to use for the boxes.

rgbSelect If –1, the color of the window text is used; otherwise, this member specifies the RGB color to use for the selection.

Comments

Use the **BOXLAYOUT** structure in conjunction with the **GetBoxLayout** and **SetBoxLayout** functions to customize your bedit controls.

See Also

CBEdit::SetBoxLayout, **WM_HEDITCTL**

CBEdit::SetBoxLayout

BOOL SetBoxLayout(LPBOXLAYOUT *lpBoxLayout* **);**

lpBoxLayout A far pointer to a **BOXLAYOUT** structure. See **GetBoxLayout** for a description of this structure.

Remarks

Use **SetBoxLayout** to change the box layout of a bedit control from the default. You can use **GetBoxLayout** to fill in a "template" **BOXLAYOUT** structure, then change only the members you need.

Return Value

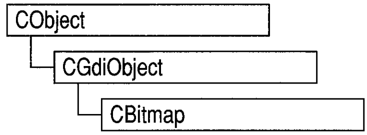
Nonzero if successful; 0 if unsuccessful.

See Also

CBEdit::GetBoxLayout, **WM_HEDITCTL**

class CBitmap : public CGdiObject

The **CBitmap** class encapsulates a Windows graphics device interface (GDI) bitmap and provides member functions to manipulate the bitmap. To use a **CBitmap** object, construct the object, install a bitmap handle in it with one of the initialization member functions, and then call the object's member functions.



#include <afxwin.h>

Construction/Destruction—Public Members

CBitmap	Constructs a **CBitmap** object.

Initialization—Public Members

LoadBitmap	Initializes the object by loading a named bitmap resource from the application's executable file and attaching the bitmap to the object.
LoadOEMBitmap	Initializes the object by loading a predefined Windows bitmap and attaching the bitmap to the object.
CreateBitmap	Initializes the object with a device-dependent memory bitmap that has a specified width, height, and bit pattern.
CreateBitmapIndirect	Initializes the object with a bitmap with the width, height, and bit pattern (if one is specified) given in a **BITMAP** structure.
CreateCompatibleBitmap	Initializes the object with a bitmap so that it is compatible with a specified device.
CreateDiscardableBitmap	Initializes the object with a discardable bitmap that is compatible with a specified device.

Operations—Public Members

FromHandle	Returns a pointer to a **CBitmap** object when given a handle to a Windows **HBITMAP** bitmap.
SetBitmapBits	Sets the bits of a bitmap to the specified bit values.
GetBitmapBits	Copies the bits of the specified bitmap into the specified buffer.

SetBitmapDimension	Assigns a width and height to a bitmap in 0.1-millimeter units.
GetBitmapDimension	Returns the width and height of the bitmap. The height and width are assumed to have been set previously by the **SetBitmapDimension** member function.

Member Functions

CBitmap::CBitmap

CBitmap( );

Remarks

Constructs a **CBitmap** object. The resulting object must be initialized with one of the initialization member functions.

See Also

CBitmap::LoadBitmap, **CBitmap::LoadOEMBitmap**, **CBitmap::CreateBitmap**, **CBitmap::CreateBitmapIndirect**, **CBitmap::CreateCompatibleBitmap**, **CBitmap::CreateDiscardableBitmap**

CBitmap::CreateBitmap

BOOL CreateBitmap(int *nWidth*, **int** *nHeight*, **UINT** *nPlanes*,
UINT *nBitcount*, **const void FAR*** *lpBits* **);**

nWidth Specifies the width (in pixels) of the bitmap.

nHeight Specifies the height (in pixels) of the bitmap.

nPlanes Specifies the number of color planes in the bitmap.

nBitcount Specifies the number of color bits per display pixel.

lpBits Points to a short-integer array that contains the initial bitmap bit values. If it is **NULL**, the new bitmap is left uninitialized.

For more information, see the description of the **bmBits** field in the **BITMAP** structure. In this manual, the **BITMAP** structure is described under the **CBitmap::CreateBitmapIndirect** member function.

Remarks

Initializes a device-dependent memory bitmap that has the specified width, height, and bit pattern. For a color bitmap, either the *nPlanes* or *nBitcount* parameter should be set to 1. If both of these parameters are set to 1, **CreateBitmap** creates a monochrome bitmap. Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a "memory device context" by using **CDC::SelectObject** and copied to any compatible device context by using the **CDC::BitBlt** function.

When you finish with the **CBitmap** object created by the **CreateBitmap** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

Return Value

Nonzero if successful; otherwise 0.

See Also

CDC::SelectObject, **CGdiObject::DeleteObject**, **CDC::BitBlt**, **::CreateBitmap**

CBitmap::CreateBitmapIndirect

BOOL CreateBitmapIndirect(LPBITMAP *lpBitmap* **);**

lpBitmap Points to a **BITMAP** structure that contains information about the bitmap.

Remarks

Initializes a bitmap that has the width, height, and bit pattern (if one is specified) given in the structure pointed to by *lpBitmap*. Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a memory device context by using **CDC::SelectObject** or and copied to any compatible device context by using the **CDC::BitBlt** or **CDC::StretchBlt** function. (The **CDC::PatBlt** function can copy the bitmap for the current brush directly to the display device context.)

If the **BITMAP** structure pointed to by the *lpBitmap* parameter has been filled in by using the **GetObject** function, the bits of the bitmap are not specified and the bitmap is uninitialized. To initialize the bitmap, an application can use a function such as **CDC::BitBlt** or **::SetDIBits** to copy the bits from the bitmap identified by the first parameter of **CGdiObject::GetObject** to the bitmap created by **CreateBitmapIndirect**.

When you finish with the **CBitmap** object created with **CreateBitmapIndirect** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

Return Value

Nonzero if successful; otherwise 0.

BITMAP Structure

A **BITMAP** structure has this form:

```
typedef struct tagBITMAP {  /* bm */
    int     bmType;
    int     bmWidth;
    int     bmHeight;
    int     bmWidthBytes;
    BYTE    bmPlanes;
    BYTE    bmBitsPixel;
    void FAR* bmBits;
} BITMAP;
```

The **BITMAP** structure defines the height, width, color format, and bit values of a logical bitmap.

Members

bmType Specifies the bitmap type. For logical bitmaps, this member must be 0.

bmWidth Specifies the width of the bitmap in pixels. The width must be greater than 0.

bmHeight Specifies the height of the bitmap in raster lines. The height must be greater than 0.

bmWidthBytes Specifies the number of bytes in each raster line. This value must be an even number since the graphics device interface (GDI) assumes that the bit values of a bitmap form an array of integer (2-byte) values. In other words, **bmWidthBytes** * 8 must be the next multiple of 16 greater than or equal to the value obtained when the **bmWidth** member is multiplied by the **bmBitsPixel** member.

bmPlanes Specifies the number of color planes in the bitmap.

bmBitsPixel Specifies the number of adjacent color bits on each plane needed to define a pixel.

bmBits Points to the location of the bit values for the bitmap. The **bmBits** member must be a long pointer to an array of 1-byte values.

Comments

The currently used bitmap formats are monochrome and color. The monochrome bitmap uses a 1-bit, 1-plane format. Each scan is a multiple of 16 bits.

Scans are organized as follows for a monochrome bitmap of height *n*:

```
Scan 0
Scan 1
.
.
.
Scan n-2
Scan n-1
```

The pixels on a monochrome device are either black or white. If the corresponding bit in the bitmap is 1, the pixel is turned on (white). If the corresponding bit in the bitmap is 0, the pixel is turned off (black).

All devices support bitmaps that have the **RC_BITBLT** bit set in the **RASTERCAPS** index of the **GetDeviceCaps** member function.

Each device has its own unique color format. In order to transfer a bitmap from one device to another, use the **GetDIBits** and **SetDIBits** Windows functions.

See Also **CDC::SelectObject**, **CDC::BitBlt**, **CGdiObject::DeleteObject**, **CGdiObject::GetObject**, **::CreateBitmapIndirect**

CBitmap::CreateCompatibleBitmap

BOOL CreateCompatibleBitmap(CDC* *pDC*, **int** *nWidth*, **int** *nHeight* **);**

pDC Specifies the device context.

nWidth Specifies the width (in bits) of the bitmap.

nHeight Specifies the height (in bits) of the bitmap.

Remarks Initializes a bitmap that is compatible with the device specified by *pDC*. The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device context. It can be selected as the current bitmap for any memory device that is compatible with the one specified by *pDC*. If *pDC* is a memory device context, the bitmap returned has the same format as the currently selected bitmap in that device context. A "memory device context" is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual display surface of the compatible device. When a memory device context is created, GDI automatically selects a monochrome stock bitmap for it.

Since a color memory device context can have either color or monochrome bitmaps selected, the format of the bitmap returned by the **CreateCompatibleBitmap** function is not always the same; however, the format of a compatible bitmap for a nonmemory device context is always in the format of the device.

When you finish with the **CBitmap** object created with the **CreateCompatibleBitmap** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

Return Value Nonzero if successful; otherwise 0.

See Also **::CreateCompatibleBitmap**, **CGdiObject::DeleteObject**

CBitmap::CreateDiscardableBitmap

BOOL CreateDiscardableBitmap(CDC* *pDC*, **int** *nWidth*, **int** *nHeight* **);**

pDC Specifies a device context.

nWidth Specifies the width (in bits) of the bitmap.

nHeight Specifies the height (in bits) of the bitmap.

Remarks Initializes a discardable bitmap that is compatible with the device context identified by *pDC*. The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device context. An application can select this bitmap as the current bitmap for a memory device that is compatible with the one specified by *pDC*. Windows can discard a bitmap created by this function only if an application has not selected it into a display context. If Windows discards the bitmap when it is not selected and the application later attempts to select it, the **CDC::SelectObject** function will return **NULL**.

When you finish with the **CBitmap** object created with the **CreateDiscardableBitmap** function, first select the bitmap out of the device context, then delete the **CBitmap** object.

Return Value Nonzero if successful; otherwise 0.

See Also **::CreateDiscardableBitmap**, **CGdiObject::DeleteObject**

CBitmap::FromHandle

static CBitmap* PASCAL FromHandle(HBITMAP *hBitmap* **);**

hBitmap Specifies a Windows GDI bitmap.

Remarks

Returns a pointer to a **CBitmap** object when given a handle to a Windows GDI bitmap. If a **CBitmap** object is not already attached to the handle, a temporary **CBitmap** object is created and attached. This temporary **CBitmap** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

Return Value

A pointer to a **CBitmap** object if successful; otherwise **NULL**.

CBitmap::GetBitmapBits

DWORD GetBitmapBits(DWORD *dwCount*, **LPVOID** *lpBits* **) const;**

dwCount Specifies the number of bytes to be copied.

lpBits Points to the buffer that is to receive the bitmap. The bitmap is an array of bytes. The bitmap byte array conforms to a structure where horizontal scan lines are multiples of 16 bits.

Remarks

Copies the bit pattern of the **CBitmap** object into the buffer pointed to by *lpBits*. The *dwCount* parameter specifies the number of bytes to be copied to the buffer. Use **GetObject** to determine the correct *dwCount* value for the given bitmap.

Return Value

The actual number of bytes in the bitmap, or 0 if there is an error.

See Also

CGdiObject::GetObject, **::GetBitmapBits**

CBitmap::GetBitmapDimension

CSize GetBitmapDimension( ) const;

Remarks

Returns the width and height of the bitmap. The height and width are assumed to have been set previously by using the **SetBitmapDimension** member function.

Return Value

The width and height of the bitmap, measured in 0.1-millimeter units. The height is in the **cy** member of the **CSize** object, and the width is in the **cx** member. If the bitmap width and height have not been set by using **SetBitmapDimension**, the return value is 0.

See Also

CBitmap::SetBitmapDimension, **::GetBitmapDimension**

CBitmap::LoadBitmap

BOOL LoadBitmap(LPCSTR *lpszResourceName* **);**

BOOL LoadBitmap(UINT *nIDResource* **);**

lpszResourceName Points to a null-terminated string that contains the name of the bitmap resource.

nIDResource Specifies the resource ID number of the bitmap resource.

Remarks

Loads the bitmap resource named by *lpszResourceName* or identified by the ID number in *nIDResource* from the application's executable file. The loaded bitmap is attached to the **CBitmap** object. If the bitmap identified by *lpszResourceName* does not exist or if there is insufficient memory to load the bitmap, the function returns 0. An application must call the **CGdiObject::DeleteObject** function to delete any bitmap loaded by the **LoadBitmap** function.

Windows 3.1 Only

The following new bitmaps have been added:

OBM_UPARRROWI
OBM_DNARROWI
OBM_RGARROWI
OBM_LFARROWI

These bitmaps are not found in device drivers for previous versions of Windows. For a complete list of bitmaps and a display of their appearance, see the *Programmer's Reference* in the Windows version 3.1 *Software Development Kit.* ♦

Return Value

Nonzero if successful; otherwise 0.

See Also

CBitmap::LoadOEMBitmap, **::LoadBitmap**, **CGdiObject::DeleteObject**

CBitmap::LoadOEMBitmap

BOOL LoadOEMBitmap(UINT *nIDBitmap* **);**

nIDBitmap ID number of the predefined Windows bitmap. The possible values are listed below from WINDOWS.H:

OBM_BTNCORNERS	**OBM_BTSIZE**
OBM_CHECK	**OBM_CHECKBOXES**
OBM_CLOSE	**OBM_COMBO**
OBM_DNARROW	**OBM_DNARROWD**
OBM_DNARROWI	**OBM_LFARROW**
OBM_LFARROWD	**OBM_LFARROWI**
OBM_MNARROW	**OBM_OLD_CLOSE**
OBM_OLD_DNARROW	**OBM_OLD_LFARROW**
OBM_OLD_REDUCE	**OBM_OLD_RESTORE**
OBM_OLD_RGARROW	**OBM_OLD_UPARROW**
OBM_OLD_ZOOM	**OBM_REDUCE**
OBM_REDUCED	**OBM_RESTORE**
OBM_RESTORED	**OBM_RGARROW**
OBM_RGARROWD	**OBM_RGARROWI**
OBM_SIZE	**OBM_UPARROW**
OBM_UPARROWD	**OBM_UPARROWI**
OBM_ZOOM	**OBM_ZOOMD**

Remarks

Loads a predefined bitmap used by Windows. Bitmap names that begin with **OBM_OLD** represent bitmaps used by Windows versions prior to 3.0. Note that the constant **OEMRESOURCE** must be defined before including WINDOWS.H in order to use any of the **OBM_** constants.

Return Value

Nonzero if successful; otherwise 0.

See Also

CBitmap::LoadBitmap, **::LoadBitmap**

CBitmap::SetBitmapBits

DWORD SetBitmapBits(DWORD *dwCount*, **const void FAR*** *lpBits* **);**

dwCount Specifies the number of bytes pointed to by *lpBits*.

lpBits Points to the **BYTE** array that contains the bit values to be copied to the **CBitmap** object.

Remarks Sets the bits of a bitmap to the bit values given by *lpBits*.

Return Value The number of bytes used in setting the bitmap bits; 0 if the function fails.

See Also **::SetBitmapBits**

CBitmap::SetBitmapDimension

CSize SetBitmapDimension(int *nWidth*, **int** *nHeight* **);**

nWidth Specifies the width of the bitmap (in 0.1-millimeter units).

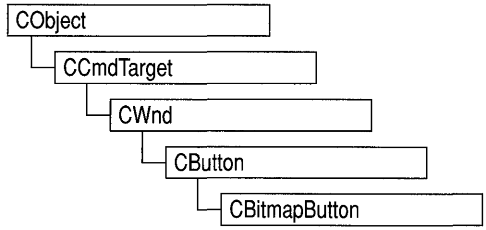
nHeight Specifies the height of the bitmap (in 0.1-millimeter units).

Remarks Assigns a width and height to a bitmap in 0.1-millimeter units. The GDI does not use these values except to return them when an application calls the **GetBitmapDimension** member function.

Return Value The previous bitmap dimensions. Height is in the **cy** member variable of the **CSize** object, and width is in the **cx** member variable.

See Also **CBitmap::GetBitmapDimension**, **::SetBitmapDimension**

class CBitmapButton : public CButton



Use the **CBitmapButton** class to create pushbutton controls labeled with bitmapped images instead of text. **CBitmapButton** objects contain up to four bitmaps, which contain images for the different states a button can assume: up (or normal), down (or selected), focused, and disabled. Only the first bitmap is required; the others are optional.

Bitmap-button images include the border around the image as well as the image itself. The border typically plays a part in showing the state of the button. For example, the bitmap for the focused state usually is like the one for the up state but with a dashed rectangle inset from the border or a thick solid line at the border. The bitmap for the disabled state usually resembles the one for the up state but has lower contrast (like a dimmed or grayed menu selection).

These bitmaps can be of any size, but all are treated as if they were the same size as the bitmap for the up state.

Various applications demand different combinations of bitmap images:

Up	Down	Focused	Disabled	Application
×				Bitmap
×	×			Button without **WS_TABSTOP** style
×	×	×	×	Dialog button with all states
×	×	×		Dialog button with **WS_TABSTOP** style

To create a bitmap-button control in a window's client area, follow these steps:

1. Create one to four bitmap images for the button.
2. Construct the **CBitmapButton** object.
3. Call the **Create** function to create the Windows button control and attach it to the **CBitmapButton** object.
4. Call the **LoadBitmaps** member function to load the bitmap resources after the bitmap button is constructed.

To include a bitmap-button control in a dialog box, follow these steps:

1. Create one to four bitmap images for the button.
2. Create a dialog template with an owner-draw button positioned where you want the bitmap button. The size of the button in the template does not matter.
3. Set the button's caption to a value such as "MYIMAGE" and define a symbol for the button such as IDC_MYIMAGE.
4. In your application's resource script, give each of the images created for the button an ID constructed by appending one of the letters "U," "D," "F," or "X" (for up, down, focused, and disabled) to the string used for the button caption in step 3. For the button caption "MYIMAGE," for example, the IDs would be "MYIMAGEU," "MYIMAGED," "MYIMAGEF," and "MYIMAGEX."
5. In your application's dialog class (derived from **CDialog**), add a **CBitmapButton** member object.
6. In the **CDialog** object's **OnInitDialog** routine, call the **CBitmapButton** object's **AutoLoad** function, using as parameters the button's control ID and the **CDialog** object's **this** pointer.

If you want to handle Windows notification messages, such as **BN_CLICKED**, sent by a bitmap-button control to its parent (usually a class derived from **CDialog**), add to the **CDialog**-derived object a message-map entry and message-handler member function for each message. The notifications sent by a **CBitmapButton** object are the same as those sent by a **CButton** object.

The class **CToolBar** takes a different approach to bitmap buttons. See **CToolBar** for more information.

#include <afxext.h>

See Also **CButton**, **CBitmapButton::AutoLoad**, **CToolBar**

Construction/Destruction — Public Members

CBitmapButton	Constructs a **CBitmapButton** object.
LoadBitmaps	Initializes the object by loading one or more named bitmap resources from the application's resource file and attaching the bitmaps to the object.
AutoLoad	Associates a button in a dialog with an object of the **CBitmapButton** class, loads the bitmap(s) by name, and sizes the button to fit the bitmap.

Operations — Public Members

SizeToContent	Sizes the button to accommodate the bitmap.

Member Functions

CBitmapButton::AutoLoad

BOOL AutoLoad(UINT *nID*, **CWnd*** *pParent* **);**

nID The button's control ID.

pParent Pointer to the object that owns the button.

Remarks

Associates a button in a dialog box with an object of the **CBitmapButton** class, loads the bitmap(s) by name, and sizes the button to fit the bitmap.

Use the **AutoLoad** function to initialize an owner-draw button in a dialog box as a bitmap button. Instructions for using this function are in the remarks for the **CBitmapButton** class.

Return Value

Nonzero if successful; otherwise 0.

See Also

CBitmapButton, **CBitmapButton::LoadBitmaps**, **CBitmapButton::SizeToContent**

CBitmapButton::CBitmapButton

CBitmapButton( );

Remarks

Creates a **CBitmapButton** object.

See Also

CBitmapButton::LoadBitmaps, **CBitmapButton::AutoLoad**, **CBitmapButton::SizeToContent**, **CButton::Create**

CBitmapButton::LoadBitmaps

BOOL LoadBitmaps(LPCSTR *lpszBitmapResource*,
LPCSTR *lpszBitmapResourceSel* = **NULL,**
LPCSTR *lpszBitmapResourceFocus* = **NULL,**
LPCSTR *lpszBitmapResourceDisabled* = **NULL);**

lpszBitmapResource Resource name of the bitmap for a bitmap button's normal or "up" state. Required.

lpszBitmapResourceSel Resource name of the bitmap for a bitmap button's selected or "down" state. May be **NULL**.

lpszBitmapResourceFocus Resource name of the bitmap for a bitmap button's focused state. May be **NULL**.

lpszBitmapResourceDisabled Resource name of the bitmap for a bitmap button's disabled state. May be **NULL**.

Remarks Use this function when you want to load bitmap images identified by their resource names or when you cannot use the **AutoLoad** function because, for example, you are creating a bitmap button that is not part of a dialog box.

Return Value Zero if successful; otherwise nonzero.

See Also **CBitmapButton**, **CBitmapButton::AutoLoad**, **CBitmapButton::SizeToContent**, **CButton::Create**, **CBitmap::LoadBitmap**

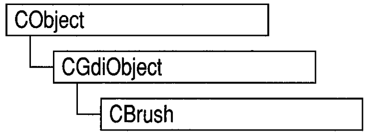
CBitmapButton::SizeToContent

void SizeToContent( );

Remarks Call this function to resize a bitmap button to the size of the bitmap.

See Also **CBitmapButton**, **CBitmapButton::LoadBitmaps**, **CBitmapButton::AutoLoad**

class CBrush : public CGdiObject



The **CBrush** class encapsulates a Windows graphics device interface (GDI) brush. To use a **CBrush** object, construct a **CBrush** object and pass it to any **CDC** member function that requires a brush. Brushes can be solid, hatched, or patterned.

#include <afxwin.h>

See Also **CBitmap**, **CDC**

Construction/Destruction—Public Members

CBrush	Constructs a **CBrush** object.

Initialization—Public Members

CreateSolidBrush	Initializes a brush with the specified solid color.
CreateHatchBrush	Initializes a brush with the specified hatched pattern and color.
CreateBrushIndirect	Initializes a brush with the style, color, and pattern specified in a **LOGBRUSH** structure.
CreatePatternBrush	Initializes a brush with a pattern specified by a bitmap.
CreateDIBPatternBrush	Initializes a brush with a pattern specified by a device-independent bitmap (DIB).

Operations—Public Members

FromHandle	Returns a pointer to a **CBrush** object when given a handle to a Windows **HBRUSH** object.

Member Functions

CBrush::CBrush

CBrush( );

CBrush(COLORREF *crColor* **)**
throw(CResourceException);

CBrush(int *nIndex*, **COLORREF** *crColor* **)**
throw(CResourceException);

CBrush(CBitmap* *pBitmap* **)**
throw(CResourceException);

crColor Specifies the foreground color of the brush as an RGB color. If the brush is hatched, this parameter specifies the color of the hatching.

nIndex Specifies the hatch style of the brush. It can be any one of the following values, with the meaning as given:

- **HS_BDIAGONAL** Downward hatch (left to right) at 45 degrees
- **HS_CROSS** Horizontal and vertical crosshatch
- **HS_DIAGCROSS** Crosshatch at 45 degrees
- **HS_FDIAGONAL** Upward hatch (left to right) at 45 degrees
- **HS_HORIZONTAL** Horizontal hatch
- **HS_VERTICAL** Vertical hatch

pBitmap Points to a **CBitmap** object that specifies a bitmap with which the brush paints.

Remarks

Has four overloaded constructors. The constructor with no arguments constructs an uninitialized **CBrush** object that must be initialized before it can be used. If you use the constructor with no arguments, you must initialize the resulting **CBrush** object with **CreateSolidBrush**, **CreateHatchBrush**, **CreateBrushIndirect**, **CreatePatternBrush**, or **CreateDIBPatternBrush**. If you use one of the constructors that takes arguments, then no further initialization is necessary. The constructors with arguments can throw an exception if errors are encountered, while the constructor with no arguments will always succeed.

The constructor with a single **COLORREF** parameter constructs a solid brush with the specified color. The color specifies an RGB value and can be constructed with the **RGB** macro in WINDOWS.H.

The constructor with two parameters constructs a hatch brush. The *nIndex* parameter specifies the index of a hatched pattern. The *crColor* parameter specifies the color.

The constructor with a **CBitmap** parameter constructs a patterned brush. The parameter identifies a bitmap. The bitmap is assumed to have been created by using **CBitmap::CreateBitmap**, **CBitmap::CreateBitmapIndirect**, **CBitmap::LoadBitmap**, or **CBitmap::CreateCompatibleBitmap**. The minimum size for a bitmap to be used in a fill pattern is 8 pixels by 8 pixels.

See Also **CBitmap::CreateBitmap**, **CBitmap::CreateBitmapIndirect**, **CBitmap::LoadBitmap**, **CBitmap::CreateCompatibleBitmap**, **CBrush::CreateSolidBrush**, **CBrush::CreateHatchBrush**, **CBrush::CreateBrushIndirect**, **CBrush::CreatePatternBrush**, **CBrush::CreateDIBPatternBrush**, **CGdiObject::CreateStockObject**

CBrush::CreateBrushIndirect

BOOL CreateBrushIndirect(LPLOGBRUSH *lpLogBrush* **);**

lpLogBrush Points to a **LOGBRUSH** structure that contains information about the brush.

The **LOGBRUSH** structure has the following form:

```
typedef struct tagLOGBRUSH {
    UINT      lbStyle;
    COLORREF  lbColor;
    int lbHatch;
} LOGBRUSH;
```

Remarks Initializes a brush with a style, color, and pattern specified in a **LOGBRUSH** structure. The brush can subsequently be selected as the current brush for any device context. A brush created using a monochrome (1 plane, 1 bit per pixel) bitmap is drawn using the current text and background colors. Pixels represented by a bit set to 0 will be drawn with the current text color. Pixels represented by a bit set to 1 will be drawn with the current background color.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CBrush::CreateDIBPatternBrush**, **CBrush::CreatePatternBrush**, **CBrush::CreateSolidBrush**, **CBrush::CreateHatchBrush**, **CGdiObject::CreateStockObject**, **CGdiObject::DeleteObject**, **::CreateBrushIndirect**

CBrush::CreateDIBPatternBrush

BOOL CreateDIBPatternBrush(HGLOBAL *hPackedDIB*, **UINT** *nUsage* **);**

hPackedDIB Identifies a global-memory object containing a packed device-independent bitmap (DIB).

nUsage Specifies whether the **bmiColors[]** fields of the **BITMAPINFO** data structure contain explicit RGB values or indexes into the currently realized logical palette. The parameter must be one of the following values, with the meaning as given:

- **DIB_PAL_COLORS** The color table consists of an array of 16-bit indexes.
- **DIB_RGB_COLORS** The color table contains literal RGB values.

Remarks

Initializes a brush with the pattern specified by a device-independent bitmap (DIB). The brush can subsequently be selected for any device context that supports raster operations. To obtain a handle to the DIB, call the Windows **GlobalAlloc** function to allocate a block of global memory and then fill the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** data structure immediately followed by the array of bytes that define the pixels of the bitmap.

The **BITMAPINFO** structure has the following form:

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER    bmiHeader;
    RGBQUAD             bmiColors[1];
} BITMAPINFO;
```

Bitmaps used as fill patterns should be 8 pixels by 8 pixels. If the bitmap is larger, the Windows operating system creates a fill pattern using only the bits corresponding to the first 8 rows and 8 columns of pixels in the upper-left corner of the bitmap.

When an application selects a two-color DIB pattern brush into a monochrome device context, the Windows operating system ignores the colors specified in the DIB and instead displays the pattern brush using the current text and background colors of the device context. Pixels mapped to the first color (at offset 0 in the DIB color table) of the DIB are displayed using the text color. Pixels mapped to the second color (at offset 1 in the color table) are displayed using the background color.

Return Value

Nonzero if successful; otherwise 0.

See Also

CBrush::CreatePatternBrush, **CBrush::CreateBrushIndirect**, **CBrush::CreateSolidBrush**, **CBrush::CreateHatchBrush**, **CGdiObject::CreateStockObject**, **::CreateDIBPatternBrush**, **::GlobalAlloc**

CBrush::CreateHatchBrush

BOOL CreateHatchBrush(int *nIndex*, **COLORREF** *crColor* **);**

nIndex Specifies the hatch style of the brush. It can be one of the following values, with the meaning as given:

- **HS_BDIAGONAL** Downward hatch (left to right) at 45 degrees
- **HS_CROSS** Horizontal and vertical crosshatch
- **HS_DIAGCROSS** Crosshatch at 45 degrees
- **HS_FDIAGONAL** Upward hatch (left to right) at 45 degrees
- **HS_HORIZONTAL** Horizontal hatch
- **HS_VERTICAL** Vertical hatch

crColor Specifies the foreground color of the brush as an RGB color (the color of the hatches).

Remarks

Initializes a brush with the specified hatched pattern and color. The brush can subsequently be selected as the current brush for any device context.

Return Value

Nonzero if successful; otherwise 0.

See Also

CBrush::CreateBrushIndirect, **CBrush::CreateDIBPatternBrush**, **CBrush::CreatePatternBrush**, **CBrush::CreateSolidBrush**, **CGdiObject::CreateStockObject**, **::CreateHatchBrush**

CBrush::CreatePatternBrush

BOOL CreatePatternBrush(CBitmap* *pBitmap* **);**

pBitmap Identifies a bitmap.

Remarks

Initializes a brush with a pattern specified by a bitmap. The brush can subsequently be selected for any device context that supports raster operations. The *pBitmap* bitmap is typically initialized using the **CBitmap** functions **CreateBitmap**, **CreateBitmapIndirect**, **LoadBitmap**, or **CreateCompatibleBitmap**. Bitmaps used as fill patterns should be 8 pixels by 8 pixels. If the bitmap is larger, Windows will only use the bits corresponding to the first 8 rows and columns of pixels in the bitmap's upper-left corner. A pattern brush can be deleted without affecting the associated bitmap, so the bitmap can be used to create any number of pattern brushes. A brush created using a monochrome bitmap (1 color plane, 1 bit per pixel) is drawn using the current text and background colors. Pixels represented by a bit set to 0 are drawn with the current text color. Pixels represented by a bit set to 1 are drawn with the current background color.

Return Value Nonzero if successful; otherwise 0.

See Also **CBrush::CreateBrushIndirect**, **CBrush::CreateDIBPatternBrush**, **CBrush::CreateHatchBrush**, **CBrush::CreateSolidBrush**, **CGdiObject::CreateStockObject**, **CBitmap::CreateBitmap**, **CBitmap::CreateBitmapIndirect**, **CBitmap::CreateCompatibleBitmap**, **CBitmap::LoadBitmap**, **::CreatePatternBrush**

CBrush::CreateSolidBrush

BOOL CreateSolidBrush(COLORREF *crColor* **);**

crColor Specifies the color of the brush. The color specifies an RGB value and can be constructed with the **RGB** macro in WINDOWS.H.

Remarks Initializes a brush with a specified solid color. The brush can then be selected as the current brush for any device context. When an application finishes using the brush created by **CreateSolidBrush**, it should select the brush out of the device context.

Return Value Nonzero if successful; otherwise 0.

See Also **CBrush::CreateBrushIndirect**, **CBrush::CreateDIBPatternBrush**, **CBrush::CreateHatchBrush**, **CBrush::CreatePatternBrush**, **::CreateSolidBrush**, **CGdiObject::DeleteObject**

CBrush::FromHandle

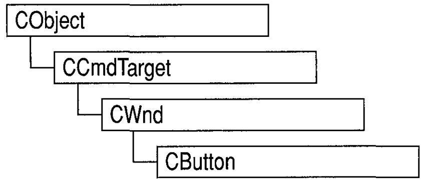
static CBrush* PASCAL FromHandle(HBRUSH *hBrush* **);**

hBrush **HANDLE** to a Windows GDI brush.

Remarks Returns a pointer to a **CBrush** object when given a handle to a Windows **HBRUSH** object. If a **CBrush** is not already attached to the handle, a temporary **CBrush** is created and attached. This temporary **CBrush** is valid only until the next time the application has idle time in its event loop. At this time, all temporary graphic objects are deleted. In other words the temporary object is only valid during the processing of one window message.

Return Value A pointer to a **CBrush** object if successful; **NULL** if not.

class CButton : public CWnd



The **CButton** class provides the functionality of Windows button controls. A button control is a small, rectangular child window that can be clicked on and off. Buttons can be used alone or in groups and can either be labeled or appear without text. A button typically changes appearance when the user clicks it. Typical buttons are the check box, radio button, and pushbutton. A **CButton** object can become any of these, according to the style specified at its initialization by the **Create** member function.

In addition, the **CBitmapButton** class derived from **CButton** supports creation of button controls labeled with bitmap images instead of text. A **CBitmapButton** can have separate bitmaps for a button's up, down, focused, and disabled states.

You can create a button control either from a dialog template or directly in your code. In both cases, first call the constructor **CButton** to construct the **CButton** object; then call the **Create** member function to create the Windows button control and attach it to the **CButton** object. Construction can be a one-step process in a class derived from **CButton**. Write a constructor for the derived class and call **Create** from within the constructor.

If you want to handle Windows notification messages sent by a button control to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification(*id, memberFxn*)

where *id* specifies the child window ID of the control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void memberFxn( );

Potential message-map entries are:

Map Entry	Sent To Parent When . . .
ON_BN_CLICKED	The user clicks a button.
ON_BN_DOUBLECLICKED	The user double-clicks a button.

If you create a **CButton** object from a dialog resource using App Studio, the **CButton** object is automatically destroyed when the user closes the dialog box.

If you create a **CButton** object within a window, you may need to destroy it. If you create the **CButton** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user closes the Windows button control. If you create the **CButton** object on the stack, or it is embedded in the parent dialog object, it is destroyed automatically.

#include <afxwin.h>

See Also **CWnd**, **CComboBox**, **CEdit**, **CListBox**, **CScrollBar**, **CStatic**, **CBitmapButton**, **CDialog**

Construction/Destruction—Public Members

CButton	Constructs a **CButton** object.

Initialization—Public Members

Create	Creates the Windows button control and attaches it to the **CButton** object.

Operations—Public Members

GetState	Retrieves the check state, highlight state, and focus state of a button control.
SetState	Sets the highlighting state of a button control.
GetCheck	Retrieves the check state of a button control.
SetCheck	Sets the check state of a button control.
GetButtonStyle	Retrieves information about the button control style.
SetButtonStyle	Changes the style of a button.

Overridables—Public Members

DrawItem	Override to draw an owner-drawn **CButton** object.

Member Functions

CButton::CButton

CButton( );

Remarks Constructs a **CButton** object.

See Also **CButton::Create**

CButton::Create

BOOL Create(LPCSTR *lpszCaption*, **DWORD** *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

lpszCaption Specifies the button control's text.

dwStyle Specifies the button control's style.

rect Specifies the button control's size and position. It can be either a **CRect** object or a **RECT** structure.

pParentWnd Specifies the button control's parent window, usually a **CDialog** or **CModalDialog**. It must not be **NULL**.

nID Specifies the button control's ID.

Remarks You construct a **CButton** object in two steps. First call the constructor, then call **Create**, which creates the Windows button control and attaches it to the **CButton** object.

If the **WS_VISIBLE** style is given, Windows sends the button control all the messages required to activate and show the button.

Apply the following window styles to a button control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_GROUP** To group controls
- **WS_TABSTOP** To include the button in the tabbing order

See the **CreateEx** member function in the **CWnd** base class for a full description of these window styles.

Return Value Nonzero if successful; otherwise 0.

Button Styles You can use any combination of the following button styles for *dwStyle*:

- **BS_AUTOCHECKBOX** Same as a check box, except that an **X** appears in the check box when the user selects the box; the **X** disappears the next time the user selects the box.
- **BS_AUTORADIOBUTTON** Same as a radio button, except that when the user selects it, the button automatically highlights itself and removes the selection from any other radio buttons with the same style in the same group.
- **BS_AUTO3STATE** Same as a three-state check box, except that the box changes its state when the user selects it.
- **BS_CHECKBOX** Creates a small square that has text displayed to its right (unless this style is combined with the **BS_LEFTTEXT** style).
- **BS_DEFPUSHBUTTON** Creates a button that has a heavy black border. The user can select this button by pressing the ENTER key. This style enables the user to quickly select the most likely option (the default option).
- **BS_GROUPBOX** Creates a rectangle in which other buttons can be grouped. Any text associated with this style is displayed in the rectangle's upper-left corner.
- **BS_LEFTTEXT** When combined with a radio-button or check-box style, the text appears on the left side of the radio button or check box.
- **BS_OWNERDRAW** Creates an owner-drawn button. The framework calls the **DrawItem** member function when a visual aspect of the button has changed. This style must be set when using the **CBitmapButton** class.
- **BS_PUSHBUTTON** Creates a pushbutton that posts a **WM_COMMAND** message to the owner window when the user selects the button.
- **BS_RADIOBUTTON** Creates a small circle that has text displayed to its right (unless this style is combined with the **BS_LEFTTEXT** style). Radio buttons are usually used in groups of related but mutually exclusive choices.
- **BS_3STATE** Same as a check box, except that the box can be dimmed as well as checked. The dimmed state typically is used to show that a check box has been disabled.

See Also **CButton::CButton**

CButton::DrawItem

virtual void DrawItem(LPDRAWITEMSTRUCT *lpDrawItemStruct* **);**

lpDrawItemStruct A long pointer to a **DRAWITEMSTRUCT** structure. The structure contains information about the item to be drawn and the type of drawing required.

Remarks Called by the framework when a visual aspect of an owner-drawn button has changed. An owner-drawn button has the **BS_OWNERDRAW** style set. Override this member function to implement drawing for an owner-drawn **CButton** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before the member function terminates.

See the **Create** member function for a list of button styles.

See Also **WM_DRAWITEM**, **CButton::SetButtonStyle**

CButton::GetButtonStyle

UINT GetButtonStyle( ) const;

Remarks Retrieves the window style of **CButton**. It only returns the **BS_** style values, not any of the other window styles.

See the **Create** member function for a list of button styles.

See Also **::GetWindowLong**, **CButton::SetButtonStyle**

CButton::GetCheck

int GetCheck( ) const;

Remarks Retrieves the check state of a radio button or check box.

Return Value

The return value from a button control created with the **BS_AUTOCHECKBOX**, **BS_AUTORADIOBUTTON**, **BS_AUTO3STATE**, **BS_CHECKBOX**, **BS_RADIOBUTTON**, or **BS_3STATE** style is one of the following values:

Value	Meaning
0	Button state is unchecked.
1	Button state is checked.
2	Button state is indeterminate (only applies if the button has the **BS_3STATE** or **BS_AUTO3STATE** style).

If the button has any other style, the return value is 0.

See Also

CButton::GetState, **CButton::SetState**, **CButton::SetCheck**, **BM_GETCHECK**

CButton::GetState

UINT GetState( ) const;

Return Value

Specifies the current state of the button control. You can use the following masks against the return value to extract information about the state:

Mask	Meaning
0x0003	Specifies the check state (radio buttons and check boxes only). A 0 indicates the button is unchecked. A 1 indicates the button is checked. A radio button is checked when it contains a bullet (•). A check box is checked when it contains an **X**. A 2 indicates the check state is indeterminate (three-state check boxes only). The state of a three-state check box is indeterminate when it contains a halftone pattern.
0x0004	Specifies the highlight state. A nonzero value indicates that the button is highlighted. A button is highlighted when the user clicks and holds the left mouse button. The highlighting is removed when the user releases the mouse button.
0x0008	Specifies the focus state. A nonzero value indicates that the button has the focus.

See Also

CButton::GetCheck, **CButton::SetCheck**, **CButton::SetState**, **BM_GETSTATE**

CButton::SetButtonStyle

void SetButtonStyle(UINT *nStyle*, **BOOL** *bRedraw* = **TRUE);**

nStyle Specifies the button style.

bRedraw Specifies whether the button is to be redrawn. A nonzero value redraws the button. A 0 value does not redraw the button. The button is redrawn by default.

Remarks

Changes the style of a button. Use the **GetButtonStyle** member function to retrieve the button style. The low-order word of the complete button style is the button-specific style.

See the **Create** member function for a list of possible button styles.

See Also

CButton::GetButtonStyle, **BM_SETSTYLE**

CButton::SetCheck

void SetCheck(int *nCheck* **);**

nCheck Specifies the check state. This parameter can be one of the following:

Value	Meaning
0	Set the button state to unchecked.
1	Set the button state to checked.
2	Set the button state to indeterminate. This value can be used only if the button has the **BS_3STATE** or **BS_AUTO3STATE** style.

Remarks

Sets or resets the check state of a radio button or check box. This member function has no effect on a pushbutton.

See Also

CButton::GetCheck, **CButton::GetState**, **CButton::SetState**, **BM_SETCHECK**

CButton::SetState

void SetState(BOOL *bHighlight* **);**

bHighlight Specifies whether the button is to be highlighted. A nonzero value highlights the button; a 0 value removes any highlighting.

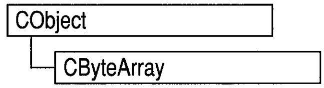
Remarks

Sets the highlighting state of a button control. Highlighting affects the exterior of a button control. It has no effect on the check state of a radio button or check box. A button control is automatically highlighted when the user clicks and holds the left mouse button. The highlighting is removed when the user releases the mouse button.

See Also

CButton::GetState, **CButton::SetCheck**, **CButton::GetCheck**, **BM_SETSTATE**

class CByteArray : public CObject



The **CByteArray** class supports dynamic arrays of bytes. The member functions of **CByteArray** are similar to the member functions of class **CObArray**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **BYTE**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
BYTE CByteArray::GetAt( int <nIndex> ) const;
```

CByteArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of bytes is stored to an archive, either with the overloaded insertion (<<) operator or with the **Serialize** member function, each element is, in turn, serialized. If you need debug output from individual elements in the array, you must set the depth of the **CDumpContext** object to 1 or greater.

#include <afxcoll.h>

See Also **CObArray**

Construction/Destruction—Public Members

CByteArray	Constructs an empty array for bytes.
~CByteArray	Destroys a **CByteArray** object.

Bounds—Public Members

GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations—Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; array not allowed to grow.
ElementAt	Returns a temporary reference to the byte within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

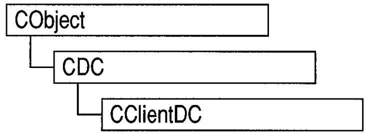
Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

class CClientDC : public CDC



The **CClientDC** class is derived from **CDC** and takes care of calling the Windows functions **GetDC** at construction time and **ReleaseDC** at destruction time. This means that the device context associated with a **CClientDC** object is the client area of a window.

#include <afxwin.h>

See Also **CDC**

Construction/Destruction—Public Members

CClientDC	Constructs a **CClientDC** object connected to the **CWnd**.

Data Members—Protected Members

m_hWnd	The **HWND** of the window for which this **CClientDC** is valid.

Member Functions

CClientDC::CClientDC

CClientDC(CWnd* *pWnd* **)**
throw(CResourceException);

pWnd The window whose client area the device context object will access.

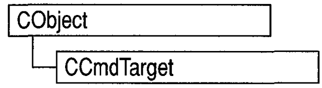
Remarks Constructs a **CClientDC** object that accesses the client area of the **CWnd** pointed to by *pWnd*. The constructor calls the Windows function **GetDC**. An exception (of type **CResourceException**) is thrown if the Windows **GetDC** call fails. A device context may not be available if Windows has already allocated all of its available device contexts. Your application competes for the five common display contexts available at any given time under the Windows operating system.

Data Members

CClientDC::m_hWnd

Remarks The **HWND** of the **CWnd** pointer used to construct the **CClientDC** object. **m_hWnd** is a protected variable.

class CCmdTarget : public CObject



CCmdTarget is the base class for the Microsoft Foundation Class Library message-map architecture. A message map routes commands or messages to the member functions you write to handle them. (A command is a message from a menu item, command button, or accelerator key.)

Key framework classes derived from **CCmdTarget** include **CView**, **CWinApp**, **CDocument**, **CWnd**, and **CFrameWnd**. If you intend for a new class to handle messages, derive the class from one of these **CCmdTarget**-derived classes. You will rarely derive a class from **CCmdTarget** directly.

For an overview of command targets and **OnCmdMsg** routing, see Chapter 3 in this manual.

CCmdTarget includes member functions that handle the display of an hourglass cursor. Display the hourglass cursor when you expect a command to take a noticeable time interval to execute.

include <afxwin.h>

See Also **CCmdUI**, **CDocument**, **CDocTemplate**, **CWinApp**, **CWnd**, **CView**, **CFrameWnd**

Operations — Public Members

BeginWaitCursor	Displays the cursor as an hourglass cursor.
EndWaitCursor	Returns to the previous cursor.
RestoreWaitCursor	Restores the hourglass cursor.

Overridables — Public Members

OnCmdMsg	Routes and dispatches command messages.

Member Functions

CCmdTarget::BeginWaitCursor

void BeginWaitCursor( );

Remarks

Call this function to display the cursor as an hourglass when you expect a command to take a noticeable time interval to execute. The framework calls this function to show the user that it is busy, such as when a **CDocument** object loads or saves itself to a file.

Call **EndWaitCursor** to restore the previous cursor.

See Also

CCmdTarget::EndWaitCursor, **CCmdTarget::RestoreWaitCursor**, **CWinApp::DoWaitCursor**

CCmdTarget::EndWaitCursor

void EndWaitCursor( );

Remarks

Call this function after you have called the **BeginWaitCursor** member function to return from the hourglass cursor to the previous cursor. The framework also calls this member function after it has invoked the hourglass cursor.

See Also

CCmdTarget::BeginWaitCursor, **CCmdTarget::RestoreWaitCursor**, **CWinApp::DoWaitCursor**

CCmdTarget::OnCmdMsg

virtual BOOL OnCmdMsg(UINT *nID*, **int** *nCode*, **void*** *pExtra*,
AFX_CMDHANDLERINFO* *pHandlerInfo* **);**

nID Contains the command ID.

nCode Identifies the command notification code.

pExtra Used according to the value of *nCode*.

pHandlerInfo If not **NULL**, **OnCmdMsg** fills in the *pHandlerInfo* structure with the **pTarget** and **pmf** members of the **CMDHANDLERINFO** structure instead of dispatching the command. Typically, this parameter should be **NULL**.

Remarks

Called by the framework to route and dispatch command messages and to handle the update of command user-interface objects. This is the main implementation routine of the framework command architecture.

At run time, **OnCmdMsg** dispatches a command to other objects or handles the command itself by calling the root class **CCmdTarget::OnCmdMsg**, which does the actual message-map lookup. For a complete description of the default command routing, see Chapter 6 in the *Class Library User's Guide*.

On rare occasions, you may want to override this member function to extend the framework's standard command routing. Please refer to Technical Note 21 in MSVC\HELP\MFCNOTES.HLP for advanced details of the command-routing architecture.

Return Value

Nonzero if the message is handled; otherwise 0.

See Also

CCmdUI

CCmdTarget::RestoreWaitCursor

void RestoreWaitCursor( );

Remarks

Call this function to restore the appropriate hourglass cursor after the system cursor has changed (for example, after a message box has opened and then closed while in the middle of a lengthy operation).

See Also

CCmdTarget::EndWaitCursor, **CCmdTarget::BeginWaitCursor**, **CWinApp::DoWaitCursor**

class CCmdUI

The **CCmdUI** class is used only within an **ON_UPDATE_COMMAND_UI** handler in a **CCmdTarget**-derived class.

When a user of your application pulls down a menu, each menu item needs to know whether it should be displayed as enabled or disabled (dimmed). The target of a menu command provides this information by implementing an **ON_UPDATE_COMMAND_UI** handler. Use ClassWizard to browse the command user-interface objects in your application and create a message-map entry and function prototype for each handler.

When the menu is pulled down, the framework searches for and calls each **ON_UPDATE_COMMAND_UI** handler, each handler calls **CCmdUI** member functions such as **Enable** and **Check**, and the framework then appropriately displays each menu item.

A menu item can be replaced with a control-bar button or other command user-interface object without changing the code within the **ON_UPDATE_COMMAND_UI** handler.

Table R.1 summarizes the effect **CCmdUI**'s member functions have on various command user-interface items.

Table R.1 Using CCmdUI Member Functions

User-Interface Item	Enable	SetCheck	SetRadio	SetText
Menu item	Enables or disables	Checks (✓) or unchecks	Checks using dot (●)	Sets item text
Toolbar button	Enables or disables	Selects, unselects, or indeterminate	Same as **SetCheck**	(Not applicable)
Status-bar pane	Makes text visible or invisible	Sets pop-out or normal border	Same as **SetCheck**	Sets pane text
Normal button in **CDialogBar**	Enables or disables	Checks or unchecks check box	Same as **SetCheck**	Sets button text
Normal control in **CDialogBar**	Enables or disables	(Not applicable)	(Not applicable)	Sets window text

For more on the use of this class, see Chapter 6 in the *Class Library User's Guide* and Chapter 3 in this manual.

#include <afxwin.h>

See Also **CCmdTarget**

Operations — Public Members

Enable	Enables or disables the user-interface item for this command.
SetCheck	Sets the check state of the user-interface item for this command.
SetRadio	Like the **SetCheck** member function, but operates on radio groups.
SetText	Sets the text for the user-interface item for this command.
ContinueRouting	Tells the command-routing mechanism to continue routing the current message down the chain of handlers.

Member Functions

CCmdUI::ContinueRouting

void ContinueRouting( );

Remarks Call this member function to tell the command-routing mechanism to continue routing the current message down the chain of handlers.

This is an advanced member function that should be used in conjunction with an **ON_COMMAND_EX** handler that returns **FALSE**. For more information, see Technical Note 21 in MSVC\HELP\MFCNOTES.HLP.

CCmdUI::Enable

virtual void Enable(BOOL *bOn* **= TRUE);**

bOn **TRUE** to enable the item, **FALSE** to disable it.

Remarks Call this member function to enable or disable the user-interface item for this command.

See Also **CCmdUI::SetCheck**

CCmdUI::SetCheck

virtual void SetCheck(int *nCheck* **= 1);**

nCheck Specifies the check state to set. If 0, unchecks; if 1, checks; and if 2, sets indeterminate.

Remarks

Call this member function to set the user-interface item for this command to the appropriate check state. This member function works for menu items and toolbar buttons. The indeterminate state applies only to toolbar buttons.

See Also

CCmdUI::SetRadio

CCmdUI::SetRadio

virtual void SetRadio(BOOL *bOn* **= TRUE);**

bOn **TRUE** to enable the item; otherwise **FALSE**.

Remarks

Call this member function to set the user-interface item for this command to the appropriate check state. This member function operates like **SetCheck**, except that it operates on user-interface items acting as part of a radio group. Unchecking the other items in the group is not automatic unless the items themselves maintain the radio-group behavior.

See Also

CCmdUI::SetCheck

CCmdUI::SetText

virtual void SetText(LPCSTR *lpszText* **);**

lpszText A pointer to a text string.

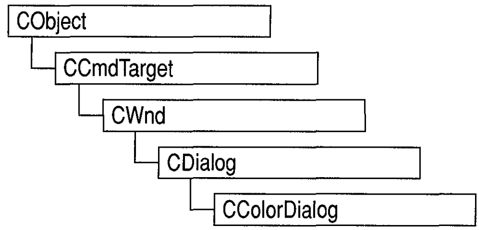
Remarks

Call this member function to set the text of the user-interface item for this command.

See Also

CCmdUI::Enable

class CColorDialog : public CDialog



The **CColorDialog** class allows you to incorporate a color-selection dialog box into your application. A **CColorDialog** object is a dialog box with a list of colors that are defined for the display system. The user can select or create a particular color from the list, which is then reported back to the application when the dialog box exits.

To construct a **CColorDialog** object, use the provided constructor or derive a new class and use your own custom constructor.

Once the dialog box has been constructed, you can set or modify any values in the **m_cc** structure to initialize the values of the dialog box's controls. The **m_cc** structure is of type **CHOOSECOLOR**. For more information on this structure, see the *Windows Software Development Kit* (SDK) documentation.

After initializing the dialog box's controls, call the **DoModal** member function to display the dialog box and allow the user to select a color. **DoModal** returns the user's selection of either the dialog box's OK (**IDOK**) or Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CColorDialog**'s member functions to retrieve the information input by the user.

You can use the Windows **CommDlgExtendedError** function to determine if an error occurred during initialization of the dialog box and to learn more about the error. For more information on this function, see the Windows SDK documentation.

CColorDialog relies on the COMMDLG.DLL file that ships with Windows version 3.1. For details about redistributing COMMDLG.DLL to Windows version 3.0 users, see the *Getting Started* manual for the Windows version 3.1 SDK.

To customize the dialog box, derive a class from **CColorDialog**, provide a custom dialog template, and add a message map to process notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

Note On some installations the **CColorDialog** object will not display with a gray background if you have used the framework to make other **CDialog** objects gray.

#include <afxdlgs.h>

Data Members—Public Members

clrSavedCustom	An array of RGB values used to store custom colors.
m_cc	A structure used to customize the settings of the dialog box.

Construction/Destruction—Public Members

CColorDialog	Constructs a **CColorDialog** object.

Operations—Public Members

DoModal	Displays a color dialog box and allows the user to make a selection.
GetColor	Returns a **COLORREF** structure containing the values of the selected color.
SetCurrentColor	Forces the current color selection to the specified color.

Overridables—Protected Members

OnColorOK	Override to validate the color entered into the dialog box.

Member Functions

CColorDialog::CColorDialog

CColorDialog(COLORREF *clrInit* **= 0, DWORD** *dwFlags* **= 0,**
CWnd* *pParentWnd* **= NULL);**

clrInit The default color selection. If no value is specified, the default is RGB(0,0,0) (black).

dwFlags A set of flags that customize the function and appearance of the dialog box. For more information, see the **CHOOSECOLOR** structure in the Windows SDK documentation.

pParentWnd A pointer to the dialog box's parent or owner window.

Remarks Constructs a **CColorDialog** object.

See Also **CDialog::DoModal**, **::ChooseColor**

CColorDialog::DoModal

virtual int DoModal( );

Remarks

Call this function to display the Windows common color dialog box and allow the user to select a color.

If you want to initialize the various color dialog-box options by setting members of the **m_cc** structure, you should do this before calling **DoModal** but after the dialog-box object is constructed.

After calling **DoModal**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

Return Value

IDOK or **IDCANCEL** if the function is successful; otherwise 0. **IDOK** and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

If **IDCANCEL** is returned, you can call the Windows **CommDlgExtendedError** function to determine if an error occurred.

See Also

CDialog::DoModal, **CColorDialog::CColorDialog**

CColorDialog::GetColor

COLORREF GetColor( ) const;

Remarks

Call this function after calling **DoModal** to retrieve the information about the color the user selected.

Return Value

A **COLORREF** value that contains the RGB information for the color selected in the color dialog box.

See Also

CColorDialog::SetCurrentColor

CColorDialog::OnColorOK

Protected **virtual BOOL OnColorOK( );** ♦

Remarks Override this function only if you want to provide custom validation of the color entered into the dialog box. This function allows you to reject a color entered by a user into a common color dialog box for any application-specific reason. Normally, you do not need to use this function because the framework provides default validation of colors and displays a message box if an invalid color is entered.

Use the **GetColor** member function to get the RGB value of the color.

If 0 is returned, the dialog box will remain displayed in order for the user to enter another filename.

Return Value Nonzero if the dialog box should not be dismissed; otherwise 0 to accept the color that was entered.

CColorDialog::SetCurrentColor

void SetCurrentColor(COLORREF *clr* **);**

clr An RGB color value.

Remarks Call this function after calling **DoModal** to force the current color selection to the color value specified in *clr*. This function is called from within a message handler or **OnColorOK**. The dialog box will automatically update the user's selection based on the value of the *clr* parameter.

See Also **CColorDialog::GetColor**

Data Members

CColorDialog::clrSavedCustom

static COLORREF clrSavedCustom[16];

Remarks

In addition to choosing colors, **CColorDialog** objects permit the user to define up to 16 custom colors. The **clrSavedCustom** member is an array of 16 RGB color values that stores these custom colors between invocations of the **CColorDialog** object. These colors can be retrieved after **DoModal** returns **IDOK**.

Each of the 16 RGB values in **clrSavedCustom** is initialized to RGB(255,255,255) (white). The **clrSavedCustom** member only allows you to save custom colors between dialog box invocations within the application. If you wish to save these colors between invocations of the application, you must save them in some other manner, such as in an initialization (.INI) file. Typically, this saving is done in your application's **ExitInstance** function.

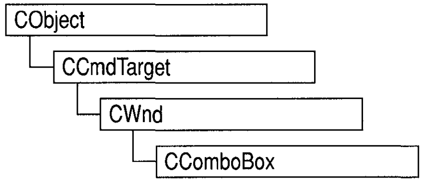
CColorDialog::m_cc

CHOOSECOLOR m_cc;

Remarks

A structure of type **CHOOSECOLOR**, whose members store the characteristics and values of the dialog box. After constructing a **CColorDialog** object, you can use **m_cc** to set various aspects of the dialog box before calling the **DoModal** member function.

class CComboBox : public CWnd



The **CComboBox** class provides the functionality of a Windows combo box.

A combo box consists of a list box combined with either a static control or edit control. The list-box portion of the control may be displayed at all times or may only drop down when the user selects the drop-down arrow next to the control.

The currently selected item (if any) in the list box is displayed in the static or edit control. In addition, if the combo box has an edit control, the user can type text in the edit control and the list box, if it is visible, will highlight the first selection that matches the typed entry.

The following table compares the three combo-box styles:

Style	When Is List Box Visible?	Static or Edit Control?
Simple	Always	Edit
Drop-down	When dropped down	Edit
Drop-down list	When dropped down	Static

You can create a **CComboBox** object from either a dialog template or directly in your code. In both cases, first call the constructor **CComboBox** to construct the **CComboBox** object; then call the **Create** member function to create the control and attach it to the **CComboBox** object. If you want to handle Windows notification messages sent by a combo box to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification(*id, memberFxn*)

where *id* specifies the child-window ID of the combo-box control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void memberFxn(**);**

The order in which certain notifications will be sent cannot be predicted. In particular, a **CBN_SELCHANGE** notification may occur either before or after a **CBN_CLOSEUP** notification.

Potential message-map entries are:

Windows 3.1 Only

- **ON_CBN_CLOSEUP** The list box of a combo box has closed. This notification message is not sent for a combo box that has the **CBS_SIMPLE** style. ♦
- **ON_CBN_DBLCLK** The user double-clicks a string in the list box of a combo box. This notification message is only sent for a combo box with the **CBS_SIMPLE** style. For a combo box with the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style, a double-click cannot occur because a single click hides the list box.
- **ON_CBN_DROPDOWN** The list box of a combo box is about to drop down (be made visible). This notification message can occur only for a combo box with the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.
- **ON_CBN_EDITCHANGE** The user has taken an action that may have altered the text in the edit-control portion of a combo box. Unlike the **CBN_EDITUPDATE** message, this message is sent after the Windows operating system updates the screen. It is not sent if the combo box has the **CBS_DROPDOWNLIST** style.
- **ON_CBN_EDITUPDATE** The edit-control portion of a combo box is about to display altered text. This notification message is sent after the control has formatted the text but before it displays the text. It is not sent if the combo box has the **CBS_DROPDOWNLIST** style.
- **ON_CBN_ERRSPACE** The combo box cannot allocate enough memory to meet a specific request.

Windows 3.1 Only

- **ON_CBN_SELENDCANCEL** Indicates the user's selection should be canceled. The user clicks an item and then clicks another window or control to hide the list box of a combo box. This notification message is sent before the **CBN_CLOSEUP** notification message to indicate that the user's selection should be ignored. The **CBN_SELENDCANCEL** or **CBN_SELENDOK** notification message is sent even if the **CBN_CLOSEUP** notification message is not sent (as in the case of a combo box with the **CBS_SIMPLE** style).
- **ON_CBN_SELENDOK** The user selects an item and then either presses the ENTER key or clicks the DOWN ARROW key to hide the list box of a combo box. This notification message is sent before the **CBN_CLOSEUP** message to indicate that the user's selection should be considered valid. The **CBN_SELENDCANCEL** or **CBN_SELENDOK** notification message is sent even if the **CBN_CLOSEUP** notification message is not sent (as in the case of a combo box with the **CBS_SIMPLE** style). ♦
- **ON_CBN_KILLFOCUS** The combo box is losing the input focus.

- **ON_CBN_SELCHANGE** The selection in the list box of a combo box is about to be changed as a result of the user either clicking in the list box or changing the selection by using the arrow keys.
- **ON_CBN_SETFOCUS** The combo box receives the input focus.

If you create a **CComboBox** object within a dialog box (through a dialog resource with App Studio), the **CComboBox** object is automatically destroyed when the user closes the dialog box. If you embed a **CComboBox** object within another window object, you do not need to destroy it. If you create the **CComboBox** object on the stack, it is destroyed automatically. If you create the **CComboBox** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the Windows combo box is destroyed.

#include <afxwin.h>

See Also **CWnd**, **CButton**, **CEdit**, **CListBox**, **CScrollBar**, **CStatic**, **CDialog**

Construction/Destruction—Public Members

CComboBox	Constructs a **CComboBox** object.

Initialization—Public Members

Create	Creates the combo box and attaches it to the **CComboBox** object.

General Operations—Public Members

GetCount	Retrieves the number of items in the list box of a combo box.
GetCurSel	Retrieves the index of the currently selected item, if any, in the list box of a combo box.
SetCurSel	Selects a string in the list box of a combo box.
GetEditSel	Gets the starting and ending character positions of the current selection in the edit control of a combo box.
SetEditSel	Selects characters in the edit control of a combo box.
SetItemData	Sets the 32-bit value associated with the specified item in a combo box.
SetItemDataPtr	Sets the 32-bit value associated with the specified item in a combo box to the specified pointer (**void***).
GetItemData	Retrieves the application-supplied 32-bit value associated with the specified combo-box item.

GetItemDataPtr	Retrieves the application-supplied 32-bit value associated with the specified combo-box item as a pointer (**void***).
Clear	Deletes (clears) the current selection (if any) in the edit control.
Copy	Copies the current selection (if any) onto the Clipboard in **CF_TEXT** format.
Cut	Deletes (cuts) the current selection, if any, in the edit control and copies the deleted text onto the Clipboard in **CF_TEXT** format.
Paste	Inserts the data from the Clipboard into the edit control at the current cursor position. Data is inserted only if the Clipboard contains data in **CF_TEXT** format.
LimitText	Limits the length of the text that the user may enter into the edit control of a combo box.
SetItemHeight	Sets the height of list items in a combo box or the height of the edit-control (or static-text) portion of a combo box.
GetItemHeight	Retrieves the height of list items in a combo box.
GetLBText	Gets a string from the list box of a combo box.
GetLBTextLen	Gets the length of a string in the list box of a combo box.
ShowDropDown	Shows or hides the list box of a combo box that has the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.
GetDroppedControlRect	Retrieves the screen coordinates of the visible (dropped-down) list box of a drop-down combo box.
GetDroppedState	Determines whether the list box of a drop-down combo box is visible (dropped down).
SetExtendedUI	Selects either the default user interface or the extended user interface for a combo box that has the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.
GetExtendedUI	Determines whether a combo box has the default user interface or the extended user interface.

String Operations—Public Members

AddString	Adds a string to the end of the list in the list box of a combo box or at the sorted position for list boxes with the **CBS_SORT** style.
DeleteString	Deletes a string from the list box of a combo box.
InsertString	Inserts a string into the list box of a combo box.
ResetContent	Removes all items from the list box and edit control of a combo box.
Dir	Adds a list of filenames to the list box of a combo box.
FindString	Finds the first string that contains the specified prefix in the list box of a combo box.
FindStringExact	Finds the first list-box string (in a combo box) that matches the specified string.
SelectString	Searches for a string in the list box of a combo box and, if the string is found, selects the string in the list box and copies the string to the edit control.

Overridables—Public Members

DrawItem	Called by the framework when a visual aspect of an owner-draw combo box changes.
MeasureItem	Called by the framework to determine combo box dimensions when an owner-draw combo box is created.
CompareItem	Called by the framework to determine the relative position of a new list item in a sorted owner-draw combo box.
DeleteItem	Called by the framework when a list item is deleted from an owner-draw combo box.

Member Functions

CComboBox::AddString

int AddString(LPCSTR *lpszString* **);**

lpszString Points to the null-terminated string that is to be added.

Remarks Adds a string to the list box of a combo box. If the list box was not created with the **CBS_SORT** style, the string is added to the end of the list. Otherwise, the string is inserted into the list, and the list is sorted. To insert a string into a specific location within the list, use the **InsertString** member function.

Return Value If the return value is greater than or equal to 0, it is the zero-based index to the string in the list box. The return value is **CB_ERR** if an error occurs; the return value is **CB_ERRSPACE** if insufficient space is available to store the new string.

See Also **CComboBox::InsertString**, **CComboBox::DeleteString**, **CB_ADDSTRING**

CComboBox::CComboBox

CComboBox( );

Remarks Constructs a **CComboBox** object.

See Also **CComboBox::Create**

CComboBox::Clear

void Clear( );

Remarks Deletes (clears) the current selection, if any, in the edit control of the combo box. To delete the current selection and place the deleted contents onto the Clipboard, use the **Cut** member function.

See Also **CComboBox::Copy**, **CComboBox::Cut**, **CComboBox::Paste**, **WM_CLEAR**

CComboBox::CompareItem

virtual int CompareItem(LPCOMPAREITEMSTRUCT *lpCompareItemStruct* **);**

lpCompareItemStruct A long pointer to a **COMPAREITEMSTRUCT** structure.

Remarks Called by the framework to determine the relative position of a new item in the list-box portion of a sorted owner-draw combo box. By default, this member function does nothing. If you create an owner-draw combo box with the **LBS_SORT** style, you must override this member function to assist the framework in sorting new items added to the list box.

Return Value Indicates the relative position of the two items described in the **COMPAREITEMSTRUCT** structure. It may be any of the following values:

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

See **CWnd::OnCompareItem** on page 956 for a description of **COMPAREITEMSTRUCT**.

See Also **WM_COMPAREITEM**, **CComboBox::DrawItem**, **CComboBox::MeasureItem**, **CComboBox::DeleteItem**

CComboBox::Copy

void Copy( );

Remarks Copies the current selection, if any, in the edit control of the combo box onto the Clipboard in **CF_TEXT** format.

See Also **CComboBox::Clear**, **CComboBox::Cut**, **CComboBox::Paste**, **WM_COPY**

CComboBox::Create

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

dwStyle Specifies the style of the combo box.

rect Points to the position and size of the combo box. Can be a **RECT** structure or a **CRect** object.

pParentWnd Specifies the combo box's parent window (usually a **CDialog**). It must not be **NULL**.

nID Specifies the combo box's control ID.

Remarks

You construct a **CComboBox** object in two steps. First call the constructor, then call **Create**, which creates the Windows combo box and attaches it to the **CComboBox** object. When **Create** executes, Windows sends the **WM_NCCREATE**, **WM_CREATE**, **WM_NCCALCSIZE**, and **WM_GETMINMAXINFO** messages to the combo box. These messages are handled by default by the **OnNcCreate**, **OnCreate**, **OnNcCalcSize**, and **OnGetMinMaxInfo** member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CComboBox**, add a message map to the new class, and override the preceding message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for a new class.

Apply the following window styles to a combo-box control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_VSCROLL** To add vertical scrolling for the list box in the combo box
- **WS_HSCROLL** To add horizontal scrolling for the list box in the combo box
- **WS_GROUP** To group controls
- **WS_TABSTOP** To include the combo box in the tabbing order

See **Create** in the **CWnd** base class for a full description of these window styles.

Return Value

Nonzero if successful; otherwise 0.

Combo-Box Styles You can use any combination of the following combo-box styles for *dwStyle*:

- **CBS_AUTOHSCROLL** Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.
- **CBS_DROPDOWN** Similar to **CBS_SIMPLE**, except that the list box is not displayed unless the user selects an icon next to the edit control.
- **CBS_DROPDOWNLIST** Similar to **CBS_DROPDOWN**, except that the edit control is replaced by a static-text item that displays the current selection in the list box.
- **CBS_HASSTRINGS** An owner-draw combo box contains items consisting of strings. The combo box maintains the memory and pointers for the strings so the application can use the **GetText** member function to retrieve the text for a particular item.
- **CBS_OEMCONVERT** Text entered in the combo-box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** Windows function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the **CBS_SIMPLE** or **CBS_DROPDOWN** styles.
- **CBS_OWNERDRAWFIXED** The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
- **CBS_OWNERDRAWVARIABLE** The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
- **CBS_SIMPLE** The list box is displayed at all times. The current selection in the list box is displayed in the edit control.
- **CBS_SORT** Automatically sorts strings entered into the list box.

Windows 3.1 Only

- **CBS_DISABLENOSCROLL** The list box shows a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
- **CBS_NOINTEGRALHEIGHT** Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, Windows sizes a combo box so that the combo box does not display partial items. ♦

See Also **CComboBox::CComboBox**

CComboBox::Cut

void Cut( );

Remarks

Deletes (cuts) the current selection, if any, in the combo-box edit control and copies the deleted text onto the Clipboard in **CF_TEXT** format.

To delete the current selection without placing the deleted text onto the Clipboard, call the **Clear** member function.

See Also

CComboBox::Clear, **CComboBox::Copy**, **CComboBox::Paste**, **WM_CUT**

CComboBox::DeleteItem

virtual void DeleteItem(LPDELETEITEMSTRUCT *lpDeleteItemStruct* **);**

lpDeleteItemStruct A long pointer to a Windows **DELETEITEMSTRUCT** structure that contains information about the deleted item.

See **CWnd::OnDeleteItem** on page 961 for a description of this structure.

Remarks

Called by the framework when the user deletes an item from an owner-draw **CComboBox** object or destroys the combo box. The default implementation of this function does nothing. Override this function to redraw the combo box as needed.

See Also

CComboBox::CompareItem, **CComboBox::DrawItem**, **CComboBox::MeasureItem**, **WM_DELETEITEM**

CComboBox::DeleteString

int DeleteString(UINT *nIndex* **);**

nIndex Specifies the index to the string that is to be deleted.

Remarks

Deletes a string in the list box of a combo box.

Return Value

If the return value is greater than or equal to 0, then it is a count of the strings remaining in the list. The return value is **CB_ERR** if *nIndex* specifies an index greater then the number of items in the list.

See Also

CComboBox::InsertString, **CComboBox::AddString**, **CB_DELETESTRING**

CComboBox::Dir

int Dir(UINT *attr*, **LPCSTR** *lpszWildCard* **);**

attr Can be any combination of the **enum** values described in **CFile::GetStatus** or any combination of the following values:

- **DDL_READWRITE** File can be read from or written to.
- **DDL_READONLY** File can be read from but not written to.
- **DDL_HIDDEN** File is hidden and does not appear in a directory listing.
- **DDL_SYSTEM** File is a system file.
- **DDL_DIRECTORY** The name specified by *lpszWildCard* specifies a directory.
- **DDL_ARCHIVE** File has been archived.
- **DDL_DRIVES** Include all drives that match the name specified by *lpszWildCard*.
- **DDL_EXCLUSIVE** Exclusive flag. If the exclusive flag is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to "normal" files.

lpszWildCard Points to a file-specification string. The string can contain wildcards (for example, *.*).

Remarks Adds a list of filenames and/or drives to the list box of a combo box.

Return Value If the return value is greater than or equal to 0, it is the zero-based index of the last filename added to the list. The return value is **CB_ERR** if an error occurs; the return value is **CB_ERRSPACE** if insufficient space is available to store the new strings.

See Also **CWnd::DlgDirList**, **CB_DIR**, **CFile::GetStatus**

CComboBox::DrawItem

virtual void DrawItem(LPDRAWITEMSTRUCT *lpDrawItemStruct* **);**

lpDrawItemStruct A pointer to a **DRAWITEMSTRUCT** structure that contains information about the type of drawing required.

Remarks

Called by the framework when a visual aspect of an owner-draw combo box changes. The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed.

See **CWnd::OnDrawItem** on page 964 for a description of this structure.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CComboBox** object. Before this member function terminates, the application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct*.

See Also

CComboBox::CompareItem, **::DrawItem**, **CComboBox::MeasureItem**, **CComboBox::DeleteItem**

CComboBox::FindString

int FindString(int *nStartAfter*, **LPCSTR** *lpszString* **) const;**

nStartAfter Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If –1, the entire list box is searched from the beginning.

lpszString Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string may contain any combination of uppercase and lowercase letters.

Remarks

Finds, but doesn't select, the first string that contains the specified prefix in the list box of a combo box.

Return Value

If the return value is greater than or equal to 0, it is the zero-based index of the matching item. It is **CB_ERR** if the search was unsuccessful.

See Also

CComboBox::SelectString, **CComboBox::SetCurSel**, **CB_FINDSTRING**

CComboBox::FindStringExact

Windows 3.1 Only

int FindStringExact(int *nIndexStart*, **LPCSTR** *lpszFind* **) const;** ♦

nIndexStart Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nIndexStart*. If *nIndexStart* is –1, the entire list box is searched from the beginning.

lpszFind Points to the null-terminated string to search for. This string can contain a complete filename, including the extension. The search is not case sensitive, so this string can contain any combination of uppercase and lowercase letters.

Remarks

Call the **FindStringExact** member function to find the first list-box string (in a combo box) that matches the string specified in *lpszFind*.

If the combo box was created with an owner-draw style but without the **CBS_HASSTRINGS** style, **FindStringExact** attempts to match the doubleword value against the value of *lpszFind*.

Return Value

The zero-based index of the matching item, or **CB_ERR** if the search was unsuccessful.

See Also

CComboBox::FindString, CB_FINDSTRINGEXACT

CComboBox::GetCount

int GetCount( ) const;

Return Value

The number of items in the list box of a combo box. The returned count is one greater then the index value of the last item (the index is zero-based). It is **CB_ERR** if an error occurs.

See Also

CB_GETCOUNT

CComboBox::GetCurSel

int GetCurSel( ) const;

Return Value

The zero-based index of the currently selected item in the list box of a combo box, or **CB_ERR** if no item is selected.

See Also

CComboBox::SetCurSel, CB_GETCURSEL

CComboBox::GetDroppedControlRect

Windows 3.1 Only **void GetDroppedControlRect(LPRECT** *lprect* **) const;** ♦

lprect Points to the **RECT** structure that is to receive the coordinates.

Remarks Call the **GetDroppedControlRect** member function to retrieve the screen coordinates of the visible (dropped-down) list box of a drop-down combo box.

See Also **CB_GETDROPPEDCONTROLRECT**

CComboBox::GetDroppedState

Windows 3.1 Only **BOOL GetDroppedState( ) const;** ♦

Remarks Call the **GetDroppedState** member function to determine whether the list box of a drop-down combo box is visible (dropped down).

Return Value Nonzero if the listbox is visible; otherwise 0.

See Also **CB_SHOWDROPDOWN**, **CB_GETDROPPEDSTATE**

CComboBox::GetEditSel

DWORD GetEditSel( ) const;

Remarks Gets the starting and ending character positions of the current selection in the edit control of a combo box.

Return Value A 32-bit value that contains the starting position in the low-order word and the position of the first nonselected character after the end of the selection in the high-order word. If this function is used on a combo box without an edit control, **CB_ERR** is returned.

See Also **CComboBox::SetEditSel**, **CB_GETEDITSEL**

CComboBox::GetExtendedUI

Windows 3.1 Only **BOOL GetExtendedUI( ) const;** ♦

Remarks Call the **GetExtendedUI** member function to determine whether a combo box has the default user interface or the extended user interface. The extended user interface can be identified in the following ways:

- Clicking the static control displays the list box only for combo boxes with the **CBS_DROPDOWNLIST** style.
- Pressing the DOWN ARROW key displays the list box (F4 is disabled).
- Scrolling in the static control is disabled when the item list is not visible (arrow keys are disabled).

Return Value Nonzero if the combo box has the extended user interface; otherwise 0.

See Also **CComboBox::SetExtendedUI**, **CB_GETEXTENDEDUI**

CComboBox::GetItemData

DWORD GetItemData(int *nIndex* **) const;**

nIndex Contains the zero-based index of an item in the combo box's list box.

Remarks Retrieves the application-supplied 32-bit value associated with the specified combo-box item. The 32-bit value can be set with the *dwItemData* parameter of a **SetItemData** member function call. Use the **GetItemDataPtr** member function if the 32-bit value to be retrieved is a pointer (**void***).

Return Value The 32-bit value associated with the item, or **CB_ERR** if an error occurs.

See Also **CComboBox::SetItemData**, **CComboBox::GetItemDataPtr**, **CComboBox::SetItemDataPtr**, **CB_GETITEMDATA**

CComboBox::GetItemDataPtr

void* GetItemDataPtr(int *nIndex* **) const;**

nIndex Contains the zero-based index of an item in the combo box's list box.

Remarks

Retrieves the application-supplied 32-bit value associated with the specified combo-box item as a pointer (**void***).

Return Value

Retrieves a pointer, or –1 if an error occurs.

See Also

CComboBox::SetItemDataPtr, **CComboBox::GetItemData**, **CComboBox::SetItemData**, **CB_GETITEMDATA**

CComboBox::GetItemHeight

Windows 3.1 Only

int GetItemHeight(int *nIndex* **) const;** ♦

nIndex Specifies the component of the combo box whose height is to be retrieved. If the *nIndex* parameter is –1, the height of the edit-control (or static-text) portion of the combo box is retrieved. If the combo box has the **CBS_OWNERDRAWVARIABLE** style, *nIndex* specifies the zero-based index of the list item whose height is to be retrieved. Otherwise, *nIndex* should be set to 0.

Remarks

Call the **GetItemHeight** member function to retrieve the height of list items in a combo box.

Return Value

The height, in pixels, of the specified item in a combo box. The return value is **CB_ERR** if an error occurs.

See Also

CComboBox::SetItemHeight, **WM_MEASUREITEM**, **CB_GETITEMHEIGHT**

CComboBox::GetLBText

int GetLBText(int *nIndex*, **LPSTR** *lpszText* **) const;**

void GetLBText(int *nIndex*, **CString&** *rString* **) const;**

nIndex Contains the zero-based index of the list-box string to be copied.

lpszText Points to a buffer that is to receive the string. The buffer must have sufficient space for the string and a terminating null character.

rString A reference to a **CString**.

Remarks

Gets a string from the list box of a combo box. The second form of this member function fills a **CString** object with the item's text.

Return Value

The length (in bytes) of the string, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **CB_ERR**.

See Also

CComboBox::GetLBTextLen, **CB_GETLBTEXT**

CComboBox::GetLBTextLen

int GetLBTextLen(int *nIndex* **) const;**

nIndex Contains the zero-based index of the list-box string.

Remarks

Gets the length of a string in the list box of a combo box.

Return Value

The length of the string in bytes, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **CB_ERR**.

See Also

CComboBox::GetLBText, **CB_GETLBTEXTLEN**

CComboBox::InsertString

int InsertString(int *nIndex*, **LPCSTR** *lpszString* **);**

nIndex Contains the zero-based index to the position in the list box that will receive the string. If this parameter is –1, the string is added to the end of the list.

lpszString Points to the null-terminated string that is to be inserted.

Remarks

Inserts a string into the list box of a combo box. Unlike the **AddString** member function, the **InsertString** member function does not cause a list with the **CBS_SORT** style to be sorted.

Return Value

The zero-based index of the position at which the string was inserted. The return value is **CB_ERR** if an error occurs. The return value is **CB_ERRSPACE** if insufficient space is available to store the new string.

See Also

CComboBox::AddString, **CComboBox::DeleteString**, **CComboBox::ResetContent**, **CB_INSERTSTRING**

CComboBox::LimitText

BOOL LimitText(int *nMaxChars* **);**

nMaxChars Specifies the length (in bytes) of the text that the user can enter. If this parameter is 0, the text length is set to 65,535 bytes.

Remarks

Limits the length in bytes of the text that the user can enter into the edit control of a combo box. If the combo box does not have the style **CBS_AUTOHSCROLL**, setting the text limit to be larger than the size of the edit control will have no effect. **LimitText** only limits the text the user can enter. It has no effect on any text already in the edit control when the message is sent, nor does it affect the length of the text copied to the edit control when a string in the list box is selected.

Return Value

Nonzero if successful. If called for a combo box with the style **CBS_DROPDOWNLIST** or for a combo box without an edit control, the return value is **CB_ERR**.

See Also

CB_LIMITTEXT

CComboBox::MeasureItem

virtual void MeasureItem(LPMEASUREITEMSTRUCT *lpMeasureItemStruct* **);**

lpMeasureItemStruct A long pointer to a **MEASUREITEMSTRUCT** structure.

Remarks

Called by the framework when a combo box with an owner-draw style is created.

By default, this member function does nothing. Override this member function and fill in the **MEASUREITEM** structure to inform Windows of the dimensions of the list box in the combo box. If the combo box is created with the **CBS_OWNERDRAWVARIABLE** style, the framework calls this member function for each item in the list box. Otherwise, this member is called only once.

Using the **CBS_OWNERDRAWFIXED** style in an owner-draw combo box created with the **SubclassDlgItem** member function of **CWnd** involves further programming considerations. See the discussion in Technical Note 14 in MSVC\HELP\MFCNOTES.HLP.

See **CWnd::OnMeasureItem** on page 980 for a description of the **MEASUREITEMSTRUCT** structure.

See Also

CComboBox::CompareItem, **CComboBox::DrawItem**, **::MeasureItem**, **CComboBox::DeleteItem**

CComboBox::Paste

void Paste( );

Remarks

Inserts the data from the Clipboard into the edit control of the combo box at the current cursor position. Data is inserted only if the Clipboard contains data in **CF_TEXT** format.

See Also

CComboBox::Clear, **CComboBox::Copy**, **CComboBox::Cut**, **WM_PASTE**

CComboBox::ResetContent

void ResetContent( );

Remarks

Removes all items from the list box and edit control of a combo box.

See Also

CB_RESETCONTENT

CComboBox::SelectString

int SelectString(int *nStartAfter*, **LPCSTR** *lpszString* **);**

nStartAfter Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If –1, the entire list box is searched from the beginning.

lpszString Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string may contain any combination of uppercase and lowercase letters.

Remarks

Searches for a string in the list box of a combo box, and if the string is found, selects the string in the list box and copies it to the edit control. A string is selected only if its initial characters (from the starting point) match the characters in the prefix string. Note that the **SelectString** and **FindString** member functions both find a string, but the **SelectString** member function also selects the string.

Return Value

The zero-based index of the selected item if the string was found. If the search was unsuccessful, the return value is **CB_ERR** and the current selection is not changed.

See Also

CComboBox::FindString, CB_SELECTSTRING

CComboBox::SetCurSel

int SetCurSel(int *nSelect* **);**

nSelect Specifies the zero-based index of the string to select. If –1, any current selection in the list box is removed and the edit control is cleared.

Remarks Selects a string in the list box of a combo box. If necessary, the list box scrolls the string into view (if the list box is visible). The text in the edit control of the combo box is changed to reflect the new selection. Any previous selection in the list box is removed.

Return Value The zero-based index of the item selected if the message is successful. The return value is **CB_ERR** if *nSelect* is greater than the number of items in the list or if *nSelect* is set to –1, which clears the selection.

See Also **CComboBox::GetCurSel**, **CB_SETCURSEL**

CComboBox::SetEditSel

BOOL SetEditSel(int *nStartChar*, **int** *nEndChar* **);**

nStartChar Specifies the starting position. If the starting position is set to –1, then any existing selection is removed.

nEndChar Specifies the ending position. If the ending position is set to –1, then all text from the starting position to the last character in the edit control is selected.

Remarks Selects characters in the edit control of a combo box. The positions are zero-based. To select the first character of the edit control, you specify a starting position of 0. The ending position is for the character just after the last character to select. For example, to select the first four characters of the edit control, you would use a starting position of 0 and an ending position of 4.

Return Value Nonzero if the member function is successful; otherwise 0. It is **CB_ERR** if **CComboBox** has the **CBS_DROPDOWNLIST** style or doesn't have a list box.

See Also **CComboBox::GetEditSel**, **CB_SETEDITSEL**

CComboBox::SetExtendedUI

Windows 3.1 Only **int SetExtendedUI(BOOL** *bExtended* = **TRUE);** ♦

bExtended Specifies whether the combo box should use the extended user interface or the default user interface. A value of **TRUE** selects the extended user interface; a value of **FALSE** selects the standard user interface.

Remarks

Call the **SetExtendedUI** member function to select either the default user interface or the extended user interface for a combo box that has the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style. The extended user interface can be identified in the following ways:

- Clicking the static control displays the list box only for combo boxes with the **CBS_DROPDOWNLIST** style.
- Pressing the DOWN ARROW key displays the list box (F4 is disabled).
- Scrolling in the static control is disabled when the item list is not visible (the arrow keys are disabled).

Return Value

CB_OKAY if the operation is successful, or **CB_ERR** if an error occurs.

See Also

CComboBox::GetExtendedUI, **CB_SETEXTENDEDUI**

CComboBox::SetItemData

int SetItemData(int *nIndex*, **DWORD** *dwItemData* **);**

nIndex Contains a zero-based index to the item to set.

dwItemData Contains the new value to associate with the item.

Remarks

Sets the 32-bit value associated with the specified item in a combo box. Use the **SetItemDataPtr** member function if the 32-bit item is to be a pointer.

Return Value

CB_ERR if an error occurs.

See Also

CComboBox::GetItemData, **CComboBox::GetItemDataPtr**, **CComboBox::SetItemDataPtr**, **CB_SETITEMDATA**, **CComboBox::AddString**, **CComboBox::InsertString**

CComboBox::SetItemDataPtr

int SetItemDataPtr(int *nIndex*, **void*** *pData* **);**

nIndex Contains a zero-based index to the item.

pData Contains the pointer to associate with the item.

Remarks

Sets the 32-bit value associated with the specified item in a combo box to be the specified pointer (**void***).

Return Value **CB_ERR** if an error occurs.

See Also **CComboBox::GetItemData**, **CComboBox::GetItemDataPtr**, **CComboBox::SetItemData**, **CB_SETITEMDATA**, **CComboBox::AddString**, **CComboBox::InsertString**

CComboBox::SetItemHeight

Windows 3.1 Only **int SetItemHeight(int** *nIndex*, **UINT** *cyItemHeight* **);** ♦

nIndex Specifies whether the height of list items or the height of the edit-control (or static-text) portion of the combo box is set.

If the combo box has the **CBS_OWNERDRAWVARIABLE** style, *nIndex* specifies the zero-based index of the list item whose height is to be set; otherwise, *nIndex* must be 0 and the height of all list items will be set.

If *nIndex* is –1, the height of the edit-control or static-text portion of the combo box is to be set.

cyItemHeight Specifies the height, in pixels, of the combo-box component identified by *nIndex*.

Remarks Call the **SetItemHeight** member function to set the height of list items in a combo box or the height of the edit-control (or static-text) portion of a combo box. The height of the edit-control (or static-text) portion of the combo box is set independently of the height of the list items. An application must ensure that the height of the edit-control (or static-text) portion isn't smaller than the height of a particular list-box item.

Return Value **CB_ERR** if the index or height is invalid; otherwise 0.

See Also **CComboBox::GetItemHeight**, **WM_MEASUREITEM**, **CB_SETITEMHEIGHT**

CComboBox::ShowDropDown

void ShowDropDown(BOOL *bShowIt* **= TRUE);**

bShowIt Specifies whether the drop-down list box is to be shown or hidden. A value of **TRUE** shows the list box. A value of **FALSE** hides the list box.

Remarks

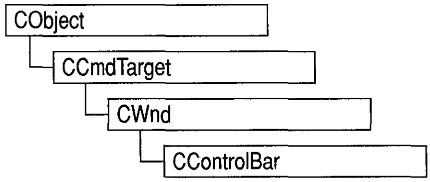
Shows or hides the list box of a combo box that has the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style. By default, a combo box of this style will show the list box.

This member function has no effect on a combo box created with the **CBS_SIMPLE** style.

See Also

CB_SHOWDROPDOWN

class CControlBar : public CWnd



CControlBar is the base class for the control-bar classes **CStatusBar**, **CToolBar**, and **CDialogBar**. A control bar is a window that is usually aligned to the top or bottom of a frame window. It may contain child items that are either **HWND**-based controls, which are Windows windows that generate and respond to Windows messages, or non-**HWND**-based items, which are not windows and are managed by application code or framework code. List boxes and edit controls are examples of **HWND**-based controls; status-bar panes and bitmap buttons are examples of non-**HWND**-based controls.

Control-bar windows are usually child windows of a parent frame window and are usually "siblings" to the client view or MDI client of the frame window. A **CControlBar** object uses information about the parent window's client rectangle to position itself. It then informs the parent window as to how much space remains unallocated in the parent window's client area.

#include <afxext.h>

See Also **CStatusBar**, **CToolBar**, **CDialogBar**

Data Members—Public Members

m_bAutoDelete	If nonzero, the **CControlBar** object is deleted when the Windows control bar is destroyed.

Attributes—Public Members

GetCount	Returns the number of non-**HWND** elements in the control bar.

Member Functions

CControlBar::GetCount

int GetCount( );

Remarks Returns the number of non-**HWND** items on the **CControlBar** object. The type of the item depends on the derived object: panes for **CStatusBar** objects, and buttons and separators for **CToolBar** objects.

Return Value The number of non-**HWND** items on the **CControlBar** object. This function returns 0 for a **CDialogBar** object.

See Also **CToolBar::SetButtons**, **CStatusBar::SetIndicators**

Data Members

CControlBar::m_bAutoDelete

Remarks **m_bAutoDelete** is a public variable of type **BOOL**. If it is nonzero when the Windows control-bar object is destroyed, the **CControlBar** object is deleted.

A control-bar object is usually embedded in a frame-window object. In this case, **m_bAutoDelete** is 0 because the embedded control-bar object is destroyed when the frame window is destroyed.

Set this variable to a nonzero value if you allocate a **CControlBar** object on the heap and you do not plan to call **delete**.

See Also **CWnd::DestroyWindow**

struct CCreateContext

The framework uses the **CCreateContext** structure when it creates the frame windows and views associated with a document. When creating a window, the values in this structure provide information used to connect the components that make up a document and the view of its data. You will only need to use **CCreateContext** if you are overriding parts of the creation process.

A **CCreateContext** structure contains pointers to the document, the frame window, the view, and the document template. It also contains a pointer to a **CRuntimeClass** that identifies the type of view to create. The run-time class information and the current document pointer are used to create a new view dynamically. The following table suggests how and when each **CCreateContext** member might be used:

Member	What It Is For
m_pNewViewClass	**CRuntimeClass** of the new view to create.
m_pCurrentDoc	The existing document to be associated with the new view.
m_pNewDocTemplate	The document template associated with the creation of a new MDI frame window.
m_pLastView	The original view upon which additional views are modeled, as in the creation of a splitter window's views or the creation of a second view on a document.
m_pCurrentFrame	The frame window upon which additional frame windows are modeled, as in the creation of a second frame window on a document.

When a document template creates a document and its associated components, it validates the information stored in the **CCreateContext** structure. For example, a view should not be created for a nonexistent document.

Note All of the pointers in **CCreateContext** are optional and may be **NULL** if unspecified or unknown.

CCreateContext is used by the member functions listed under "See Also." Consult the descriptions of these functions for specific information if you plan to override them.

Here are a few general guidelines:

- When passed as an argument for window creation, as in **CWnd::Create**, **CFrameWnd::Create**, and **CFrameWnd::LoadFrame**, the create context specifies what the new window should be connected to. For most windows, the entire structure is optional and a **NULL** pointer may be passed.
- For overridable member functions, such as **CFrameWnd::OnCreateClient**, the **CCreateContext** argument is optional.
- For member functions involved in view creation, you must provide enough information to create the view. For example, for the first view in a splitter window, you must supply the view class information and the current document.

In general, if you use the framework defaults, you can ignore **CCreateContext**. If you attempt more advanced modifications, refer to the Microsoft Foundation Class Library source code or the sample programs, such as the VIEWEX example in the MFC\SAMPLES\VIEWEX subdirectory. If you do forget a required parameter, a framework assertion will tell you what you forgot.

#include <afxext.h>

See Also **CFrameWnd::Create**, **CFrameWnd::LoadFrame**, **CFrameWnd::OnCreateClient**, **CSplitterWnd::Create**, **CSplitterWnd::CreateView**, **CWnd::Create**

class CDataExchange

The **CDataExchange** class supports the dialog data exchange (DDX) and dialog data validation (DDV) routines used by the Microsoft Foundation classes. Use this class if you are writing data exchange routines for custom data types or controls, or if you are writing your own data validation routines. For more information on writing your own DDX and DDV routines, see Technical Note 26 in MSVC\HELP\MFCNOTES.HLP. For an overview of DDX and DDV, see the *App Studio User's Guide.*

A **CDataExchange** object provides the context information needed for DDX and DDV to take place. The flag **m_bSaveAndValidate** is **FALSE** when DDX is used to fill the initial values of dialog controls from data members. The flag **m_bSaveAndValidate** is **TRUE** when DDX is used to set the current values of dialog controls into data members and when DDV is used to validate the data values. If the DDV validation fails, the DDV procedure will display a message box explaining the input error. The DDV procedure will then call **Fail** to reset the focus to the offending control and throw an exception to stop the validation process.

See Also **CWnd::DoDataExchange**, **CWnd::UpdateData**

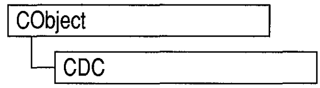
Data Members

m_bSaveAndValidate	Flag for the direction of DDX and DDV.
m_pDlgWnd	The dialog box or window where the data exchange takes place.

Operations—Public Members

PrepareCtrl	Prepares the specified control for data exchange or validation. Use for nonedit controls.
PrepareEditCtrl	Prepares the specified edit control for data exchange or validation.
Fail	Called when validation fails. Resets focus to the previous control and throws an exception.
PrepareVBCtrl	Prepares a Visual Basic control for data exchange or validation.

class CDC : public CObject



The **CDC** class defines a class of device-context objects. The **CDC** object provides member functions for working with a device context, such as a display or printer, as well as members for working with a display context associated with the client area of a window.

Do all drawing through the member functions of a **CDC** object. The class provides member functions for device-context operations, working with drawing tools, type-safe graphics device interface (GDI) object selection, and working with colors and palettes. It also provides member functions for getting and setting drawing attributes, mapping, working with the viewport, working with the window extent, converting coordinates, working with regions, clipping, drawing lines, and drawing simple shapes, ellipses, and polygons. Member functions are also provided for drawing text, working with fonts, using printer escapes, scrolling, and playing metafiles.

To use a **CDC** object, construct it, and then call its member functions, which parallel Windows functions that use device contexts or display contexts.

For specific uses, the Microsoft Foundation Class Library provides several classes derived from **CDC**. **CPaintDC** encapsulates calls to **BeginPaint** and **EndPaint**. **CClientDC** manages a display context associated with a window's client area. **CWindowDC** manages a display context associated with an entire window, including its frame and controls. **CMetaFileDC** associates a device context with a metafile.

CDC contains two device contexts, **m_hDC** and **m_hAttribDC**, which, on creation of a **CDC** object, refer to the same device. **CDC** directs all output GDI calls to **m_hDC** and most attribute GDI calls to **m_hAttribDC**. (An example of an attribute call is **GetTextColor**, while **SetTextColor** is an output call.)

The framework uses these two device contexts to, for example, implement a **CMetaFileDC** object that will send output to a metafile while reading attributes from a physical device. Print preview is implemented in the framework in a similar fashion. You can also use the two device contexts in a similar way in your application-specific code.

There are times when you may need text-metric information from both the **m_hDC** and **m_hAttribDC** device contexts. The following pairs of functions provide this capability:

Uses m_hAttribDC	Uses m_hDC
GetTextExtent	**GetOutputTextExtent**
GetTabbedTextExtent	**GetOutputTabbedTextExtent**
GetTextMetrics	**GetOutputTextMetrics**
GetCharWidth	**GetOutputCharWidth**

#include <afxwin.h>

See Also **CPaintDC**, **CWindowDC**, **CClientDC**, **CMetaFileDC**

Data Members—Public Members

m_hDC	The output-device context used by this **CDC** object.
m_hAttribDC	The attribute-device context used by this **CDC** object.

Construction/Destruction—Public Members

CDC	Constructs a **CDC** object.

Initialization—Public Members

CreateDC	Creates a device context for a specific device.
CreateIC	Creates an information context for a specific device. This provides a fast way to get information about the device without creating a device context.
CreateCompatibleDC	Creates a memory-device context that is compatible with another device context. You can use it to prepare images in memory.
DeleteDC	Deletes the Windows device context associated with this **CDC** object.
FromHandle	Returns a pointer to a **CDC** object when given a handle to a device context. If a **CDC** object is not attached to the handle, a temporary **CDC** object is created and attached.

DeleteTempMap	Called by the **CWinApp** idle-time handler to delete any temporary **CDC** object created by **FromHandle**. Also detaches the device context.
Attach	Attaches a Windows device context to this **CDC** object.
Detach	Detaches the Windows device context from this **CDC** object.
SetAttribDC	Sets **m_hAttribDC**, the attribute device context.
SetOutputDC	Sets **m_hDC**, the output device context.
ReleaseAttribDC	Releases **m_hAttribDC**, the attribute device context.
ReleaseOutputDC	Releases **m_hDC**, the output device context.

Device-Context Functions—Public Members

GetSafeHdc	Returns **m_hDC**, the output device context.
SaveDC	Saves the current state of the device context.
RestoreDC	Restores the device context to a previous state saved with **SaveDC**.
ResetDC	Updates the **m_hAttribDC** device context.
GetDeviceCaps	Retrieves a specified kind of device-specific information about a given display device's capabilities.
IsPrinting	Determines if the device context is being used for printing.

Drawing-Tool Functions—Public Members

GetBrushOrg	Retrieves the origin of the current brush.
SetBrushOrg	Specifies the origin for the next brush selected into a device context.
EnumObjects	Enumerates the pens and brushes available in a device context.

Type-Safe Selection Helpers—Public Members

SelectObject	Selects a GDI drawing object such as a pen.
SelectStockObject	Selects one of the predefined stock pens, brushes, or fonts provided by Windows.

Color and Color Palette Functions—Public Members

GetNearestColor	Retrieves the closest logical color to a specified logical color that the given device can represent.
SelectPalette	Selects the logical palette.
RealizePalette	Maps palette entries in the current logical palette to the system palette.
UpdateColors	Updates the client area of the device context by matching the current colors in the client area to the system palette on a pixel-by-pixel basis.

Drawing-Attribute Functions—Public Members

GetBkColor	Retrieves the current background color.
SetBkColor	Sets the current background color.
GetBkMode	Retrieves the background mode.
SetBkMode	Sets the background mode.
GetPolyFillMode	Retrieves the current polygon-filling mode.
SetPolyFillMode	Sets the polygon-filling mode.
GetROP2	Retrieves the current drawing mode.
SetROP2	Sets the current drawing mode.
GetStretchBltMode	Retrieves the current bitmap-stretching mode.
SetStretchBltMode	Sets the bitmap-stretching mode.
GetTextColor	Retrieves the current text color.
SetTextColor	Sets the text color.

Mapping Functions—Public Members

GetMapMode	Retrieves the current mapping mode.
SetMapMode	Sets the current mapping mode.
GetViewportOrg	Retrieves the x- and y-coordinates of the viewport origin.
SetViewportOrg	Sets the viewport origin.
OffsetViewportOrg	Modifies the viewport origin relative to the coordinates of the current viewport origin.
GetViewportExt	Retrieves the x- and y-extents of the viewport.
SetViewportExt	Sets the x- and y-extents of the viewport.
ScaleViewportExt	Modifies the viewport extent relative to the current values.

GetWindowOrg	Retrieves the x- and y-coordinates of the origin of the associated window.
SetWindowOrg	Sets the window origin of the device context.
OffsetWindowOrg	Modifies the window origin relative to the coordinates of the current window origin.
GetWindowExt	Retrieves the x- and y-extents of the associated window.
SetWindowExt	Sets the x- and y-extents of the associated window.
ScaleWindowExt	Modifies the window extents relative to the current values.

Coordinate Functions—Public Members

DPtoLP	Converts device points or rectangles into logical points or rectangles.
LPtoDP	Converts logical points or rectangles into device points or rectangles.

Region Functions—Public Members

FillRgn	Fills a specific region with the specified brush.
FrameRgn	Draws a border around a specific region using a brush.
InvertRgn	Inverts the colors in a region.
PaintRgn	Fills a region with the selected brush.

Clipping Functions—Public Members

SetBoundsRect	Controls the accumulation of bounding-rectangle information for the specified device context.
GetBoundsRect	Returns the current accumulated bounding rectangle for the specified device context.
GetClipBox	Retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary.
SelectClipRgn	Selects the given region as the current clipping region.
ExcludeClipRect	Creates a new clipping region that consists of the existing clipping region minus the specified rectangle.

ExcludeUpdateRgn	Prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.
IntersectClipRect	Creates a new clipping region by forming the intersection of the current region and a rectangle.
OffsetClipRgn	Moves the clipping region of the given device.
PtVisible	Specifies whether the given point is within the clipping region.
RectVisible	Determines whether any part of the given rectangle lies within the clipping region.

Line-Output Functions—Public Members

GetCurrentPosition	Retrieves the current position of the pen (in logical coordinates).
MoveTo	Moves the current position.
LineTo	Draws a line from the current position up to, but not including, a point.
Arc	Draws an elliptical arc.
Polyline	Draws a set of line segments connecting the specified points.

Simple Drawing Functions—Public Members

FillRect	Fills a given rectangle by using a specific brush.
FrameRect	Draws a border around a rectangle.
InvertRect	Inverts the contents of a rectangle.
DrawIcon	Draws an icon.

Ellipse and Polygon Functions—Public Members

Chord	Draws a chord (a closed figure bounded by the intersection of an ellipse and a line segment).
DrawFocusRect	Draws a rectangle in the style used to indicate focus.
Ellipse	Draws an ellipse.
Pie	Draws a pie-shaped wedge.
Polygon	Draws a polygon consisting of two or more points (vertices) connected by lines.

PolyPolygon	Creates two or more polygons that are filled using the current polygon-filling mode. The polygons may be disjoint or they may overlap.
Rectangle	Draws a rectangle using the current pen and fills it using the current brush.
RoundRect	Draws a rectangle with rounded corners using the current pen and filled using the current brush.

Bitmap Functions—Public Members

PatBlt	Creates a bit pattern.
BitBlt	Copies a bitmap from a specified device context.
StretchBlt	Moves a bitmap from a source rectangle and device into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle.
GetPixel	Retrieves the RGB color value of the pixel at the specified point.
SetPixel	Sets the pixel at the specified point to the closest approximation of the specified color.
FloodFill	Fills an area with the current brush.
ExtFloodFill	Fills an area with the current brush. Provides more flexibility than the **FloodFill** member function.

Text Functions—Public Members

TextOut	Writes a character string at a specified location using the currently selected font.
ExtTextOut	Writes a character string within a rectangular region using the currently selected font.
TabbedTextOut	Writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions.
DrawText	Draws formatted text in the specified rectangle.
GetTextExtent	Computes the width and height of a line of text on the attribute device context using the current font to determine the dimensions.
GetOutputTextExtent	Computes the width and height of a line of text on the output device context using the current font to determine the dimensions.

GetTabbedTextExtent	Computes the width and height of a character string on the attribute device context.
GetOutputTabbedTextExtent	Computes the width and height of a character string on the output device context.
GrayString	Draws dimmed (grayed) text at the given location.
GetTextAlign	Retrieves the text-alignment flags.
SetTextAlign	Sets the text-alignment flags.
GetTextFace	Copies the typeface name of the current font into a buffer as a null-terminated string.
GetTextMetrics	Retrieves the metrics for the current font from the attribute device context.
GetOutputTextMetrics	Retrieves the metrics for the current font from the output device context.
SetTextJustification	Adds space to the break characters in a string.
GetTextCharacterExtra	Retrieves the current setting for the amount of intercharacter spacing.
SetTextCharacterExtra	Sets the amount of intercharacter spacing.

Font Functions—Public Members

GetFontData	Retrieves font metric information from a scalable font file. The information to retrieve is identified by specifying an offset into the font file and the length of the information to return.
GetKerningPairs	Retrieves the character kerning pairs for the font that is currently selected in the specified device context.
GetOutlineTextMetrics	Retrieves font metric information for TrueType fonts.
GetGlyphOutline	Retrieves the outline curve or bitmap for an outline character in the current font.
GetCharABCWidths	Retrieves the widths of consecutive characters in a specified range from the current TrueType font. The widths are returned in logical units. This function succeeds only with TrueType fonts.

GetCharWidth	Retrieves the widths of individual characters in a consecutive group of characters from the current font using the attribute device context.
GetOutputCharWidth	Retrieves the widths of individual characters in a consecutive group of characters from the current font using the output device context.
SetMapperFlags	Alters the algorithm that the font mapper uses when it maps logical fonts to physical fonts.
GetAspectRatioFilter	Retrieves the setting for the current aspect-ratio filter.

Printer Escape Functions—Public Members

QueryAbort	Calls the **AbortProc** callback function for a printing application and queries whether the printing should be terminated.
Escape	Allows applications to access facilities that are not directly available from a particular device through GDI. Escape calls made by an application are translated and sent to the device driver.
StartDoc	Informs the device driver that a new print job is starting.
StartPage	Informs the device driver that a new page is starting.
EndPage	Informs the device driver that a page is ending.
SetAbortProc	Sets a programmer-supplied callback function that Windows calls if a print job must be aborted.
AbortDoc	Terminates the current print job, erasing everything the application has written to the device since the last call of the **StartDoc** member function.
EndDoc	Ends a print job started by the **StartDoc** member function.

Scrolling Functions—Public Members

ScrollDC	Scrolls a rectangle of bits horizontally and vertically.

Metafile Functions—Public Members

PlayMetaFile	Plays the contents of the specified metafile on the given device. The metafile can be played any number of times.

Member Functions

CDC::AbortDoc

int AbortDoc( );

Remarks

Terminates the current print job and erases everything the application has written to the device since the last call to the **StartDoc** member function. This member function replaces the **ABORTDOC** printer escape.

AbortDoc should be used to terminate:

- Printing operations that do not specify an abort function using **SetAbortProc**.
- Printing operations that have not yet reached their first **NEWFRAME** or **NEXTBAND** escape call.

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using either the **EndDoc** or **AbortDoc** member functions of class **CDC**. GDI automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must call **AbortDoc** before destroying the dialog box.

If Print Manager was used to start the print job, calling **AbortDoc** erases the entire spool job—the printer receives nothing. If Print Manager was not used to start the print job, the data may have been sent to the printer before **AbortDoc** was called. In this case, the printer driver would have reset the printer (when possible) and closed the print job.

When running under Windows version 3.0, this member function sends an **ABORTDOC** printer escape.

Return Value

A value greater than or equal to 0 if successful, or a negative value if an error has occurred. The following list shows common error values and their meanings:

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.
- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User terminated the job through the Print Manager.

See Also

CDC::StartDoc, **CDC::EndDoc**, **CDC::SetAbortProc**

CDC::Arc

BOOL Arc(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3*, **int** *x4*, **int** *y4* **);**

BOOL Arc(LPCRECT *lpRect*, **POINT** *ptStart*, **POINT** *ptEnd* **);**

x1 Specifies the x-coordinate of the upper-left corner of the bounding rectangle (in logical units).

y1 Specifies the y-coordinate of the upper-left corner of the bounding rectangle (in logical units).

x2 Specifies the x-coordinate of the lower-right corner of the bounding rectangle (in logical units).

y2 Specifies the y-coordinate of the lower-right corner of the bounding rectangle (in logical units).

x3 Specifies the x-coordinate of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

y3 Specifies the y-coordinate of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

x4 Specifies the x-coordinate of the point that defines the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

y4 Specifies the y-coordinate of the point that defines the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

lpRect Specifies the bounding rectangle (in logical units). You can pass either an **LPRECT** or a **CRect** object for this parameter.

ptStart Specifies the x- and y-coordinates of the point that defines the arc's starting point (in logical units). This point does not have to lie exactly on the arc. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

ptEnd Specifies the x- and y-coordinates of the point that defines the arc's ending point (in logical units). This point does not have to lie exactly on the arc. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks Draws an elliptical arc. The arc drawn by using the function is a segment of the ellipse defined by the specified bounding rectangle. The actual starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The actual ending point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified ending point intersects the ellipse. The arc is drawn in a counterclockwise direction. Since an arc is not a closed figure, it is not filled. Both the width and height of the rectangle must be greater than 2 units and less than 32,767 units.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::Chord**, **::Arc**, **POINT**, **RECT**

CDC::Attach

BOOL Attach(HDC *hDC* **);**

hDC A Windows device context.

Remarks Use this member function to attach an *hDC* to the **CDC** object. The *hDC* is stored in both **m_hDC**, the output device context, and in **m_hAttribDC**, the attribute device context.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::Detach**, **CDC::m_hDC**, **CDC::m_hAttribDC**

CDC::BitBlt

BOOL BitBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **CDC*** *pSrcDC*, **int** *xSrc*, **int** *ySrc*, **DWORD** *dwRop* **);**

x Specifies the logical x-coordinate of the upper-left corner of the destination rectangle.

y Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.

nWidth Specifies the width (in logical units) of the destination rectangle and source bitmap.

nHeight Specifies the height (in logical units) of the destination rectangle and source bitmap.

pSrcDC Pointer to a **CDC** object that identifies the device context from which the bitmap will be copied. It must be **NULL** if *dwRop* specifies a raster operation that does not include a source.

xSrc Specifies the logical x-coordinate of the upper-left corner of the source bitmap.

ySrc Specifies the logical y-coordinate of the upper-left corner of the source bitmap.

dwRop Specifies the raster operation to be performed. Raster-operation codes define how the GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. The following lists raster-operation codes for *dwRop* and their descriptions:

- **BLACKNESS** Turns all output black.
- **DSTINVERT** Inverts the destination bitmap.
- **MERGECOPY** Combines the pattern and the source bitmap using the Boolean AND operator.
- **MERGEPAINT** Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.
- **NOTSRCCOPY** Copies the inverted source bitmap to the destination.
- **NOTSRCERASE** Inverts the result of combining the destination and source bitmaps using the Boolean OR operator.
- **PATCOPY** Copies the pattern to the destination bitmap.
- **PATINVERT** Combines the destination bitmap with the pattern using the Boolean XOR operator.

- **PATPAINT** Combines the inverted source bitmap with the pattern using the Boolean OR operator. Combines the result of this operation with the destination bitmap using the Boolean OR operator.
- **SRCAND** Combines pixels of the destination and source bitmaps using the Boolean AND operator.
- **SRCCOPY** Copies the source bitmap to the destination bitmap.
- **SRCERASE** Inverts the desination bitmap and combines the result with the source bitmap using the Boolean AND operator.
- **SRCINVERT** Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
- **SRCPAINT** Combines pixels of the destination and source bitmaps using the Boolean OR operator.
- **WHITENESS** Turns all output white.

For a complete list of raster-operation codes, see the *Windows Software Development Kit* (SDK) documentation.

Remarks

Copies a bitmap from the source device context to this current device context. The application can align the windows or client areas on byte boundaries to ensure that the **BitBlt** operations occur on byte-aligned rectangles. (Set the **CS_BYTEALIGNWINDOW** or **CS_BYTEALIGNCLIENT** flags when you register the window classes.) **BitBlt** operations on byte-aligned rectangles are considerably faster than **BitBlt** operations on rectangles that are not byte aligned. If you want to specify class styles such as byte-alignment for your own device context, you will have to register a window class rather than relying on the Microsoft Foundation classes to do it for you. Use the global function **AfxRegisterWndClass**.

GDI transforms *nWidth* and *nHeight*, once by using the destination device context, and once by using the source device context. If the resulting extents do not match, GDI uses the Windows **StretchBlt** function to compress or stretch the source bitmap as necessary.

If destination, source, and pattern bitmaps do not have the same color format, the **BitBlt** function converts the source and pattern bitmaps to match the destination. The foreground and background colors of the destination bitmap are used in the conversion. When the **BitBlt** function converts a monochrome bitmap to color, it sets white bits (1) to the background color and black bits (0) to the foreground color. The foreground and background colors of the destination device context are used. To convert color to monochrome, **BitBlt** sets pixels that match the background color to white and sets all other pixels to black. **BitBlt** uses the foreground and background colors of the color device context to convert from color to monochrome.

Note that not all device contexts support **BitBlt**. To check whether a given device context does support **BitBlt**, use the **GetDeviceCaps** member function and specify the **RASTERCAPS** index.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::GetDeviceCaps**, **CDC::PatBlt**, **CDC::SetTextColor**, **CDC::StretchBlt**, **::StretchDIBits**, **::BitBlt**

CDC::CDC

CDC( );

Remarks Constructs a **CDC** object.

See Also **CDC::CreateDC**, **CDC::CreateIC**, **CDC::CreateCompatibleDC**

CDC::Chord

BOOL Chord(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3*, **int** *x4*, **int** *y4* **);**

BOOL Chord(LPCRECT *lpRect*, **POINT** *ptStart*, **POINT** *ptEnd* **);**

x1 Specifies the x-coordinate of the upper-left corner of the chord's bounding rectangle (in logical units).

y1 Specifies the y-coordinate of the upper-left corner of the chord's bounding rectangle (in logical units).

x2 Specifies the x-coordinate of the lower-right corner of the chord's bounding rectangle (in logical units).

y2 Specifies the y-coordinate of the lower-right corner of the chord's bounding rectangle (in logical units).

x3 Specifies the x-coordinate of the point that defines the chord's starting point (in logical units).

y3 Specifies the y-coordinate of the point that defines the chord's starting point (in logical units).

x4 Specifies the x-coordinate of the point that defines the chord's endpoint (in logical units).

y4 Specifies the y-coordinate of the point that defines the chord's endpoint (in logical units).

lpRect Specifies the bounding rectangle (in logical units). You can pass either a **LPRECT** or a **CRect** object for this parameter.

ptStart Specifies the x- and y-coordinates of the point that defines the chord's starting point (in logical units). This point does not have to lie exactly on the chord. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

ptEnd Specifies the x- and y-coordinates of the point that defines the chord's ending point (in logical units). This point does not have to lie exactly on the chord. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Draws a chord (a closed figure bounded by the intersection of an ellipse and a line segment). The (*x1*, *y1*) and (*x2*, *y2*) parameters specify the upper-left and lower-right corners, respectively, of a rectangle bounding the ellipse that is part of the chord. The (*x3*, *y3*) and (*x4*, *y4*) parameters specify the endpoints of a line that intersects the ellipse. The chord is drawn by using the selected pen and filled by using the selected brush. The figure drawn by the **Chord** function extends up to, but does not include the right and bottom coordinates. This means that the height of the figure is $y2 - y1$ and the width of the figure is $x2 - x1$.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::Arc, ::Chord, POINT

CDC::CreateCompatibleDC

virtual BOOL CreateCompatibleDC(CDC* *pDC* **);**

pDC A pointer to a device context. If *pDC* is **NULL**, the function creates a memory device context that is compatible with the system display.

Remarks

Creates a memory device context that is compatible with the device specified by *pDC*. A memory device context is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual device surface of the compatible device.

When a memory device context is created, GDI automatically selects a 1-by-1 monochrome stock bitmap for it. GDI output functions can be used with a memory device context only if a bitmap has been created and selected into that context.

This function can only be used to create compatible device contexts for devices that support raster operations. See the **CDC::BitBlt** member function for information regarding bit-block transfers between device contexts. To determine if a device context supports raster operations, see the **RC_BITBLT** raster capability in the member function **CDC::GetDeviceCaps**.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::CDC, **CDC::GetDeviceCaps**, **::CreateCompatibleDC**, **CDC::BitBlt**, **CDC::CreateDC**, **CDC::CreateIC**, **CDC::DeleteDC**

CDC::CreateDC

virtual BOOL CreateDC(LPCSTR *lpszDriverName*, **LPCSTR** *lpszDeviceName*, **LPCSTR** *lpszOutput*, **const void FAR*** *lpInitData* **);**

lpszDriverName Points to a null-terminated string that specifies the MS-DOS filename (without extension) of the device driver (for example, “EPSON”). You can also pass a **CString** object for this parameter.

lpszDeviceName Points to a null-terminated string that specifies the name of the specific device to be supported (for example, “EPSON FX-80”). The *lpszDeviceName* parameter is used if the module supports more than one device. You can also pass a **CString** object for this parameter.

lpszOutput Points to a null-terminated string that specifies the MS-DOS file or device name for the physical output medium (file or output port). You can also pass a **CString** object for this parameter.

lpInitData Points to a **DEVMODE** structure containing device-specific initialization data for the device driver. The Windows **ExtDeviceMode** function retrieves this structure filled in for a given device. The *lpInitData* parameter must be **NULL** if the device driver is to use the default initialization (if any) specified by the user through the Control Panel.

A **DEVMODE** structure has this form:

```
#include <print.h>

typedef struct tagDEVMODE {   /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    UINT  dmSpecVersion;
    UINT  dmDriverVersion;
    UINT  dmSize;
    UINT  dmDriverExtra;
    DWORD dmFields;
    int   dmOrientation;
    int   dmPaperSize;
    int   dmPaperLength;
    int   dmPaperWidth;
    int   dmScale;
    int   dmCopies;
    int   dmDefaultSource;
    int   dmPrintQuality;
    int   dmColor;
    int   dmDuplex;
    int   dmYResolution;
    int   dmTTOption;
} DEVMODE;
```

For more information about this structure, see **DEVMODE** in the Windows SDK documentation.

Remarks

Creates a device context for the specified device. The PRINT.H header file is required if the **DEVMODE** structure is used.

MS-DOS device names follow MS-DOS conventions; an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon. The driver and port names must not contain leading or trailing spaces. GDI output functions cannot be used with information contexts.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

::ExtDeviceMode, **::CreateDC**, **CDC::DeleteDC**, **CDC::CreateIC**

CDC::CreateIC

virtual BOOL CreateIC(LPCSTR *lpszDriverName*,
LPCSTR *lpszDeviceName*, **LPCSTR** *lpszOutput*,
const void FAR* *lpInitData* **);**

lpszDriverName Points to a null-terminated string that specifies the MS-DOS filename (without extension) of the device driver (for example, "EPSON"). You can pass a **CString** object for this parameter.

lpszDeviceName Points to a null-terminated string that specifies the name of the specific device to be supported (for example, "EPSON FX-80"). The *lpszDeviceName* parameter is used if the module supports more than one device. You can pass a **CString** object for this parameter.

lpszOutput Points to a null-terminated string that specifies the MS-DOS file or device name for the physical output medium (file or port). You can pass a **CString** object for this parameter.

lpInitData Points to device-specific initialization data for the device driver. The *lpInitData* parameter must be **NULL** if the device driver is to use the default initialization (if any) specified by the user through the Control Panel. See **CreateDC** for the data format for device-specific initialization.

Remarks

Creates an information context for the specified device. The information context provides a fast way to get information about the device without creating a device context.

MS-DOS device names follow MS-DOS conventions; an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon. The driver and port names must not contain leading or trailing spaces. GDI output functions cannot be used with information contexts.

Return Value

Nonzero if successful; otherwise 0.

See Also

CDC::CreateDC, **::CreateIC**, **CDC::DeleteDC**

CDC::DeleteDC

virtual BOOL DeleteDC( );

Remarks

In general, do not call this function; the destructor will do it for you. The **DeleteDC** member function deletes the Windows device contexts that are associated with **m_hDC** in the current **CDC** object. If this **CDC** object is the last active device context for a given device, the device is notified and all storage and system resources used by the device are released. An application should not call **DeleteDC** if objects have been selected into the device context. Objects must first be selected out of the device context before it it is deleted. An application must not delete a device context whose handle was obtained by calling **CWnd::GetDC**. Instead, it must call

CWnd::ReleaseDC to free the device context. The **CClientDC** and **CWindowDC** classes are provided to wrap this functionality. The **DeleteDC** function is generally used to delete device contexts created with **CreateDC**, **CreateIC**, or **CreateCompatibleDC**.

Return Value Nonzero if the function completed successfully; otherwise 0.

See Also **CDC::CDC**, **::DeleteDC**, **CDC::CreateDC**, **CDC::CreateIC**, **CDC::CreateCompatibleDC**, **CWnd::GetDC**, **CWnd::ReleaseDC**

CDC::DeleteTempMap

static void PASCAL DeleteTempMap( );

Remarks Called automatically by the **CWinApp** idle-time handler, **DeleteTempMap** deletes any temporary **CDC** objects created by **FromHandle**, but does not destroy the device context handles (**HDC**s) temporarily associated with the **CDC** objects.

See Also **CDC::Detach**, **CDC::FromHandle**, **CWinApp::OnIdle**

CDC::Detach

HDC Detach( );

Remarks Call this function to detach **m_hDC** (the output device context) from the **CDC** object and set both **m_hDC** and **m_hAttribDC** to **NULL**.

Return Value A Windows device context.

See Also **CDC::Attach**, **CDC::m_hDC**, **CDC::m_hAttribDC**

CDC::DPtoLP

void DPtoLP(LPPOINT *lpPoints*, **int** *nCount* **= 1) const;**

void DPtoLP(LPRECT *lpRect* **) const;**

lpPoints Points to an array of **POINT** structures or **CPoint** objects.

nCount Specifies the number of points in the array.

lpRect Points to a **RECT** structure or **CRect** object. This parameter is used for the simple case of converting one rectangle from device points to logical points.

Remarks Converts device points into logical points. The function maps the coordinates of each point from the device coordinate system into the GDI's logical coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents for the device's window and viewport.

See Also **CDC::LPtoDP**, **::DPtoLP**, **POINT**, **RECT**

CDC::DrawFocusRect

void DrawFocusRect(LPCRECT *lpRect* **);**

lpRect Points to a **RECT** structure or a **CRect** object that specifies the logical coordinates of the rectangle to be drawn.

Remarks Draws a rectangle in the style used to indicate that the rectangle has the focus. Since this is a Boolean XOR function, calling this function a second time with the same rectangle removes the rectangle from the display. The rectangle drawn by this function cannot be scrolled. To scroll an area containing a rectangle drawn by this function, first call **DrawFocusRect** to remove the rectangle from the display, then scroll the area, and then call **DrawFocusRect** again to draw the rectangle in the new position.

See Also **CDC::FrameRect**, **::DrawFocusRect**, **RECT**

CDC::DrawIcon

BOOL DrawIcon(int *x*, **int** *y*, **HICON** *hIcon* **);**

BOOL DrawIcon(POINT *point*, **HICON** *hIcon* **);**

x Specifies the logical x-coordinate of the upper-left corner of the icon.

y Specifies the logical y-coordinate of the upper-left corner of the icon.

hIcon Identifies the handle of the icon to be drawn.

point Specifies the logical x- and y-coordinates of the upper-left corner of the icon. You can pass a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Draws an icon on the device represented by the current **CDC** object. The function places the icon's upper-left corner at the location specified by *x* and *y*. The location is subject to the current mapping mode of the device context. The icon resource must have been previously loaded by using the functions **CWinApp::LoadIcon**, **CWinApp::LoadStandardIcon**, or **CWinApp::LoadOEMIcon**. The **MM_TEXT** mapping mode must be selected prior to using this function.

Return Value

Nonzero if the function completed successfully; otherwise 0.

See Also

CWinApp::LoadIcon, **CWinApp::LoadStandardIcon**, **CWinApp::LoadOEMIcon**, **CDC::GetMapMode**, **CDC::SetMapMode**, **::DrawIcon**, **POINT**

CDC::DrawText

virtual int DrawText(LPCSTR *lpszString*, **int** *nCount*, **LPRECT** *lpRect*, **UINT** *nFormat* **);**

lpszString Points to the string to be drawn. If *nCount* is –1, the string must be null-terminated.

nCount Specifies the number of bytes in the string. If *nCount* is –1, then *lpszString* is assumed to be a long pointer to a null-terminated string and **DrawText** computes the character count automatically.

lpRect Points to a **RECT** structure or **CRect** object that contains the rectangle (in logical coordinates) in which the text is to be formatted.

nFormat Specifies the method of formatting the text. It can be any combination of the following values (combine using the bitwise-OR operator), with the meanings as given:

- **DT_BOTTOM** Specifies bottom-justified text. This value must be combined with **DT_SINGLELINE**.
- **DT_CALCRECT** Determines the width and height of the rectangle. If there are multiple lines of text, **DrawText** will use the width of the rectangle pointed to by *lpRect* and extend the base of the rectangle to bound the last line of text. If there is only one line of text, **DrawText** will modify the right

side of the rectangle so that it bounds the last character in the line. In either case, **DrawText** returns the height of the formatted text but does not draw the text.

- **DT_CENTER** Centers text horizontally.
- **DT_EXPANDTABS** Expands tab characters. The default number of characters per tab is eight.
- **DT_EXTERNALLEADING** Includes the font's external leading in the line height. Normally, external leading is not included in the height of a line of text.
- **DT_LEFT** Aligns text flush-left.
- **DT_NOCLIP** Draws without clipping. **DrawText** is somewhat faster when **DT_NOCLIP** is used.
- **DT_NOPREFIX** Turns off processing of prefix characters. Normally, **DrawText** interprets the ampersand (**&**) mnemonic-prefix character as a directive to underscore the character that follows, and the two-ampersand (**&&**) mnemonic-prefix characters as a directive to print a single ampersand. By specifiying **DT_NOPREFIX** this processing is turned off.
- **DT_RIGHT** Aligns text flush-right.
- **DT_SINGLELINE** Specifies single line only. Carriage returns and linefeeds do not break the line.
- **DT_TABSTOP** Sets tab stops. The high-order byte of *nFormat* is the number of characters for each tab. The default number of characters per tab is eight.
- **DT_TOP** Specifies top-justified text (single line only).
- **DT_VCENTER** Specifies vertically centered text (single line only).
- **DT_WORDBREAK** Specifies word-breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by *lpRect*. A carriage return–linefeed sequence will also break the line.

Note that the values **DT_CALCRECT**, **DT_EXTERNALLEADING**, **DT_INTERNAL**, **DT_NOCLIP**, and **DT_NOPREFIX** cannot be used with the **DT_TABSTOP** value.

Remarks

Draws formatted text in the rectangle specified by *lpRect*. It formats text by expanding tabs into appropriate spaces, aligning text to the left, right, or center of the given rectangle, and breaking text into lines that fit within the given rectangle. The type of formatting is specified by *nFormat*. This member function uses the device context's selected font, text color, and background color to draw the text. Unless the **DT_NOCLIP** format is used, **DrawText** clips the text so that the text does not appear outside the given rectangle. All formatting is assumed to have multiple lines unless the **DT_SINGLELINE** format is given. If the selected font is too large for the specified rectangle, the **DrawText** member function does not attempt to substitute a smaller font.

If the **DT_CALCRECT** flag is specified, the rectangle specified by *lpRect* will be updated to reflect the width and height needed to draw the text.

If the **TA_UPDATECP** text-alignment flag has been set (see **CDC::SetTextAlign**), **DrawText** will display text starting at the current position, rather than at the left of the given rectangle. **DrawText** will not wrap text when the **TA_UPDATECP** flag has been set (that is, the **DT_WORDBREAK** flag will have no effect).

The text color may be set by **CDC::SetTextColor**.

Return Value The height of the text if the function is successful.

See Also **CDC::SetTextColor**, **CDC::ExtTextOut**, **CDC::TabbedTextOut**, **CDC::TextOut**, **::DrawText**, **RECT**, **CDC::SetTextAlign**

CDC::Ellipse

BOOL Ellipse(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

BOOL Ellipse(LPCRECT *lpRect* **);**

x1 Specifies the logical x-coordinate of the upper-left corner of the ellipse's bounding rectangle.

y1 Specifies the logical y-coordinate of the upper-left corner of the ellipse's bounding rectangle.

x2 Specifies the logical x-coordinate of the lower-right corner of the ellipse's bounding rectangle.

y2 Specifies the logical y-coordinate of the lower-right corner of the ellipse's bounding rectangle.

lpRect Specifies the ellipse's bounding rectangle. You can also pass a **CRect** object for this parameter.

Remarks Draws an ellipse. The center of the ellipse is the center of the bounding rectangle specified by *x1*, *y1*, *x2*, and *y2*, or *lpRect*. The ellipse is drawn with the current pen and its interior is filled with the current brush. The figure drawn by this function extends up to but does not include the right and bottom coordinates. This means that the height of the figure is $y2 - y1$ and the width of the figure is $x2 - x1$. If either the width or the height of the bounding rectangle is 0, no ellipse is drawn.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::Arc**, **CDC::Chord**, **::Ellipse**

CDC::EndDoc

int EndDoc( );

Remarks Ends a print job started by a call to the **StartDoc** member function. This member function replaces the **ENDDOC** printer escape, and should be called immediately after finishing a successful print job. If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using either **EndDoc** or **AbortDoc**. GDI automatically terminates the operation before returning the error value.

This function should not be used inside metafiles.

When used with Windows version 3.0, this member function sends the **ENDDOC** escape.

Return Value Greater than or equal to 0 if the function is successful, or a negative value if an error occurred. The following list shows common error values and their meanings:

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.
- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User ended the job through the Print Manager.

See Also **CDC::AbortDoc**, **CDC::Escape**, **CDC::StartDoc**

CDC::EndPage

int EndPage( );

Remarks Informs the device that the application has finished writing to a page. This member function is typically used to direct the device driver to advance to a new page. This member function replaces the **NEWFRAME** printer escape. Unlike **NEWFRAME**, this function is always called after printing a page.

When used with Windows version 3.0, this member function sends the **NEWFRAME** escape.

Return Value

Greater than or equal to 0 if successful; otherwise it is an error value, which can be one of the following, with its meaning as given:

- **SP_ERROR** General error.
- **SP_APPABORT** Job was ended because the application's abort function returned 0.
- **SP_USERABORT** User ended the job through Print Manager.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.
- **SP_OUTOFMEMORY** Not enough memory is available for spooling.

See Also

CDC::StartPage, **CDC::StartDoc**, **CDC::Escape**

CDC::EnumObjects

int EnumObjects(int *nObjectType*,
int (CALLBACK EXPORT* *lpfn* **)(LPVOID, LPARAM),**
LPARAM *lpData* **);**

nObjectType Specifies the object type. It can have the values **OBJ_BRUSH** or **OBJ_PEN**.

lpfn Is the procedure-instance address of the application-supplied callback function. See the "Remarks" section below.

lpData Points to the application-supplied data. The data is passed to the callback function along with the object information.

Remarks

Enumerates the pens and brushes available in a device context. For each object of a given type, the callback function that you pass is called with the information for that object. The system calls the callback function until there are no more objects or the callback function returns 0.

Note that the features of Microsoft Visual C++ let you use an ordinary function as the function passed to **EnumObjects**. The address passed to **EnumObjects** is a **FAR** pointer to a function exported with **__export** and with the Pascal calling convention. In protect-mode applications, you do not have to create this function with the Windows **MakeProcInstance** function or free the function after use with the **FreeProcInstance** Windows function. You also do not have to export the function name in an **EXPORTS** statement in your application's module-definition file. You can instead use the **__export** function modifier, as in

int FAR PASCAL __export AFunction(**LPSTR, LPSTR**);

to cause the compiler to emit the proper export record for export by name without aliasing. This works for most needs. For some special cases, such as exporting a function by ordinal or aliasing the export, you still need to use an **EXPORTS** statement in a module-definition file.

For compiling Microsoft Foundation programs, you will normally use the /GA and /GEs compiler options. The /Gw compiler option is not used with the Microsoft Foundation classes. (If you do use the Windows function **MakeProcInstance**, you will need to explicitly cast the returned function pointer from **FARPROC** to the type needed in this API.) Callback registration interfaces are now type-safe (you must pass in a function pointer that points to the right kind of function for the specific callback).

Also note that all callback functions must trap Microsoft Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see Chapter 16 in the *Class Library User's Guide.*

Callback Function

The callback function passed to **EnumObjects** must use the Pascal calling convention and must be declared **FAR**.

int CALLBACK EXPORT ObjectFunc(**LPSTR** *lpszLogObject*,
  LPSTR* *lpData* **);**

The *ObjectFunc* name is a placeholder for the application-supplied function name. The actual name must be exported as described in the "Remarks" section above. The parameters are described below:

- *lpszLogObject* Points to a **LOGPEN** or **LOGBRUSH** data structure that contains information about the logical attributes of the object.
- *lpData* Points to the application-supplied data passed to the **EnumObjects** function.

Return Value

The callback function returns an **int**. The value of this return is user-defined. If the callback function returns 0, **EnumObjects** stops enumeration early.

Return Value Specifies the last value returned by the callback function. Its meaning is user-defined.

See Also **::EnumObjects**

CDC::Escape

virtual int Escape(int *nEscape*, **int** *nCount*, **LPCSTR** *lpszInData*, **LPVOID** *lpOutData* **);**

nEscape Specifies the escape function to be performed.

For a complete list of escape functions, see Chapter 5 on printer escapes in the *Microsoft Windows Programmer's Reference, Volume 3* in the *Software Development Kit* documentation.

nCount Specifies the number of bytes of data pointed to by *lpszInData*.

lpszInData Points to the input data structure required for this escape.

lpOutData Points to the structure that is to receive output from this escape. The *lpOutData* parameter is **NULL** if no data is returned.

Remarks

Allows applications to access facilities of a particular device that are not directly available through GDI. Escape calls made by an application are translated and sent to the device driver. The *nEscape* parameter specifies the escape function to be performed. For possible values, see the chapter on printer escapes in the Windows SDK documentation. Windows version 3.1 substitutes function calls for some escapes. The following **CDC** member functions call the 3.1 functions if running with Windows version 3.1, and otherwise send the printer escapes:

- **AbortDoc** Terminates a print job. Supersedes the **ABORTDOC** escape.
- **EndDoc** Ends a print job. Supersedes the **ENDDOC** escape.
- **EndPage** Ends a page. Supersedes the **NEWFRAME** escape. Unlike **NEWFRAME**, this function is always called after printing a page.
- **SetAbortProc** Sets the abort function for a print job. Supersedes the **SETABORTPROC** escape.
- **StartDoc** Starts a print job. Supersedes the **STARTDOC** escape.
- **StartPage** Prepares printer driver to receive data. Supercedes the **NEWFRAME** and **BANDINFO** escapes.

Return Value

Positive if the function is successful, except for the **QUERYESCSUPPORT** escape, which only checks for implementation. Zero is returned if the escape is not implemented, and a negative value is returned if an error occurred. The following list shows common error values and their meanings:

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.

- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User ended the job through the Print Manager.

See Also **CDC::StartDoc**, **CDC::StartPage**, **CDC::EndPage**, **CDC::SetAbortProc**, **CDC::AbortDoc**, **CDC::EndDoc**, **::Escape**

CDC::ExcludeClipRect

virtual int ExcludeClipRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

virtual int ExcludeClipRect(LPCRECT *lpRect* **);**

x1 Specifies the logical x-coordinate of the upper-left corner of the rectangle.

y1 Specifies the logical y-coordinate of the upper-left corner of the rectangle.

x2 Specifies the logical x-coordinate of the lower-right corner of the rectangle.

y2 Specifies the logical y-coordinate of the lower-right corner of the rectangle.

lpRect Specifies the rectangle. Can also be a **CRect** object.

Remarks Creates a new clipping region that consists of the existing clipping region minus the specified rectangle. The width of the rectangle, specified by the absolute value of $x2 - x1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

Return Value Specifies the new clipping region's type. It can be any one of the following values, with meaning as given:

- **COMPLEXREGION** The region has overlapping borders.
- **ERROR** No region was created.
- **NULLREGION** The region is empty.
- **SIMPLEREGION** The region has no overlapping borders.

See Also **CDC::ExcludeUpdateRgn**, **::ExcludeClipRect**

CDC::ExcludeUpdateRgn

int ExcludeUpdateRgn(CWnd* *pWnd* **);**

pWnd Points to the window object whose window is being updated.

Remarks

Prevents drawing within invalid areas of a window by excluding an updated region in the window from the clipping region associated with the **CDC** object.

Return Value

The type of excluded region. It can be any one of the following values, with the meaning as given:

- **COMPLEXREGION** The region has overlapping borders.
- **ERROR** No region was created.
- **NULLREGION** The region is empty.
- **SIMPLEREGION** The region has no overlapping borders.

See Also

CDC::ExcludeClipRect, **::ExcludeUpdateRgn**

CDC::ExtFloodFill

BOOL ExtFloodFill(int *x*, **int** *y*, **COLORREF** *crColor*, **UINT** *nFillType* **);**

x Specifies the logical x-coordinate of the point where filling begins.

y Specifies the logical y-coordinate of the point where filling begins.

crColor Specifies the color of the boundary or of the area to be filled. The interpretation of *crColor* depends on the value of *nFillType*.

nFillType Specifies the type of flood fill to be performed. It must be one of the following values, with the meaning as given:

- **FLOODFILLBORDER** The fill area is bounded by the color specified by *crColor*. This style is identical to the filling performed by **FloodFill**.
- **FLOODFILLSURFACE** The fill area is defined by the color specified by *crColor*. Filling continues outward in all directions as long as the color is encountered. This style is useful for filling areas with multicolored boundaries.

Remarks

Fills an area of the display surface with the current brush. This member function provides more flexibility than **FloodFill** because you can specify a fill type in *nFillType*. If *nFillType* is set to **FLOODFILLBORDER**, the area is assumed to be completely bounded by the color specified by *crColor*. The function begins at the point specified by *x* and *y* and fills in all directions to the color boundary. If *nFillType* is set to **FLOODFILLSURFACE**, the function begins at the point specified by *x* and *y* and continues in all directions, filling all adjacent areas containing the color specified by *crColor*.

Only memory-device contexts and devices that support raster-display technology support **ExtFloodFill**. For more information, see the **GetDeviceCaps** member function.

Return Value

Nonzero if the function is successful; otherwise 0 if the filling could not be completed, if the given point has the boundary color specified by *crColor* (if **FLOODFILLBORDER** was requested), if the given point does not have the color specified by *crColor* (if **FLOODFILLSURFACE** was requested), or if the point is outside the clipping region.

See Also

CDC::FloodFill, **CDC::GetDeviceCaps**, **::ExtFloodFill**

CDC::ExtTextOut

virtual BOOL ExtTextOut(int *x*, **int** *y*, **UINT** *nOptions*, **LPCRECT** *lpRect*, **LPCSTR** *lpszString*, **UINT** *nCount*, **LPINT** *lpDxWidths* **);**

x Specifies the logical x-coordinate of the character cell for the first character in the specified string.

y Specifies the logical y-coordinate of the character cell for the first character in the specified string.

nOptions Specifies the rectangle type. This parameter can be one, both, or neither of the following values:

- **ETO_CLIPPED** Specifies that text is clipped to the rectangle.
- **ETO_OPAQUE** Specifies that the current background color fills the rectangle. (You can set and query the current background color with the **SetBkColor** and **GetBkColor** member functions.)

lpRect Points to a **RECT** structure that determines the dimensions of the rectangle. This parameter can be **NULL**. You can also pass a **CRect** object for this parameter.

lpszString Points to the specified character string. You can also pass a **CString** object for this parameter.

nCount Specifies the number of characters in the string.

lpDxWidths Points to an array of values that indicate the distance between origins of adjacent character cells. For instance, *lpDxWidths*[*i*] logical units will separate the origins of character cell *i* and character cell *i* + 1. If *lpDxWidths* is **NULL**, **ExtTextOut** uses the default spacing between characters.

Remarks

Writes a character string within a rectangular region using the currently selected font. The rectangular region can be opaque (filled with the current background color) and it can be a clipping region.

If *nOptions* is 0 and *lpRect* is **NULL**, the function writes text to the device context without using a rectangular region. By default, the current position is not used or updated by the function. If an application needs to update the current position when it calls **ExtTextOut**, the application can call the **CDC** member function **SetTextAlign** with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores *x* and *y* on subsequent calls to **ExtTextOut** and uses the current position instead. When an application uses **TA_UPDATECP** to update the current position, **ExtTextOut** sets the current position either to the end of the previous line of text or to the position specified by the last element of the array pointed to by *lpDxWidths*, whichever is greater.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::SetTextAlign, **CDC::TabbedTextOut**, **CDC::TextOut**, **CDC::GetBkColor**, **CDC::SetBkColor**, **CDC::SetTextColor**, **::ExtTextOut**, **RECT**

CDC::FillRect

void FillRect(LPCRECT *lpRect*, **CBrush*** *pBrush* **);**

lpRect Points to a **RECT** structure that contains the logical coordinates of the rectangle to be filled. You can also pass a **CRect** object for this parameter.

pBrush Identifies the brush used to fill the rectangle.

Remarks

Fills a given rectangle using the specified brush. The function fills the complete rectangle, including the left and top borders, but it does not fill the right and bottom borders.

The brush needs to either be created using the **CBrush** member functions **CreateHatchBrush**, **CreatePatternBrush**, and **CreateSolidBrush**, or retrieved by the **GetStockObject** Windows function. When filling the specified rectangle, **FillRect** does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but does not include, the right column and bottom row, regardless of the current mapping mode. **FillRect** compares the values of the **top**, **bottom**, **left**, and **right** members of the specified rectangle. If **bottom** is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

See Also **CBrush::CreateHatchBrush**, **CBrush::CreatePatternBrush**, **CBrush::CreateSolidBrush**, **::FillRect**, **::GetStockObject**, **RECT**, **CBrush**

CDC::FillRgn

BOOL FillRgn(CRgn* *pRgn*, **CBrush*** *pBrush* **);**

pRgn A pointer to the region to be filled. The coordinates for the given region are specified in device units.

pBrush Identifies the brush to be used to fill the region.

Remarks Fills the region specified by *pRgn* with the brush specified by *pBrush*.

The brush needs to either be created using the **CBrush** member functions **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**, or retrieved by **GetStockObject**.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::PaintRgn**, **CDC::FillRect**, **CBrush**, **CRgn**, **::FillRgn**

CDC::FloodFill

BOOL FloodFill(int *x*, **int** *y*, **COLORREF** *crColor* **);**

x Specifies the logical x-coordinate of the point where filling begins.

y Specifies the logical y-coordinate of the point where filling begins.

crColor Specifies the color of the boundary.

Remarks Fills an area of the display surface with the current brush. The area is assumed to be bounded as specified by *crColor*. The **FloodFill** function begins at the point specified by *x* and *y* and continues in all directions to the color boundary. Only memory-device contexts and devices that support raster-display technology support the **FloodFill** member function. For information about **RC_BITBLT** capability, see the **GetDeviceCaps** member function. The **ExtFloodFill** function provides similar capability but greater flexibility.

Return Value Nonzero if the function is successful; otherwise 0 is returned if the filling could not be completed, the given point has the boundary color specified by *crColor*, or the point is outside the clipping region.

See Also **CDC::ExtFloodFill**, **CDC::GetDeviceCaps**, **::FloodFill**

CDC::FrameRect

void FrameRect(LPCRECT *lpRect*, **CBrush*** *pBrush* **);**

lpRect Points to a **RECT** structure or **CRect** object that contains the logical coordinates of the upper-left and lower-right corners of the rectangle. You can also pass a **CRect** object for this parameter.

pBrush Identifies the brush to be used for framing the rectangle.

Remarks Draws a border around the rectangle specified by *lpRect*. The function uses the given brush to draw the border. The width and height of the border is always 1 logical unit. If the rectangle's **bottom** coordinate is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn. The border drawn by **FrameRect** is in the same position as a border drawn by the **Rectangle** member function using the same coordinates (if **Rectangle** uses a pen that is 1 logical unit wide). The interior of the rectangle is not filled by **FrameRect**.

See Also **CBrush**, **CDC::Rectangle**, **CDC::FrameRgn**, **::FrameRect**, **RECT**

CDC::FrameRgn

BOOL FrameRgn(CRgn* *pRgn*, **CBrush*** *pBrush*, **int** *nWidth*, **int** *nHeight* **);**

pRgn Points to the **CRgn** object that identifies the region to be enclosed in a border. The coordinates for the given region are specified in device units.

pBrush Points to the **CBrush** object that identifies the brush to be used to draw the border.

nWidth Specifies the width of the border in vertical brush strokes (in logical units, or device units if running under Windows version 3.1).

nHeight Specifies the height of the border in horizontal brush strokes (in logical units, or device units if running under Windows version 3.1).

Remarks Draws a border around the region specified by *pRgn* using the brush specified by *pBrush*.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::Rectangle**, **CDC::FrameRect**, **CBrush**, **CRgn**, **::FrameRgn**

CDC::FromHandle

static CDC* PASCAL FromHandle(HDC *hDC* **);**

hDC Contains a handle to a Windows device context.

Remarks Returns a pointer to a **CDC** object when given a handle to a device context. If a **CDC** object is not attached to the handle, a temporary **CDC** object is created and attached.

Return Value The pointer may be temporary and should not be stored beyond immediate use.

See Also **CDC::DeleteTempMap**

CDC::GetAspectRatioFilter

CSize GetAspectRatioFilter( ) const;

Remarks Retrieves the setting for the current aspect-ratio filter. The aspect ratio is the ratio formed by a device's pixel width and height. Information about a device's aspect ratio is used in the creation, selection, and display of fonts. Windows provides a special filter, the aspect-ratio filter, to select fonts designed for a particular aspect ratio from all of the available fonts. The filter uses the aspect ratio specified by the **SetMapperFlags** member function.

Return Value A **CSize** object representing the aspect ratio used by the current aspect ratio filter.

See Also **CDC::SetMapperFlags**, **::GetAspectRatioFilter**, **CSize**

CDC::GetBkColor

COLORREF GetBkColor( ) const;

Remarks

Returns the current background color. If the background mode is **OPAQUE**, the system uses the background color to fill the gaps in styled lines, the gaps between hatched lines in brushes, and the background in character cells. The system also uses the background color when converting bitmaps between color and monochrome device contexts.

Return Value

An RGB color value.

See Also

CDC::GetBkMode, **CDC::SetBkColor**, **CDC::SetBkMode**, **::GetBkColor**

CDC::GetBkMode

int GetBkMode( ) const;

Remarks

Returns the background mode. The background mode defines whether the system removes existing background colors on the drawing surface before drawing text, hatched brushes, or any pen style that is not a solid line.

Return Value

The current background mode, which can be **OPAQUE**, **TRANSPARENT**, or **TRANSPARENT1**.

See Also

CDC::GetBkColor, **CDC::SetBkColor**, **CDC::SetBkMode**, **::GetBkMode**

CDC::GetBoundsRect

Windows 3.1 Only

UINT GetBoundsRect(LPRECT *lpRectBounds*, **UINT** *flags* **);** ♦

lpRectBounds Points to a buffer that will receive the current bounding rectangle. The rectangle is returned in logical coordinates.

flags Specifies whether the bounding rectangle is to be cleared after it is returned. This parameter can be one of the following values, with the meaning as given:

- **DCB_RESET** Forces the bounding rectangle to be cleared after it is returned.
- **DCB_WINDOWMGR** Queries the Windows bounding rectangle instead of the application's.

Remarks

Returns the current accumulated bounding rectangle for the specified device context.

Return Value

Specifies the current state of the bounding rectangle if the function is successful. It can be a combination of the following values, with the meaning as given:

- **DCB_ACCUMULATE** Bounding rectangle accumulation is occuring.
- **DCB_RESET** Bounding rectangle is empty.
- **DCB_SET** Bounding rectangle is not empty.
- **DCB_ENABLE** Bounding accumulation is on.
- **DCB_DISABLE** Bounding accumulation is off.

See Also

CDC::SetBoundsRect, **::GetBoundsRect**

CDC::GetBrushOrg

CPoint GetBrushOrg( ) const;

Remarks

Retrieves the origin (in device units) of the brush currently selected for the device context. The initial brush origin is at (0,0) of the client area. The return value specifies this point in device units relative to the origin of the desktop window.

Return Value

The current origin of the brush (in device units) as a **CPoint** object.

See Also

CDC::SetBrushOrg, **::GetBrushOrg**, **CPoint**

CDC::GetCharABCWidths

Windows 3.1 Only

BOOL GetCharABCWidths(UINT *nFirst*, **UINT** *nLast*, **LPABC** *lpabc* **) const;** ♦

nFirst Specifies the first character in the range of characters from the current font for which character widths are returned.

nLast Specifies the last character in the range of characters from the current font for which character widths are returned.

lpabc Points to an array of **ABC** structures that receive the character widths when the function returns. This array must contain at least as many **ABC** structures as there are characters in the range specified by the *nFirst* and *nLast* parameters.

Remarks

Retrieves the widths of consecutive characters in a specified range from the current TrueType font. The widths are returned in logical units. This function succeeds only with TrueType fonts.

The TrueType rasterizer provides "ABC" character spacing after a specific point size has been selected. "A" spacing is the distance that is added to the current position before placing the glyph. "B" spacing is the width of the black part of the glyph. "C" spacing is added to the current position to account for the white space to the right of the glyph. The total advanced width is given by A + B + C.

When the **GetCharABCWidths** member function retrieves negative "A" or "C" widths for a character, that character includes underhangs or overhangs.

To convert the ABC widths to font design units, an application should create a font whose height (as specified in the **lfHeight** member of the **LOGFONT** structure) is equal to the value stored in the **ntmSizeEM** member of the **NEWTEXTMETRIC** structure. (The value of the **ntmSizeEM** member can be retrieved by calling the **EnumFontFamilies** Windows function.)

The ABC widths of the default character are used for characters that are outside the range of the currently selected font. To retrieve the widths of characters in non-TrueType fonts, applications should use the **GetCharWidth** member function.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

::EnumFontFamilies, **CDC::GetCharWidth**, **::GetCharABCWidths**, **ABC**

CDC::GetCharWidth

BOOL GetCharWidth(UINT *nFirstChar*, **UINT** *nLastChar*, **LPINT** *lpBuffer* **) const;**

nFirstChar Specifies the first character in a consecutive group of characters in the current font.

nLastChar Specifies the last character in a consecutive group of characters in the current font.

lpBuffer Points to a buffer that will receive the width values for a consecutive group of characters in the current font.

Remarks

Retrieves the widths of individual characters in a consecutive group of characters from the current font, using **m_hAttribDC**, the input device context. For example, if *nFirstChar* identifies the letter 'a' and *nLastChar* identifies the letter 'z', the function retrieves the widths of all lowercase characters. The function stores the values in the buffer pointed to by *lpBuffer*. This buffer must be large enough to hold all of the widths. That is, there must be at least 26 entries in the example given. If a character in the consecutive group of characters does not exist in a particular font, it will be assigned the width value of the default character.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::GetOutputCharWidth, **CDC::m_hAttribDC**, **CDC::m_hDC**, **::GetCharWidth**, **::GetCharABCWidths**, **CDC::GetCharABCWidths**

CDC::GetClipBox

virtual int GetClipBox(LPRECT *lpRect* **) const;**

lpRect Points to the **RECT** structure or **CRect** object that is to receive the rectangle dimensions.

Remarks

Retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary. The dimensions are copied to the buffer pointed to by *lpRect*.

Return Value

The clipping region's type. It can be any one of the following values, with the meaning as given:

- **COMPLEXREGION** Clipping region has overlapping borders.
- **ERROR** Device context is not valid.
- **NULLREGION** Clipping region is empty.
- **SIMPLEREGION** Clipping region has no overlapping borders.

See Also

CDC::SelectClipRgn, **::GetClipBox**, **RECT**

CDC::GetCurrentPosition

CPoint GetCurrentPosition( ) const;

Remarks

Retrieves the current position (in logical coordinates). The current position can be set with the **MoveTo** member function.

Return Value

The current position as a **CPoint** object.

See Also

CDC::MoveTo, CPoint, ::GetCurrentPosition

CDC::GetDeviceCaps

int GetDeviceCaps(int *nIndex* **) const;**

nIndex Specifies the type of information to return. It can be any one of the following values:

- **DRIVERVERSION** Version number; for example, 0x100 for 1.0.
- **TECHNOLOGY** Device technology. It can be any one of the following:

Value	Meaning
DT_PLOTTER	Vector plotter
DT_RASDISPLAY	Raster display
DT_RASPRINTER	Raster printer
DT_RASCAMERA	Raster camera
DT_CHARSTREAM	Character stream
DT_METAFILE	Metafile
DT_DISPFILE	Display file

- **HORZSIZE** Width of the physical display (in millimeters).
- **VERTSIZE** Height of the physical display (in millimeters).
- **HORZRES** Width of the display (in pixels).
- **VERTRES** Height of the display (in raster lines).
- **LOGPIXELSX** Number of pixels per logical inch along the display width.
- **LOGPIXELSY** Number of pixels per logical inch along the display height.
- **BITSPIXEL** Number of adjacent color bits for each pixel.

- **PLANES** Number of color planes.
- **NUMBRUSHES** Number of device-specific brushes.
- **NUMPENS** Number of device-specific pens.
- **NUMFONTS** Number of device-specific fonts.
- **NUMCOLORS** Number of entries in the device's color table.
- **ASPECTX** Relative width of a device pixel as used for line drawing.
- **ASPECTY** Relative height of a device pixel as used for line drawing.
- **ASPECTXY** Diagonal width of the device pixel as used for line drawing.
- **PDEVICESIZE** Size of the **PDEVICE** internal data structure.
- **CLIPCAPS** Clipping capabilities of the device. It can be one of the following:

Value	Meaning
CP_NONE	Output is not clipped.
CP_RECTANGLE	Output is clipped to rectangles.
CP_REGION	Output is clipped to regions.

- **SIZEPALETTE** Number of entries in the system palette. This index is valid only if the device driver sets the **RC_PALETTE** bit in the **RASTERCAPS** index. It is available only if the driver is written for Windows version 3.0 or later.
- **NUMRESERVED** Number of reserved entries in the system palette. This index is valid only if the device driver sets the **RC_PALETTE** bit in the **RASTERCAPS** index and is available only if the driver is written for Windows version 3.0 or higher.
- **COLORRES** Actual color resolution of the device in bits per pixel. This index is valid only if the device driver sets the **RC_PALETTE** bit in the **RASTERCAPS** index and is available only if the driver is written for Windows version 3.0 or later.
- **RASTERCAPS** Value that indicates the raster capabilities of the device. It can be a combination of the following:

Capability	Meaning
RC_BANDING	Requires banding support.
RC_BIGFONT	Supports fonts larger than 64K.
RC_BITBLT	Capable of transferring bitmaps.
RC_BITMAP64	Spports bitmaps larger than 64K.
RC_DEVBITS	Supports device bitmaps.

Capability	Meaning
RC_DI_BITMAP	Capable of supporting the **SetDIBits** and **GetDIBits** Windows functions.
RC_DIBTODEV	Capable of supporting the **SetDIBitsToDevice** Windows function.
RC_FLOODFILL	Capable of performing flood fills.
RC_GDI20_OUTPUT	Capable of supporting Windows version 2.0 features.
RC_GDI20_STATE	Includes a state block in the device context.
RC_NONE	Supports no raster operations.
RC_OP_DX_OUTPUT	Supports dev opaque and DX array.
RC_PALETTE	Specifies a palette-based device.
RC_SAVEBITMAP	Capable of saving bitmaps locally.
RC_SCALING	Capable of scaling.
RC_STRETCHBLT	Capable of performing the **StretchBlt** member function.
RC_STRETCHDIB	Capable of performing the **StretchDIBits** Windows function.

- **CURVECAPS** The curve capabilities of the device. It can be a combination of the following:

Value	Meaning
CC_NONE	Supports curves.
CC_CIRCLES	Supports circles.
CC_PIE	Supports pie wedges.
CC_CHORD	Supports chords.
CC_ELLIPSES	Supports ellipses.
CC_WIDE	Supports wide borders.
CC_STYLED	Supports styled borders.
CC_WIDESTYLED	Supports wide, styled borders.
CC_INTERIORS	Supports interiors.
CC_ROUNDRECT	Supports rectangles with rounded corners.

- **LINECAPS** Line capabilities the device supports. It can be a combination of the following:

Value	Meaning
LC_NONE	Supports no lines.
LC_POLYLINE	Supports polylines.

Value	Meaning
LC_MARKER	Supports markers.
LC_POLYMARKER	Supports polymarkers.
LC_WIDE	Supports wide lines.
LC_STYLED	Supports styled lines.
LC_WIDESTYLED	Supports wide, styled lines.
LC_INTERIORS	Supports interiors.

- **POLYGONALCAPS** Polygonal capabilities the device supports. It can be a combination of the following:

Value	Meaning
PC_NONE	Supports no polygons.
PC_POLYGON	Supports alternate fill polygons.
PC_RECTANGLE	Supports rectangles.
PC_WINDPOLYGON	Supports winding number fill polygons.
PC_SCANLINE	Supports scan lines.
PC_WIDE	Supports wide borders.
PC_STYLED	Supports styled borders.
PC_WIDESTYLED	Supports wide, styled borders.
PC_INTERIORS	Supports interiors.

- **TEXTCAPS** Text capabilities the device supports. It can be a combination of the following:

Value	Meaning
TC_OP_CHARACTER	Supports character output precision, which indicates the device can place device fonts at any pixel location. This is required for any device with device fonts.
TC_OP_STROKE	Supports stroke output precision, which indicates the device can omit any stroke of a device font.
TC_CP_STROKE	Supports stroke clip precision, which indicates the device can clip device fonts to a pixel boundary.
TC_CR_90	Supports 90-degree character rotation, which indicates the device can rotate characters only 90 degrees at a time.
TC_CR_ANY	Supports character rotation at any degree, which indicates the device can rotate device fonts through any angle.

Value	Meaning
TC_SF_X_YINDEP	Supports scaling independent of x and y directions, which indicates the device can scale device fonts separately in x and y directions.
TC_SA_DOUBLE	Supports doubled characters for scaling, which indicates the device can double the size of device fonts.
TC_SA_INTEGER	Supports integer multiples for scaling, which indicates the device can scale the size of device fonts in any integer multiple.
TC_SA_CONTIN	Supports any multiples for exact scaling, which indicates the device can scale device fonts by any amount but still preserve the x and y ratios.
TC_EA_DOUBLE	Supports double-weight characters, which indicates the device can make device fonts bold. If this bit is not set for printer drivers, GDI attempts to create bold device fonts by printing them twice.
TC_IA_ABLE	Supports italics, which indicates the device can make device fonts italic. If this bit is not set, GDI assumes italics are not available.
TC_UA_ABLE	Supports underlining, which indicates the device can underline device fonts. If this bit is not set, GDI creates underlines for device fonts.
TC_SO_ABLE	Supports strikeouts, which indicates the device can strikeout device fonts. If this bit is not set, GDI creates strikeouts for device fonts.
TC_RA_ABLE	Supports raster fonts, which indicates that GDI should enumerate any raster or TrueType fonts available for this device in response to a call to the **EnumFonts** or **EnumFontFamilies** Windows functions. If this bit is not set, GDI-supplied raster or TrueType fonts are not enumerated when these functions are called.
TC_VA_ABLE	Supports vector fonts, which indicates that GDI should enumerate any vector fonts available for this device in response to a call to the **EnumFonts** or **EnumFontFamilies** Windows functions. This is significant for vector devices only (that is, for plotters). Display drivers (which must be able to use raster fonts) and raster printer drivers always enumerate vector fonts, because GDI rasterizes vector fonts before sending them to the driver.
TC_RESERVED	Reserved; must be 0.

Remarks Retrieves a wide range of device-specific information about the display device.

Return Value The value of the requested capability if the function is successful.

See Also **::GetDeviceCaps**

CDC::GetFontData

Windows 3.1 Only **DWORD GetFontData(DWORD** *dwTable*, **DWORD** *dwOffset*, **LPVOID** *lpData*, **DWORD** *cbData* **) const; ♦**

dwTable Specifies the name of the metric table to be returned. This parameter can be one of the metric tables documented in the TrueType Font Files specification published by Microsoft Corporation. If this parameter is 0, the information is retrieved starting at the beginning of the font file.

dwOffset Specifies the offset from the beginning of the table at which to begin retrieving information. If this parameter is 0, the information is retrieved starting at the beginning of the table specified by the *dwTable* parameter. If this value is greater than or equal to the size of the table, **GetFontData** returns 0.

lpData Points to a buffer that will receive the font information. If this value is **NULL**, the function returns the size of the buffer required for the font data specified in the *dwTable* parameter.

cbData Specifies the length, in bytes, of the information to be retrieved. If this parameter is 0, **GetFontData** returns the size of the data specified in the *dwTable* parameter.

Remarks Retrieves font-metric information from a scalable font file. The information to retrieve is identified by specifying an offset into the font file and the length of the information to return. An application can sometimes use the **GetFontData** member function to save a TrueType font with a document. To do this, the application determines whether the font can be embedded and then retrieves the entire font file, specifying 0 for the *dwTable*, *dwOffset*, and *cbData* parameters.

Applications can determine whether a font can be embedded by checking the **otmfsType** member of the **OUTLINETEXTMETRIC** structure. If bit 1 of **otmfsType** is set, embedding is not permitted for the font. If bit 1 is clear, the font can be embedded. If bit 2 is set, the embedding is read only. If an application attempts to use this function to retrieve information for a non-TrueType font, the **GetFontData** member function returns –1.

Return Value Specifies the number of bytes returned in the buffer pointed to by *lpData* if the function is successful; otherwise –1.

See Also **CDC::GetOutlineTextMetrics**, **::GetFontData**, **OUTLINETEXTMETRIC**

CDC::GetGlyphOutline

Windows 3.1 Only **DWORD GetGlyphOutline(UINT** *nChar*, **UINT** *nFormat*, **LPGLYPHMETRICS** *lpgm*, **DWORD** *cbBuffer*, **LPVOID** *lpBuffer*, **const MAT2 FAR*** *lpmat2* **) const;** ♦

nChar Specifies the character for which information is to be returned.

nFormat Specifies the format in which the function is to return information. It can be one of the following values, or 0:

Value	Meaning
GGO_BITMAP	Returns the glyph bitmap. When the function returns, the buffer pointed to by *lpBuffer* contains a 1-bit-per-pixel bitmap whose rows start on doubleword boundaries.
GGO_NATIVE	Returns the curve data points in the rasterizer's native format, using device units. When this value is specified, any transformation specified in *lpmat2* is ignored.

When the value of *nFormat* is 0, the function fills in a **GLYPHMETRICS** structure but does not return glyph-outline data.

lpgm Points to a **GLYPHMETRICS** structure that describes the placement of the glyph in the character cell.

cbBuffer Specifies the size of the buffer into which the function copies information about the outline character. If this value is 0 and the *nFormat* parameter is either the **GGO_BITMAP** or **GGO_NATIVE** values, the function returns the required size of the buffer.

lpBuffer Points to a buffer into which the function copies information about the outline character. If *nFormat* specifies the **GGO_NATIVE** value, the information is copied in the form of **TTPOLYGONHEADER** and **TTPOLYCURVE** structures. If this value is **NULL** and *nFormat* is either the **GGO_BITMAP** or **GGO_NATIVE** value, the function returns the required size of the buffer.

lpmat2 Points to a **MAT2** structure that contains a transformation matrix for the character. This parameter cannot be **NULL**, even when the **GGO_NATIVE** value is specified for *nFormat*.

Remarks

Retrieves the outline curve or bitmap for an outline character in the current font. An application can rotate characters retrieved in bitmap format by specifying a 2-by-2 transformation matrix in the structure pointed to by *lpmat2*.

A glyph outline is returned as a series of contours. Each contour is defined by a **TTPOLYGONHEADER** structure followed by as many **TTPOLYCURVE** structures as are required to describe it. All points are returned as **POINTFX** structures and represent absolute positions, not relative moves. The starting point given by the **pfxStart** member of the **TTPOLYGONHEADER** structure is the point at which the outline for a contour begins. The **TTPOLYCURVE** structures that follow can be either polyline records or spline records. Polyline records are a series of points; lines drawn between the points describe the outline of the character. Spline records represent the quadratic curves used by TrueType (that is, quadratic b-splines).

Return Value

The size, in bytes, of the buffer required for the retrieved information if *cbBuffer* is 0 or *lpBuffer* is **NULL**. Otherwise, it is a positive value if the function is successful, or –1 if there is an error.

See Also

CDC::GetOutlineTextMetrics, **::GetGlyphOutline**, **GLYPHMETRICS**, **TTPOLYGONHEADER**, **TTPOLYCURVE**

CDC::GetKerningPairs

Windows 3.1 Only

int GetKerningPairs(int *nPairs*, **LPKERNINGPAIR** *lpkrnpair* **) const;** ♦

nPairs Specifies the number of **KERNINGPAIR** structures pointed to by *lpkrnpair*. The function will not copy more kerning pairs than specified by *nPairs*.

lpkrnpair Points to an array of **KERNINGPAIR** structures that receive the kerning pairs when the function returns. This array must contain at least as many structures as specified by *nPairs*. If this parameter is **NULL**, the function returns the total number of kerning pairs for the font.

Remarks

Retrieves the character kerning pairs for the font that is currently selected in the specified device context.

Return Value Specifies the number of kerning pairs retrieved or the total number of kerning pairs in the font, if the function is successful. Zero is returned if the function fails or there are no kerning pairs for the font.

See Also **::GetKerningPairs, KERNINGPAIR**

CDC::GetMapMode

int GetMapMode( ) const;

Remarks Retrieves the current mapping mode. See the **SetMapMode** member function for a description of the mapping modes.

Return Value The mapping mode.

See Also **CDC::SetMapMode, ::GetMapMode**

CDC::GetNearestColor

COLORREF GetNearestColor(COLORREF *crColor* **) const;**

crColor Specifies the color to be matched.

Remarks Returns the solid color that best matches a specified logical color. The given device must be able to represent this color.

Return Value An RGB (red, green, blue) color value that defines the solid color closest to the *crColor* value that the device can represent.

See Also **::GetNearestColor, CPalette::GetNearestPaletteIndex**

CDC::GetOutlineTextMetrics

Windows 3.1 Only **UINT GetOutlineTextMetrics(UINT** *cbData*, **LPOUTLINETEXTMETRIC** *lpotm* **) const;** ♦

cbData Specifies the size, in bytes, of the buffer to which information is returned.

lpotm Points to an **OUTLINETEXTMETRIC** structure. If this parameter is **NULL**, the function returns the size of the buffer required for the retrieved metric information.

Remarks

Retrieves metric information for TrueType fonts. The **OUTLINETEXTMETRIC** structure contains most of the font metric information provided with the TrueType format, including a **TEXTMETRIC** structure. The last four members of the **OUTLINETEXTMETRIC** structure are pointers to strings. Applications should allocate space for these strings in addition to the space required for the other members. Because there is no system-imposed limit to the size of the strings, the simplest method for allocating memory is to retrieve the required size by specifying **NULL** for *lpotm* in the first call to the **GetOutlineTextMetrics** function.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

::GetTextMetrics, **::GetOutlineTextMetrics**, **CDC::GetTextMetrics**

CDC::GetOutputCharWidth

BOOL GetOutputCharWidth(UINT *nFirstChar*, **UINT** *nLastChar*, **LPINT** *lpBuffer* **) const;**

nFirstChar Specifies the first character in a consecutive group of characters in the current font.

nLastChar Specifies the last character in a consecutive group of characters in the current font.

lpBuffer Points to a buffer that will receive the width values for a consecutive group of characters in the current font.

Remarks

Uses the output device context, **m_hDC**, and retrieves the widths of individual characters in a consecutive group of characters from the current font. For example, if *nFirstChar* identifies the letter 'a' and *nLastChar* identifies the letter 'z', the function retrieves the widths of all lowercase characters. The function stores the values in the buffer pointed to by *lpBuffer*. This buffer must be large enough to hold all of the widths; that is, there must be at least 26 entries in the example given. If a character in the consecutive group of characters does not exist in a particular font, it will be assigned the width value of the default character.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::GetCharWidth, **CDC::m_hAttribDC**, **CDC::m_hDC**, **::GetCharWidth**

CDC::GetOutputTabbedTextExtent

CSize GetOutputTabbedTextExtent(LPCSTR *lpszString*, **int** *nCount*, **int** *nTabPositions*, **LPINT** *lpnTabStopPositions* **) const;**

lpszString Points to a character string. You can also pass a **CString** object for this parameter.

nCount Specifies the number of characters in the string.

nTabPositions Specifies the number of tab-stop positions in the array pointed to by *lpnTabStopPositions*.

lpnTabStopPositions Points to an array of integers containing the tab-stop positions in logical units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array. Back tabs are not allowed.

Remarks

Computes the width and height of a character string using **m_hDC**, the output device context. If the string contains one or more tab characters, the width of the string is based upon the tab stops specified by *lpnTabStopPositions*. The function uses the currently selected font to compute the dimensions of the string. The current clipping region does not offset the width and height returned by the **GetOutputTabbedTextExtent** function.

Since some devices do not place characters in regular cell arrays (that is, they kern the characters), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If *nTabPositions* is 0 and *lpnTabStopPositions* is **NULL**, tabs are expanded to eight average character widths. If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points. If *lpnTabStopPositions* points to more than a single value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

Return Value

The dimensions of the string (in logical units).

See Also

CDC::GetTextExtent, **CDC::m_hAttribDC**, **CDC::m_hDC**, **CDC::GetTabbedTextExtent**, **CDC::GetOutputTextExtent**, **CDC::TabbedTextOut**, **::GetTabbedTextExtent**, **CSize**

CDC::GetOutputTextExtent

CSize GetOutputTextExtent(LPCSTR *lpszString*, **int** *nCount* **) const;**

lpszString Points to a string of characters. You can also pass a **CString** object for this parameter.

nCount Specifies the number of characters in the string.

Remarks

This member function uses the output device context, **m_hDC**, and computes the width and height of a line of text, using the current font. The current clipping region does not affect the width and height returned by **GetOutputTextExtent**.

Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

Return Value

The dimensions of the string (in logical units) returned in a **CSize** object

See Also

CDC::GetTabbedTextExtent, **CDC::m_hAttribDC**, **CDC::m_hDC**, **CDC::GetTextExtent**, **::GetTextExtent**, **CDC::SetTextJustification**, **CSize**

CDC::GetOutputTextMetrics

BOOL GetOutputTextMetrics(LPTEXTMETRIC *lpMetrics* **) const;**

lpMetrics Points to the **TEXTMETRIC** structure that receives the metrics.

Remarks

Retrieves the metrics for the current font using **m_hDC**, the output device context.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::GetTextAlign, **CDC::m_hAttribDC**, **CDC::m_hDC**, **CDC::GetTextMetrics**, **CDC::GetTextExtent**, **CDC::GetTextFace**, **CDC::SetTextJustification**, **::GetTextMetrics**

CDC::GetPixel

COLORREF GetPixel(int *x*, **int** *y* **) const;**

COLORREF GetPixel(POINT *point* **) const;**

x Specifies the logical x-coordinate of the point to be examined.

y Specifies the logical y-coordinate of the point to be examined.

point Specifies the logical x- and y-coordinates of the point to be examined.

Remarks Retrieves the RGB color value of the pixel at the point specified by *x* and *y*. The point must be in the clipping region. If the point is not in the clipping region, the function has no effect and returns –1. Not all devices support the **GetPixel** function. For more information, see the **RC_BITBLT** raster capability under the **GetDeviceCaps** member function.

The **GetPixel** member function has two forms. The first takes two coordinate values; the second takes either a **POINT** structure or a **CPoint** object.

Return Value For either version of the function, an RGB color value for the color of the given point. It is –1 if the coordinates do not specify a point in the clipping region.

See Also **CDC::GetDeviceCaps**, **CDC::SetPixel**, **::GetPixel**, **POINT**, **CPoint**

CDC::GetPolyFillMode

int GetPolyFillMode( ) const;

Remarks Retrieves the current polygon-filling mode. See the **SetPolyFillMode** member function for a description of the polygon-filling modes.

Return Value The current polygon-filled mode, **ALTERNATE** or **WINDING**, if the function is successful.

See Also **CDC::SetPolyFillMode**, **::GetPolyFillMode**

CDC::GetROP2

int GetROP2( ) const;

Remarks Retrieves the current drawing mode. The drawing mode specifies how the colors of the pen and the interior of filled objects are combined with the color already on the display surface.

Return Value The drawing mode. For a list of the drawing mode values, see the **SetROP2** member function.

See Also **CDC::GetDeviceCaps**, **CDC::SetROP2**, **::GetROP2**

CDC::GetSafeHdc

HDC GetSafeHdc( ) const;

Remarks Call this member function to get **m_hDC**, the output device context. This member function also works with null pointers.

Return Value A device context handle.

CDC::GetStretchBltMode

int GetStretchBltMode( ) const;

Remarks Retrieves the current bitmap-stretching mode. The bitmap-stretching mode defines how information is removed from bitmaps that are stretched or compressed by the **StretchBlt** member function. The **STRETCH_ANDSCANS** and **STRETCH_ORSCANS** modes are typically used to preserve foreground pixels in monochrome bitmaps. The **STRETCH_DELETESCANS** mode is typically used to preserve color in color bitmaps.

Return Value The return value specifies the current bitmap-stretching mode—**STRETCH_ANDSCANS**, **STRETCH_DELETESCANS**, or **STRETCH_ORSCANS**—if the function is successful.

See Also **CDC::StretchBlt**, **CDC::SetStretchBltMode**, **::GetStretchBltMode**

CDC::GetTabbedTextExtent

CSize GetTabbedTextExtent(LPCSTR *lpszString*, **int** *nCount*, **int** *nTabPositions*, **LPINT** *lpnTabStopPositions* **) const;**

lpszString Points to a character string. You can also pass a **CString** object for this parameter.

nCount Specifies the number of characters in the string.

nTabPositions Specifies the number of tab-stop positions in the array pointed to by *lpnTabStopPositions*.

lpnTabStopPositions Points to an array of integers containing the tab-stop positions in logical units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array. Back tabs are not allowed.

Remarks

Computes the width and height of a character string using **m_hAttribDC**, the attribute device context. If the string contains one or more tab characters, the width of the string is based upon the tab stops specified by *lpnTabStopPositions*. The function uses the currently selected font to compute the dimensions of the string. The current clipping region does not offset the width and height returned by the **GetTabbedTextExtent** function.

Since some devices do not place characters in regular cell arrays (that is, they kern the characters), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If *nTabPositions* is 0 and *lpnTabStopPositions* is **NULL**, tabs are expanded to eight times the average character width. If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points. If *lpnTabStopPositions* points to more than a single value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

Return Value

The dimensions of the string (in logical units).

See Also

CDC::GetTextExtent, **CDC::GetOutputTabbedTextExtent**, **CDC::GetOutputTextExtent**, **CDC::TabbedTextOut**, **::GetTabbedTextExtent**, **CSize**

CDC::GetTextAlign

UINT GetTextAlign( ) const;

Remarks

Retrieves the status of the text-alignment flags for the device context. The text-alignment flags determine how the **TextOut** and **ExtTextOut** member functions align a string of text in relation to the string's starting point. The text-alignment flags are not necessarily single-bit flags and may be equal to 0. To test whether a flag is set, an application should follow these steps:

1. Apply the bitwise-OR operator to the flag and its related flags. The following list shows the groups of related flags:
   - **TA_LEFT**, **TA_CENTER**, and **TA_RIGHT**
   - **TA_BASELINE**, **TA_BOTTOM**, and **TA_TOP**
   - **TA_NOUPDATECP** and **TA_UPDATECP**
2. Apply the bitwise-AND operator to the result and the return value of **GetTextAlign**.
3. Test for the equality of this result and the flag.

Return Value

The status of the text-alignment flags. The return value is one or more of the following values, with the meaning as given:

- **TA_BASELINE** Specifies alignment of the x-axis and the baseline of the chosen font within the bounding rectangle.
- **TA_BOTTOM** Specifies alignment of the x-axis and the bottom of the bounding rectangle.
- **TA_CENTER** Specifies alignment of the y-axis and the center of the bounding rectangle.
- **TA_LEFT** Specifies alignment of the y-axis and the left side of the bounding rectangle.
- **TA_NOUPDATECP** Specifies that the current position is not updated.
- **TA_RIGHT** Specifies alignment of the y-axis and the right side of the bounding rectangle.
- **TA_TOP** Specifies alignment of the x-axis and the top of the bounding rectangle.
- **TA_UPDATECP** Specifies that the current position is updated.

See Also

CDC::ExtTextOut, **CDC::SetTextAlign**, **CDC::TextOut**, **::GetTextAlign**

CDC::GetTextCharacterExtra

int GetTextCharacterExtra( ) const;

Remarks

Retrieves the current setting for the amount of intercharacter spacing. GDI adds this spacing to each character, including break characters, when it writes a line of text to the device context. The default value for the amount of intercharacter spacing is 0.

Return Value The amount of the intercharacter spacing.

See Also **CDC::SetTextCharacterExtra**, **::GetTextCharacterExtra**

CDC::GetTextColor

COLORREF GetTextColor( ) const;

Remarks Retrieves the current text color. The text color is the foreground color of characters drawn by using the GDI text-output member functions **TextOut**, **ExtTextOut**, and **TabbedTextOut**.

Return Value The current text color as an RGB color value.

See Also **CDC::GetBkColor**, **CDC::GetBkMode**, **CDC::SetBkMode**, **CDC::SetTextColor**, **::GetTextColor**

CDC::GetTextExtent

CSize GetTextExtent(LPCSTR *lpszString*, **int** *nCount* **) const;**

lpszString Points to a string of characters. You can also pass a **CString** object for this parameter.

nCount Specifies the number of characters in the string.

Remarks Computes the width and height of a line of text using the current font to determine the dimensions. The information is retrieved from **m_hAttribDC**, the attribute device context. The current clipping region does not affect the width and height returned by **GetTextExtent**.

Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

Return Value The dimensions of the string (in logical units) in a **CSize** object.

See Also **CDC::GetTabbedTextExtent**, **CDC::m_hAttribDC**, **CDC::m_hDC**, **CDC::GetOutputTextExtent**, **::GetTextExtent**, **CDC::SetTextJustification**, **CSize**

CDC::GetTextFace

int GetTextFace(int *nCount*, **LPSTR** *lpszFacename* **) const;**

nCount Specifies the size of the buffer (in bytes). If the typeface name is longer than the number of bytes specified by this parameter, the name is truncated.

lpszFacename Points to the buffer for the typeface name.

Remarks Copies the typeface name of the current font into a buffer. The typeface name is copied as a null-terminated string.

Return Value The number of bytes copied to the buffer, not including the terminating null character. It is 0 if an error occurs.

See Also **CDC::GetTextMetrics**, **CDC::SetTextAlign**, **CDC::TextOut**, **::GetTextFace**

CDC::GetTextMetrics

BOOL GetTextMetrics(LPTEXTMETRIC *lpMetrics* **) const;**

lpMetrics Points to the **TEXTMETRIC** structure that receives the metrics.

A **TEXTMETRIC** structure has this form:

```
typedef struct tagTEXTMETRIC {  /* tm */
    int  tmHeight;
    int  tmAscent;
    int  tmDescent;
    int  tmInternalLeading;
    int  tmExternalLeading;
    int  tmAveCharWidth;
    int  tmMaxCharWidth;
    int  tmWeight;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmFirstChar;
    BYTE tmLastChar;
    BYTE tmDefaultChar;
    BYTE tmBreakChar;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
    int  tmOverhang;
    int  tmDigitizedAspectX;
    int  tmDigitizedAspectY;
} TEXTMETRIC;
```

For more complete information about this structure, see **TEXTMETRIC** in the Windows SDK documentation.

Remarks Retrieves the metrics for the current font using the attribute device context.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::GetTextAlign**, **CDC::m_hAttribDC**, **CDC::m_hDC**, **CDC::GetOutputTextMetrics**, **CDC::GetTextExtent**, **CDC::GetTextFace**, **CDC::SetTextJustification**, **::GetTextMetrics**

CDC::GetViewportExt

CSize GetViewportExt( ) const;

Remarks Retrieves the x- and y-extents of the device context's viewport.

Return Value The x- and y-extents (in device units) as a **CSize** object.

See Also **CDC::SetViewportExt**, **CSize**, **::GetViewportExt**, **CDC::SetWindowExt**

CDC::GetViewportOrg

CPoint GetViewportOrg( ) const;

Remarks Retrieves the x- and y-coordinates of the origin of the viewport associated with the device context.

Return Value The origin of the viewport (in device coordinates) as a **CPoint** object.

See Also **CDC::GetWindowOrg**, **CPoint**, **::GetViewportOrg**, **CDC::SetViewportOrg**

CDC::GetWindowExt

CSize GetWindowExt( ) const;

Remarks Retrieves the x- and y-extents of the window associated with the device context.

Return Value The x- and y-extents (in logical units) as a **CSize** object.

See Also **CDC::SetWindowExt**, **CSize**, **::GetWindowExt**, **CDC::GetViewportExt**

CDC::GetWindowOrg

CPoint GetWindowOrg( ) const;

Remarks Retrieves the x- and y-coordinates of the origin of the window associated with the device context.

Return Value The origin of the window (in logical coordinates) as a **CPoint** object.

See Also **CDC::GetViewportOrg**, **CDC::SetWindowOrg**, **CPoint**, **::GetWindowOrg**

CDC::GrayString

virtual BOOL GrayString(CBrush* *pBrush*,
BOOL (CALLBACK EXPORT* *lpfnOutput* **)(HDC, LPARAM, int),**
LPARAM *lpData*, **int** *nCount*, **int** *x*, **int** *y*, **int** *nWidth*, **int** *nHeight* **);**

pBrush Identifies the brush to be used for dimming (graying).

lpfnOutput Specifies the procedure-instance address of the application-supplied callback function that will draw the string. For more information, see the description of the Windows **OutputFunc** callback function below. If this parameter is **NULL**, the system uses the Windows **TextOut** function to draw the string, and *lpData* is assumed to be a long pointer to the character string to be output.

lpData Specifies a far pointer to data to be passed to the output function. If *lpfnOutput* is **NULL**, *lpData* must be a long pointer to the string to be output.

nCount Specifies the number of characters to be output. If this parameter is 0, **GrayString** calculates the length of the string (assuming that *lpData* is a pointer to the string). If *nCount* is –1 and the function pointed to by *lpfnOutput* returns 0, the image is shown but not dimmed.

x Specifies the logical x-coordinate of the starting position of the rectangle that encloses the string.

y Specifies the logical y-coordinate of the starting position of the rectangle that encloses the string.

nWidth Specifies the width (in logical units) of the rectangle that encloses the string. If *nWidth* is 0, **GrayString** calculates the width of the area, assuming *lpData* is a pointer to the string.

nHeight Specifies the height (in logical units) of the rectangle that encloses the string. If *nHeight* is 0, **GrayString** calculates the height of the area, assuming *lpData* is a pointer to the string.

Remarks

Draws dimmed (gray) text at the given location by writing the text in a memory bitmap, dimming the bitmap, and then copying the bitmap to the display. The function dims the text regardless of the selected brush and background. The **GrayString** member function uses the currently selected font. The **MM_TEXT** mapping mode must be selected before using this function.

An application can draw dimmed (grayed) strings on devices that support a solid gray color without calling the **GrayString** member function. The system color **COLOR_GRAYTEXT** is the solid-gray system color used to draw disabled text. The application can call the **GetSysColor** Windows function to retrieve the color value of **COLOR_GRAYTEXT**. If the color is other than 0 (black), the application can call the **SetTextColor** member function to set the text color to the color value and then draw the string directly. If the retrieved color is black, the application must call **GrayString** to dim (gray) the text.

If *lpfnOutput* is **NULL**, GDI uses the Windows **TextOut** function, and *lpData* is assumed to be a far pointer to the character to be output. If the characters to be output cannot be handled by the **TextOut** member function (for example, the string is stored as a bitmap), the application must supply its own output function. Also note that all callback functions must trap Microsoft Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see Chapter 16 in the *Class Library User's Guide*. The callback function passed to **GrayString** must use the Pascal calling convention, must be exported with **__export**, and must be declared **FAR**.

When the framework is in preview mode, a call to the **GrayString** member function is translated to a **TextOut** call, and the callback function is not called.

Callback Function

BOOL CALLBACK EXPORT *OutputFunc*(**HDC** *hDC*, **LPARAM** *lpData*, **int** *nCount*);

OutputFunc is a placeholder for the application-supplied callback function name. The callback function (*OutputFunc*) must draw an image relative to the coordinates (0,0) rather than (*x*, *y*). The parameters are described below:

hDC Identifies a memory device context with a bitmap of at least the width and height specified by *nWidth* and *nHeight* to **GrayString**.

lpData Points to the character string to be drawn.

nCount Specifies the number of characters to output.

Return Value

The callback function's return value must be **TRUE** to indicate success; otherwise it is **FALSE**.

Return Value Nonzero if the string is drawn, or 0 if either the **TextOut** function or the application-supplied output function returned 0, or there was insufficient memory to create a memory bitmap for dimming.

See Also **::GetSysColor**, **CDC::SetTextColor**, **CDC::TextOut**, **::GrayString**

CDC::IntersectClipRect

virtual int IntersectClipRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

virtual int IntersectClipRect(LPCRECT *lpRect* **);**

x1 Specifies the logical x-coordinate of the upper-left corner of the rectangle.

y1 Specifies the logical y-coordinate of the upper-left corner of the rectangle.

x2 Specifies the logical x-coordinate of the lower-right corner of the rectangle.

y2 Specifies the logical y-coordinate of the lower-right corner of the rectangle.

lpRect Specifies the rectangle. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

Remarks

Creates a new clipping region by forming the intersection of the current region and the rectangle specified by *x1*, *y1*, *x2*, and *y2*. GDI clips all subsequent output to fit within the new boundary. The width and height must not exceed 32,767.

Return Value

The new clipping region's type. It can be any one of the following values, with the meaning as given:

- **COMPLEXREGION** New clipping region has overlapping borders.
- **ERROR** Device context is not valid.
- **NULLREGION** New clipping region is empty.
- **SIMPLEREGION** New clipping region has no overlapping borders.

See Also

::IntersectClipRect, **CRect**, **RECT**

CDC::InvertRect

void InvertRect(LPCRECT *lpRect* **);**

lpRect Points to a **RECT** that contains the logical coordinates of the rectangle to be inverted. You can also pass a **CRect** object for this parameter.

Remarks

Inverts the contents of the given rectangle. Inversion is a logical NOT operation and flips the bits of each pixel. On monochrome displays, the function makes white pixels black and black pixels white. On color displays, the inversion depends on how colors are generated for the display. Calling **InvertRect** twice with the same rectangle restores the display to its previous colors. If the rectangle is empty, nothing is drawn.

See Also **CDC::FillRect**, **::InvertRect**, **CRect**, **RECT struct**

CDC::InvertRgn

BOOL InvertRgn(CRgn* *pRgn* **);**

pRgn Identifies the region to be inverted. The coordinates for the region are specified in device units.

Remarks Inverts the colors in the region specified by *pRgn*. On monochrome displays, the function makes white pixels black and black pixels white. On color displays, the inversion depends on how the colors are generated for the display.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::FillRgn**, **CDC::PaintRgn**, **CRgn**, **::InvertRgn**

CDC::IsPrinting

BOOL IsPrinting( ) const;

Return Value Nonzero if the **CDC** object is currently printing; otherwise 0.

CDC::LineTo

BOOL LineTo(int *x*, **int** *y* **);**

BOOL LineTo(POINT *point* **);**

x Specifies the logical x-coordinate of the endpoint for the line.

y Specifies the logical y-coordinate of the endpoint for the line.

point Specifies the endpoint for the line. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks Draws a line from the current position up to, but not including, the point specified by *x* and *y* (or *point*). The line is drawn with the selected pen. The current position is set to *x*,*y* or to *point*.

Return Value Nonzero if the line is drawn; otherwise 0.

See Also **CDC::MoveTo**, **CDC::GetCurrentPosition**, **::LineTo**, **CPoint**, **POINT**

CDC::LPtoDP

void LPtoDP(LPPOINT *lpPoints*, **int** *nCount* **= 1) const;**

void LPtoDP(LPRECT *lpRect* **) const;**

lpPoints Points to an array of points. Each point in the array is a **POINT** structure or a **CPoint** object.

nCount Specifies the number of points in the array.

lpRect Points to a **RECT** structure or a **CRect** object. This parameter is used for the common case of mapping a rectangle from logical to device units.

Remarks Converts logical points into device points. The function maps the coordinates of each point from GDI's logical coordinate system into a device coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents of the device's window and viewport. The x- and y-coordinates of points are 2-byte signed integers in the range –32,768 through 32,767. In cases where the mapping mode would result in values larger than these limits, the system sets the values to –32,768 and 32,767, respectively.

See Also **CDC::DPtoLP**, **::LPtoDP**, **CPoint**, **POINT**, **RECT**, **CRect**

CDC::MoveTo

CPoint MoveTo(int *x*, **int** *y* **);**

CPoint MoveTo(POINT *point* **);**

x Specifies the logical x-coordinate of the new position.

y Specifies the logical y-coordinate of the new position.

point Specifies the new position. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks Moves the current position to the point specified by *x* and *y* (or by *point*).

Return Value The x- and y-coordinates of the previous position as a **CPoint** object.

See Also **CDC::GetCurrentPosition**, **CDC::LineTo**, **::MoveTo**, **CPoint**, **POINT**

CDC::OffsetClipRgn

virtual int OffsetClipRgn(int *x*, **int** *y* **);**

virtual int OffsetClipRgn(SIZE *size* **);**

x Specifies the number of logical units to move left or right.

y Specifies the number of logical units to move up or down.

size Specifies the amount to offset.

Remarks Moves the clipping region of the device context by the specified offsets. The function moves the region *x* units along the x-axis and *y* units along the y-axis.

Return Value The new region's type. It can be any one of the following values, with the meanings as given:

- **COMPLEXREGION** Clipping region has overlapping borders.
- **ERROR** Device context is not valid.
- **NULLREGION** Clipping region is empty.
- **SIMPLEREGION** Clipping region has no overlapping borders.

See Also **CDC::SelectClipRgn**, **::OffsetClipRgn**

CDC::OffsetViewportOrg

virtual CPoint OffsetViewportOrg(int *nWidth*, **int** *nHeight* **);**

nWidth Specifies the number of device units to add to the current origin's x-coordinate.

nHeight Specifies the number of device units to add to the current origin's y-coordinate.

Remarks Modifies the coordinates of the viewport origin relative to the coordinates of the current viewport origin.

Return Value The previous viewport origin (in device coordinates) as a **CPoint** object.

See Also **CDC::GetViewportOrg**, **CDC::OffsetWindowOrg**, **CDC::SetViewportOrg**, **::OffsetViewportOrg**, **CPoint**

CDC::OffsetWindowOrg

CPoint OffsetWindowOrg(int *nWidth*, **int** *nHeight* **);**

nWidth Specifies the number of logical units to add to the current origin's x-coordinate.

nHeight Specifies the number of logical units to add to the current origin's y-coordinate.

Remarks Modifies the coordinates of the window origin relative to the coordinates of the current window origin.

Return Value The previous window origin (in logical coordinates) as a **CPoint** object.

See Also **CDC::GetWindowOrg**, **CDC::OffsetViewportOrg**, **CDC::SetWindowOrg**, **::OffsetWindowOrg**, **CPoint**

CDC::PaintRgn

BOOL PaintRgn(CRgn* *pRgn* **);**

pRgn Identifies the region to be filled. The coordinates for the given region are specified in device units.

Remarks Fills the region specified by *pRgn* using the current brush.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CBrush**, **CDC::SelectObject**, **CDC::FillRgn**, **::PaintRgn**, **CRgn**

CDC::PatBlt

BOOL PatBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*,
DWORD *dwRop* **);**

x Specifies the logical x-coordinate of the upper-left corner of the rectangle that is to receive the pattern.

y Specifies the logical y-coordinate of the upper-left corner of the rectangle that is to receive the pattern.

nWidth Specifies the width (in logical units) of the rectangle that is to receive the pattern.

nHeight Specifies the height (in logical units) of the rectangle that is to receive the pattern.

dwRop Specifies the raster-operation code. Raster-operation codes (ROPs) define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. This parameter may be one of the following values, with the meanings as given:

- **PATCOPY** Copies pattern to destination bitmap.
- **PATINVERT** Combines destination bitmap with pattern using the Boolean XOR operator.
- **DSTINVERT** Inverts the destination bitmap.
- **BLACKNESS** Turns all output black.

- **WHITENESS** Turns all output white.

Windows 3.1 Only

- **PATPAINT** Paints the destination bitmap. ♦

Remarks

Creates a bit pattern on the device. The pattern is a combination of the selected brush and the pattern already on the device. The raster-operation code specified by *dwRop* defines how the patterns are to be combined. The raster operations listed for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, a raster-operation code that refers to a source cannot be used.

Not all device contexts support the **PatBlt** function. To determine whether a device context supports **PatBlt**, call the **GetDeviceCaps** member function with the **RASTERCAPS** index and check the return value for the **RC_BITBLT** flag.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::GetDeviceCaps, **::PatBlt**

CDC::Pie

BOOL Pie(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3*, **int** *x4*, **int** *y4* **);**

BOOL Pie(LPCRECT *lpRect*, **POINT** *ptStart*, **POINT** *ptEnd* **);**

x1 Specifies the x-coordinate of the upper-left corner of the bounding rectangle (in logical units).

y1 Specifies the y-coordinate of the upper-left corner of the bounding rectangle (in logical units).

x2 Specifies the x-coordinate of the lower-right corner of the bounding rectangle (in logical units).

y2 Specifies the y-coordinate of the lower-right corner of the bounding rectangle (in logical units).

x3 Specifies the x-coordinate of the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

y3 Specifies the y-coordinate of the arc's starting point (in logical units). This point does not have to lie exactly on the arc.

x4 Specifies the x-coordinate of the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

y4 Specifies the y-coordinate of the arc's endpoint (in logical units). This point does not have to lie exactly on the arc.

lpRect Specifies the bounding rectangle. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

ptStart Specifies the starting point of the arc. This point does not have to lie exactly on the arc. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

ptEnd Specifies the endpoint of the arc. This point does not have to lie exactly on the arc. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by *x1*, *y1*, *x2*, and *y2* (or by *lpRect*). The starting and ending points of the arc are specified by *x3*, *y3*, *x4*, and *y4* (or by *ptStart* and *ptEnd*). The arc is drawn with the selected pen, moving in a counterclockwise direction. Two additional lines are drawn from each endpoint to the arc's center. The pie-shaped area is filled with the current brush. If *x3* equals *x4* and *y3* equals *y4*, the result is an ellipse with a single line from the center of the ellipse to the point (*x3*, *y3*) or (*x4*, *y4*). The figure drawn by this function extends up to but does not include the right and bottom coordinates. This means that the height of the figure is *y2* – *y1* and the width of the figure is *x2* – *x1*. Both the width and the height of the bounding rectangle must be greater than 2 units and less than 32,767 units.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::Chord, **::Pie**, **RECT**, **POINT**, **CRect**, **CPoint**

CDC::PlayMetaFile

BOOL PlayMetaFile(HMETAFILE *hMF* **);**

hMF Identifies the metafile to be played.

Remarks

Plays the contents of the specified metafile on the device context. The metafile can be played any number of times.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

::PlayMetaFile

CDC::Polygon

BOOL Polygon(LPPOINT *lpPoints*, **int** *nCount* **);**

lpPoints Points to an array of points that specify the vertices of the polygon. Each point in the array is a **POINT** structure or a **CPoint** object.

nCount Specifies the number of vertices in the array.

Remarks

Draws a polygon consisting of two or more points (vertices) connected by lines, using the current pen. The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first. The current polygon-filling mode can be retrieved or set by using the **GetPolyFillMode** and **SetPolyFillMode** member functions.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::GetPolyFillMode, **CDC::PolyLine**, **CDC::PolyPolygon**, **CDC::SetPolyFillMode**, **::Polygon**, **CPoint**

CDC::Polyline

BOOL Polyline(LPPOINT *lpPoints*, **int** *nCount* **);**

lpPoints Points to an array of **POINT** structures or **CPoint** objects to be connected.

nCount Specifies the number of points in the array. This value must be at least 2.

Remarks

Draws a set of line segments connecting the points specified by *lpPoints*. The lines are drawn from the first point through subsequent points using the current pen. Unlike the **LineTo** member function, the **Polyline** function neither uses nor updates the current position.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::LineTo, **CDC::Polygon**, **::PolyLine**, **POINT**, **CPoint**

CDC::PolyPolygon

BOOL PolyPolygon(LPPOINT *lpPoints*, **LPINT** *lpPolyCounts*, **int** *nCount* **);**

lpPoints Points to an array of **POINT** structures or **CPoint** objects that define the vertices of the polygons.

lpPolyCounts Points to an array of integers, each of which specifies the number of points in one of the polygons in the *lpPoints* array.

nCount The number of entries in the *lpPolyCounts* array. This number specifies the number of polygons to be drawn. This value must be at least 2.

Remarks

Creates two or more polygons that are filled using the current polygon-filling mode. The polygons may be disjoint or overlapping. Each polygon specified in a call to the **PolyPolygon** function must be closed. Unlike polygons created by the **Polygon** member function, the polygons created by **PolyPolygon** are not closed automatically.

The function creates two or more polygons. To create a single polygon, an application should use the **Polygon** member function. The current polygon-filling mode can be retrieved or set by using the **GetPolyFillMode** and **SetPolyFillMode** member functions.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::GetPolyFillMode, **CDC::Polygon**, **CDC::Polyline**, **CDC::SetPolyFillMode**, **::PolyPolygon**, **POINT**, **CPoint**

CDC::PtVisible

virtual BOOL PtVisible(int *x*, **int** *y* **) const;**

virtual BOOL PtVisible(POINT *point* **) const;**

x Specifies the logical x-coordinate of the point.

y Specifies the logical y-coordinate of the point.

point Specifies the point to check in logical coordinates. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Determines whether the given point is within the clipping region of the device context.

Return Value Nonzero if the specified point is within the clipping region; otherwise 0.

See Also **CDC::RectVisible**, **CDC::SelectClipRgn**, **CPoint**, **::PtVisible**, **POINT**

CDC::QueryAbort

Windows 3.1 Only **BOOL QueryAbort( ) const;** ♦

Remarks Calls the abort function installed by the **SetAbortProc** member function for a printing application and queries whether the printing should be terminated.

Return Value The return value is **TRUE** if printing should continue or if there is no abort procedure. It is **FALSE** if the print job should be terminated. The return value is supplied by the abort function.

See Also **CDC::SetAbortProc**, **::QueryAbort**

CDC::RealizePalette

UINT RealizePalette( );

Remarks Maps entries from the current logical palette to the system palette. A logical color palette acts as a buffer between color-intensive applications and the system, allowing an application to use as many colors as needed without interfering with its own displayed colors or with colors displayed by other windows. When a window has the input focus and calls **RealizePalette**, Windows ensures that the window will display all the requested colors, up to the maximum number simultaneously available on the screen. Windows also displays colors not found in the window's palette by matching them to available colors. In addition, Windows matches the colors requested by inactive windows that call the function as closely as possible to the available colors. This significantly reduces undesirable changes in the colors displayed in inactive windows.

Return Value Indicates how many entries in the logical palette were mapped to different entries in the system palette. This represents the number of entries that this function remapped to accommodate changes in the system palette since the logical palette was last realized.

See Also **CDC::SelectPalette**, **CPalette**, **::RealizePalette**

CDC::Rectangle

BOOL Rectangle(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

BOOL Rectangle(LPCRECT *lpRect* **);**

x1 Specifies the x-coordinate of the upper-left corner of the rectangle (in logical units).

y1 Specifies the y-coordinate of the upper-left corner of the rectangle (in logical units).

x2 Specifies the x-coordinate of the lower-right corner of the rectangle (in logical units).

y2 Specifies the y-coordinate of the lower-right corner of the rectangle (in logical units).

lpRect Specifies the rectangle in logical units. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

Remarks

Draws a rectangle using the current pen. The interior of the rectangle is filled using the current brush. The rectangle extends up to, but does not include, the right and bottom coordinates. This means that the height of the rectangle is $y2 - y1$ and the width of the rectangle is $x2 - x1$. Both the width and the height of a rectangle must be greater than 2 units and less than 32,767 units.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

::Rectangle, **CDC::PolyLine**, **CDC::RoundRect**, **RECT**, **CRect**

CDC::RectVisible

virtual BOOL RectVisible(LPCRECT *lpRect* **) const;**

lpRect Points to a **RECT** structure or a **CRect** object that contains the logical coordinates of the specified rectangle.

Remarks

Determines whether any part of the given rectangle lies within the clipping region of the display context.

Return Value Nonzero if some portion of the given rectangle lies within the clipping region; otherwise 0.

See Also **CDC::PtVisible**, **CDC::SelectClipRgn**, **CRect**, **::RectVisible**, **RECT**

CDC::ReleaseAttribDC

virtual void ReleaseAttribDC( );

Remarks Call this member function to set **m_hAttribDC** to **NULL**. This does not cause a **Detach** to occur. Only the output device context is attached to the **CDC** object, and only it can be detached.

See Also **CDC::SetOutputDC**, **CDC::SetAttribDC**, **CDC::ReleaseOutputDC**, **CDC::m_hAttribDC**

CDC::ReleaseOutputDC

virtual void ReleaseOutputDC( );

Remarks Call this member function to set the **m_hDC** member to **NULL**. This member function cannot be called when the output device context is attached to the **CDC** object. Use the **Detach** member function to detach the output device context.

See Also **CDC::SetAttribDC**, **CDC::SetOutputDC**, **CDC::ReleaseAttribDC**, **CDC::m_hDC**

CDC::ResetDC

Windows 3.1 Only **BOOL ResetDC(const DEVMODE FAR*** *lpDevMode* **);** ♦

lpDevMode A pointer to a Windows **DEVMODE** structure.

Remarks Call this member function to update the device context wrapped by the **CDC** object. The device context is updated from the information specified in the Windows **DEVMODE** structure. This member function only resets the attribute device context.

An application will typically use the **ResetDC** member function when a window handles a **WM_DEVMODECHANGE** message. You can also use this member function to change the paper orientation or paper bins while printing a document.

You cannot use this member function to change the driver name, device name or the output port. When the user changes the port connection or device name, you must delete the original device context and create a new device context with the new information.

Before you call this member function, you must ensure that all objects (other than stock objects) that had been selected into the device context have been selected out.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CDC::m_hAttribDC**, **::ResetDC**, **WM_DEVMODECHANGE**, **DEVMODE**

CDC::RestoreDC

virtual BOOL RestoreDC(int *nSavedDC* **);**

nSavedDC Specifies the device context to be restored. It can be a value returned by a previous **SaveDC** function call. If *nSavedDC* is –1, the most recently saved device context is restored.

Remarks Restores the device context to the previous state identified by *nSavedDC*. **RestoreDC** restores the device context by popping state information off a stack created by earlier calls to the **SaveDC** member function. The stack can contain the state information for several device contexts. If the context specified by *nSavedDC* is not at the top of the stack, **RestoreDC** deletes all state information between the device context specified by *nSavedDC* and the top of the stack. The deleted information is lost.

Return Value Nonzero if the specified context was restored; otherwise 0.

See Also **CDC::SaveDC**, **::RestoreDC**

CDC::RoundRect

BOOL RoundRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3* **);**

BOOL RoundRect(LPCRECT *lpRect*, **POINT** *point* **);**

x1 Specifies the x-coordinate of the upper-left corner of the rectangle (in logical units).

y1 Specifies the y-coordinate of the upper-left corner of the rectangle (in logical units).

x2 Specifies the x-coordinate of the lower-right corner of the rectangle (in logical units).

y2 Specifies the y-coordinate of the lower-right corner of the rectangle (in logical units).

x3 Specifies the width of the ellipse used to draw the rounded corners (in logical units).

y3 Specifies the height of the ellipse used to draw the rounded corners (in logical units).

lpRect Specifies the bounding rectangle in logical units. You can pass either a **CRect** object or a pointer to a **RECT** structure for this parameter.

point The x-coordinate of *point* specifies the width of the ellipse to draw the rounded corners (in logical units). The y-coordinate of *point* specifies the height of the ellipse to draw the rounded corners (in logical units). You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Draws a rectangle with rounded corners using the current pen. The interior of the rectangle is filled using the current brush. The figure this function draws extends up to but does not include the right and bottom coordinates. This means that the height of the figure is $y2 - y1$ and the width of the figure is $x2 - x1$. Both the height and the width of the bounding rectangle must be greater than 2 units and less than 32,767 units.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::Rectangle, ::RoundRect, CRect, RECT, POINT, CPoint

CDC::SaveDC

virtual int SaveDC( );

Remarks

Saves the current state of the device context by copying state information (such as clipping region, selected objects, and mapping mode) to a context stack maintained by Windows. The saved device context can later be restored by using **RestoreDC**.

SaveDC can be used any number of times to save any number of device-context states.

Return Value An integer identifying the saved device context. It is 0 if an error occurs. This return value can be used to restore the device context by calling **RestoreDC**.

See Also **CDC::RestoreDC**, **::SaveDC**

CDC::ScaleViewportExt

virtual CSize ScaleViewportExt(int *xNum*, **int** *xDenom*, **int** *yNum*, **int** *yDenom* **);**

xNum Specifies the amount by which to multiply the current x-extent.

xDenom Specifies the amount by which to divide the result of multiplying the current x-extent by the value of the *xNum* parameter.

yNum Specifies the amount by which to multiply the current y-extent.

yDenom Specifies the amount by which to divide the result of multiplying the current y-extent by the value of the *yNum* parameter.

Remarks Modifies the viewport extents relative to the current values. The formulas are written as follows:

```
xNewVE = ( xOldVE * xNum ) / xDenom
yNewVE = ( yOldVE * yNum ) / yDenom
```

The new viewport extents are calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Return Value The previous viewport extents (in device units) as a **CSize** object.

See Also **CDC::GetViewportExt**, **::ScaleViewportExt**, **CSize**

CDC::ScaleWindowExt

virtual CSize ScaleWindowExt(int *xNum*, **int** *xDenom*, **int** *yNum*, **int** *yDenom* **);**

xNum Specifies the amount by which to multiply the current x-extent.

xDenom Specifies the amount by which to divide the result of multiplying the current x-extent by the value of the *xNum* parameter.

yNum Specifies the amount by which to multiply the current y-extent.

yDenom Specifies the amount by which to divide the result of multiplying the current y-extent by the value of the *yNum* parameter.

Remarks

Modifies the window extents relative to the current values. The formulas are written as follows:

```
xNewWE = ( xOldWE * xNum ) / xDenom
yNewWE = ( yOldWE * yNum ) / yDenom
```

The new window extents are calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Return Value

The previous window extents (in logical units) as a **CSize** object.

See Also

CDC::GetWindowExt, **::ScaleWindowExt**, **CSize**

CDC::ScrollDC

BOOL ScrollDC(int *dx*, **int** *dy*, **LPCRECT** *lpRectScroll*, **LPCRECT** *lpRectClip*, **CRgn*** *pRgnUpdate*, **LPRECT** *lpRectUpdate* **);**

dx Specifies the number of horizontal scroll units.

dy Specifies the number of vertical scroll units.

lpRectScroll Points to the **RECT** structure or **CRect** object that contains the coordinates of the scrolling rectangle.

lpRectClip Points to the **RECT** structure or **CRect** object that contains the coordinates of the clipping rectangle. When this rectangle is smaller than the original one pointed to by *lpRectScroll*, scrolling occurs only in the smaller rectangle.

pRgnUpdate Identifies the region uncovered by the scrolling process. The **ScrollDC** function defines this region; it is not necessarily a rectangle.

lpRectUpdate Points to the **RECT** structure or **CRect** object that receives the coordinates of the rectangle that bounds the scrolling update region. This is the largest rectangular area that requires repainting. The values in the structure or object when the function returns are in client coordinates, regardless of the mapping mode for the given device context.

Remarks

Scrolls a rectangle of bits horizontally and vertically. If *lpRectUpdate* is **NULL**, Windows does not compute the update rectangle. If both *pRgnUpdate* and *lpRectUpdate* are **NULL**, Windows does not compute the update region. If *pRgnUpdate* is not **NULL**, Windows assumes that it contains a valid pointer to the region uncovered by the scrolling process (defined by the **ScrollDC** member function). The update region returned in *lpRectUpdate* can be passed to **CWnd::InvalidateRgn** if required.

An application should use the **ScrollWindow** member function of class **CWnd** when it is necessary to scroll the entire client area of a window. Otherwise, it should use **ScrollDC**.

Return Value

Nonzero if scrolling is executed; otherwise 0.

See Also

CWnd::InvalidateRgn, **CWnd::ScrollWindow**, **::ScrollDC**, **CRgn**, **RECT**, **CRect**

CDC::SelectClipRgn

virtual int SelectClipRgn(CRgn* *pRgn* **);**

pRgn Identifies the region to be selected. If this value is **NULL**, the entire client area is selected and output is still clipped to the window.

Remarks

Selects the given region as the current clipping region for the device context. Only a copy of the selected region is used. The region itself can be selected for any number of other device contexts, or it can be deleted.

The function assumes that the coordinates for the given region are specified in device units. Some printer devices support text output at a higher resolution than graphics output in order to retain the precision needed to express text metrics. These devices report device units at the higher resolution, that is, in text units. These devices then scale coordinates for graphics so that several reported device units map to only 1 graphic unit. You should always call the **SelectClipRgn** function using text units.

Applications that must take the scaling of graphics objects in the GDI can use the **GETSCALINGFACTOR** printer escape to determine the scaling factor. This scaling factor affects clipping. If a region is used to clip graphics, GDI divides the coordinates by the scaling factor. If the region is used to clip text, GDI makes no scaling adjustment. A scaling factor of 1 causes the coordinates to be divided by 2; a scaling factor of 2 causes the coordinates to be divided by 4; and so on.

Return Value The region's type. It can be any one of the following values, with the meanings as given:

- **COMPLEXREGION** New clipping region has overlapping borders.
- **ERROR** Device context or region is not valid.
- **NULLREGION** New clipping region is empty.
- **SIMPLEREGION** New clipping region has no overlapping borders.

See Also **CDC::GetClipBox**, **CDC::Escape**, **CRgn**, **::SelectClipRgn**

CDC::SelectObject

CPen* SelectObject(CPen* *pPen* **);**

CBrush* SelectObject(CBrush* *pBrush* **);**

virtual CFont* SelectObject(CFont* *pFont* **);**

CBitmap* SelectObject(CBitmap* *pBitmap* **);**

int SelectObject(CRgn* *pRgn* **);**

pPen A pointer to a **CPen** object to be selected.

pBrush A pointer to a **CBrush** object to be selected.

pFont A pointer to a **CFont** object to be selected.

pBitmap A pointer to a **CBitmap** object to be selected.

pRgn A pointer to a **CRgn** object to be selected.

Remarks Selects an object into the device context. Class **CDC** provides five versions specialized for particular kinds of GDI objects, including pens, brushes, fonts, bitmaps, and regions. The newly selected object replaces the previous object of the same type. For example, if *pObject* of the general version of **SelectObject** points to a **CPen** object, the function replaces the current pen with the pen specified by *pObject*.

An application can select a bitmap into memory device contexts only and into only one memory device context at a time. The format of the bitmap must either be monochrome or compatible with the device context; if it is not, **SelectObject** returns an error.

Windows 3.1 Only For Windows 3.1, the **SelectObject** function returns the same value whether or not it is used in a metafile. Under previous versions of Windows, **SelectObject** returned a nonzero value for success and 0 for failure when it was used in a metafile. ♦

Return Value A pointer to the object being replaced. This is a pointer to an object of one of the classes derived from **CGdiObject**, such as **CPen**, depending on which version of the function is used. The return value is **NULL** if there is an error.

The version of the member function that takes a region parameter performs the same task as the **SelectClipRgn** member function. Its return value can be any one of the following, with the meanings as given:

- **COMPLEXREGION** New clipping region has overlapping borders.
- **ERROR** Device context or region is not valid.
- **NULLREGION** New clipping region is empty.
- **SIMPLEREGION** New clipping region has no overlapping borders.

See Also **CGdiObject::DeleteObject**, **CDC::SelectClipRgn**, **CDC::SelectPalette**, **::SelectObject**

CDC::SelectPalette

CPalette* SelectPalette(CPalette* *pPalette*, **BOOL** *bForceBackground* **);**

pPalette Identifies the logical palette to be selected. This palette must already have been created with the **CPalette** member function **CreatePalette**.

bForceBackground Specifies whether the logical palette is forced to be a background palette. If *bForceBackground* is nonzero, the selected palette is always a background palette, regardless of whether the window has the input focus. If *bForceBackground* is 0 and the device context is attached to a window, the logical palette is a foreground palette when the window has the input focus.

Remarks Selects the logical palette that is specified by *pPalette* as the selected palette object of the device context. The new palette becomes the palette object used by GDI to control colors displayed in the device context and replaces the previous palette. An application can select a logical palette into more than one device context. However, changes to a logical palette will affect all device contexts for which it is selected. If an application selects a palette into more than one device context, the device contexts must all belong to the same physical device.

Return Value A pointer to a **CPalette** object identifying the logical palette replaced by the palette specified by *pPalette*. It is **NULL** if there is an error.

See Also **CDC::RealizePalette**, **CPalette**, **::SelectPalette**

CDC::SelectStockObject

virtual CGdiObject* SelectStockObject(int *nIndex* **);**

nIndex Specifies the kind of stock object desired. It can be one of the following values, with meanings as given:

- **BLACK_BRUSH** Black brush.
- **DKGRAY_BRUSH** Dark gray brush.
- **GRAY_BRUSH** Gray brush.
- **HOLLOW_BRUSH** Hollow brush.
- **LTGRAY_BRUSH** Light gray brush.
- **NULL_BRUSH** Null brush.
- **WHITE_BRUSH** White brush.
- **BLACK_PEN** Black pen.
- **NULL_PEN** Null pen.
- **WHITE_PEN** White pen.
- **ANSI_FIXED_FONT** ANSI fixed system font.
- **ANSI_VAR_FONT** ANSI variable system font.
- **DEVICE_DEFAULT_FONT** Device-dependent font.
- **OEM_FIXED_FONT** OEM-dependent fixed font.
- **SYSTEM_FONT** The system font. By default, Windows uses the system font to draw menus, dialog-box controls, and other text. In Windows versions 3.0 and later, the system font is proportional width; earlier versions of Windows use a fixed-width system font.

- **SYSTEM_FIXED_FONT** The fixed-width system font used in Windows prior to version 3.0. This object is available for compatibility with earlier versions of Windows.
- **DEFAULT_PALETTE** Default color palette. This palette consists of the 20 static colors in the system palette.

Remarks Selects a **CGdiObject** object that corresponds to one of the predefined stock pens, brushes, or fonts.

Return Value A pointer to the **CGdiObject** object that was replaced if the function is successful. The actual object pointed to is a **CPen**, **CBrush**, or **CFont** object. If the call is unsuccessful, the return value is **NULL**.

See Also **CGdiObject::GetObject**

CDC::SetAbortProc

int SetAbortProc(BOOL (CALLBACK EXPORT* *lpfn* **)(HDC, int));**

lpfn A pointer to the abort function to install as the abort procedure. For more about this callback function, see the "Callback Function" section below.

Remarks Installs the abort procedure for the print job. If an application is to allow the print job to be canceled during spooling, it must set the abort function before the print job is started with the **StartDoc** member function. The Print Manager calls the abort function during spooling to allow the application to cancel the print job or to process out-of-disk-space conditions. If no abort function is set, the print job will fail if there is not enough disk space for spooling.

Note that the features of Microsoft Visual C++ simplify the creation of the callback function passed to **SetAbortProc**. The address passed to the **EnumObjects** member function is a **FAR** pointer to a function exported with **__export** and with the Pascal calling convention. In protect-mode applications, you do not have to create this function with the Windows **MakeProcInstance** function or free the function after use with the Windows function **FreeProcInstance**.

You also do not have to export the function name in an **EXPORTS** statement in your application's module-definition file. You can instead use the **__export** function modifier, as in

BOOL CALLBACK __export AFunction(**HDC**, **int**);

to cause the compiler to emit the proper export record for export by name without aliasing. This works for most needs. For some special cases, such as exporting a function by ordinal or aliasing the export, you still need to use an **EXPORTS** statement in a module-definition file.

For compiling Microsoft Foundation programs, you'll normally use the /GA and /GEs compiler options. The /Gw compiler option is not used with the Microsoft Foundation classes. (If you do use the Windows function **MakeProcInstance**, you will need to explicitly cast the returned function pointer from **FARPROC** to the type needed by this member function.) Callback registration interfaces are now type-safe (you must pass in a function pointer that points to the right kind of function for the specific callback).

Also note that all callback functions must trap Microsoft Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see Chapter 16 in the *Class Library User's Guide*.

Callback Function

The callback function must use the Pascal calling convention, must be exported with **__export**, and must be declared **FAR**.

BOOL FAR PASCAL __export AbortFunc(**HDC** *hPr*, **int** *code*);

The name AbortFunc is a placeholder for the application-supplied function name. The actual name must be exported as described in the "Remarks" section above. The parameters are described below:

- *hPr* Identifies the device context.
- *code* Specifies whether an error has occurred. It is 0 if no error has occurred. It is **SP_OUTOFDISK** if the Print Manager is currently out of disk space and more disk space will become available if the application waits. If *code* is **SP_OUTOFDISK**, the application does not have to abort the print job. If it does not, it must yield to the Print Manager by calling the **PeekMessage** or **GetMessage** Windows function.

Return Value

The return value of the abort-handler function is nonzero if the print job is to continue, and 0 if it is canceled.

Return Value

Specifies the outcome of the **SetAbortProc** function. Some of the following values are more probable than others, but all are possible.

- **SP_ERROR** General error.
- **SP_OUTOFDISK** Not enough disk space is currently available for spooling, and no more space will become available.

- **SP_OUTOFMEMORY** Not enough memory is available for spooling.
- **SP_USERABORT** User ended the job through the Print Manager.

CDC::SetAttribDC

virtual void SetAttribDC(HDC *hDC* **);**

hDC A Windows device context.

Remarks Call this function to set the attribute device context, **m_hAttribDC**. This member function does not attach the device context to the **CDC** object. Only the output device context is attached to a **CDC** object.

See Also **CDC::SetOutputDC**, **CDC::ReleaseAttribDC**, **CDC::ReleaseOutputDC**

CDC::SetBkColor

virtual COLORREF SetBkColor(COLORREF *crColor* **);**

crColor Specifies the new background color.

Remarks Sets the current background color to the specified color. If the background mode is **OPAQUE**, the system uses the background color to fill the gaps in styled lines, the gaps between hatched lines in brushes, and the background in character cells. The system also uses the background color when converting bitmaps between color and monochrome device contexts. If the device cannot display the specified color, the system sets the background color to the nearest physical color.

Return Value The previous background color as an RGB color value. If an error occurs, the return value is 0x80000000.

See Also **CDC::BitBlt**, **CDC::GetBkColor**, **CDC::GetBkMode**, **CDC::SetBkMode**, **CDC::StretchBlt**, **::SetBkColor**

CDC::SetBkMode

int SetBkMode(int *nBkMode* **);**

nBkMode Specifies the mode to be set. This parameter can be either of the following values, with the meanings as given:

- **OPAQUE** Background is filled with the current background color before the text, hatched brush, or pen is drawn. This is the default background mode.
- **TRANSPARENT** Background is not changed before drawing.

Remarks

Sets the background mode. The background mode defines whether the system removes existing background colors on the drawing surface before drawing text, hatched brushes, or any pen style that is not a solid line.

Return Value

The previous background mode.

See Also

CDC::GetBkColor, **CDC::GetBkMode**, **CDC::SetBkColor**, **::SetBkMode**

CDC::SetBoundsRect

Windows 3.1 Only

UINT SetBoundsRect(LPCRECT *lpRectBounds*, **UINT** *flags* **);** ♦

lpRectBounds Points to a **RECT** structure or **CRect** object that is used to set the bounding rectangle. Rectangle dimensions are given in logical coordinates. This parameter can be **NULL**.

flags Specifies how the new rectangle will be combined with the accumulated rectangle. This parameter may be a combination of the following values:

- **DCB_ACCUMULATE** Add the rectangle specified by *lpRectBounds* to the bounding rectangle (using a rectangle-union operation).
- **DCB_DISABLE** Turn off bounds accumulation.
- **DCB_ENABLE** Turn on bounds accumulation. (The default setting for bounds accumulation is disabled.)

Remarks

Controls the accumulation of bounding-rectangle information for the specified device context. Windows can maintain a bounding rectangle for all drawing operations. This rectangle can be queried and reset by the application. The drawing bounds are useful for invalidating bitmap caches.

Return Value

The current state of the bounding rectangle, if the function is successful. Like *flags*, the return value can be a combination of **DCB_** values, as shown in the following list:

- **DCB_ACCUMULATE** The bounding rectangle is not empty. This value will always be set.

- **DCB_DISABLE** Bounds accumulation is off.
- **DCB_ENABLE** Bounds accumulation is on.

See Also **CDC::GetBoundsRect**, **::SetBoundsRect**, **RECT**, **CRect**

CDC::SetBrushOrg

CPoint SetBrushOrg(int *x*, **int** *y* **);**

CPoint SetBrushOrg(POINT *point* **);**

x Specifies the x-coordinate (in device units) of the new origin. This value must be in the range 0–7.

y Specifies the y-coordinate (in device units) of the new origin. This value must be in the range 0–7.

point Specifies the x- and y-coordinates of the new origin. Each value must be in the range 0–7. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks Specifies the origin that GDI will assign to the next brush that the application selects into the device context. The default coordinates for the brush origin are (0, 0). To alter the origin of a brush, call the **UnrealizeObject** function for the **CBrush** object, call **SetBrushOrg**, and then call the **SelectObject** member function to select the brush into the device context. Do not use **SetBrushOrg** with stock **CBrush** objects.

Return Value The previous origin of the brush in device units.

See Also **CBrush**, **CDC::GetBrushOrg**, **CDC::SelectObject**, **CGdiObject::UnrealizeObject**, **::SetBrushOrg**, **POINT**, **CPoint**

CDC::SetMapMode

virtual int SetMapMode(int *nMapMode* **);**

nMapMode Specifies the new mapping mode. It can be any one of the following values, with the meanings as given:

- **MM_ANISOTROPIC** Logical units are converted to arbitrary units with arbitrarily scaled axes. Setting the mapping mode to **MM_ANISOTROPIC** does not change the current window or viewport settings. To change the units, orientation, and scaling, call the **SetWindowExt** and **SetViewportExt** member functions.
- **MM_HIENGLISH** Each logical unit is converted to 0.001 inch. Positive x is to the right; positive y is up.
- **MM_HIMETRIC** Each logical unit is converted to 0.01 millimeter. Positive x is to the right; positive y is up.
- **MM_ISOTROPIC** Logical units are converted to arbitrary units with equally scaled axes; that is, 1 unit along the x-axis is equal to 1 unit along the y-axis. Use the **SetWindowExt** and **SetViewportExt** member functions to specify the desired units and the orientation of the axes. GDI makes adjustments as necessary to ensure that the x and y units remain the same size.
- **MM_LOENGLISH** Each logical unit is converted to 0.01 inch. Positive x is to the right; positive y is up.
- **MM_LOMETRIC** Each logical unit is converted to 0.1 millimeter. Positive x is to the right; positive y is up.
- **MM_TEXT** Each logical unit is converted to 1 device pixel. Positive x is to the right; positive y is down.
- **MM_TWIPS** Each logical unit is converted to 1/20 of a point. (Because a point is 1/72 inch, a twip is 1/1440 inch.) Positive x is to the right; positive y is up.

Remarks

Sets the mapping mode. The mapping mode defines the unit of measure used to convert logical units to device units; it also defines the orientation of the device's x- and y-axes. GDI uses the mapping mode to convert logical coordinates into the appropriate device coordinates. The **MM_TEXT** mode allows applications to work in device pixels, where 1 unit is equal to 1 pixel. The physical size of a pixel varies from device to device. The **MM_HIENGLISH**, **MM_HIMETRIC**, **MM_LOENGLISH**, **MM_LOMETRIC**, and **MM_TWIPS** modes are useful for applications that must draw in physically meaningful units (such as inches or millimeters). The **MM_ISOTROPIC** mode ensures a 1:1 aspect ratio, which is useful when it is important to preserve the exact shape of an image. The **MM_ANISOTROPIC** mode allows the x- and y-coordinates to be adjusted independently.

Return Value

The previous mapping mode.

See Also

CDC::SetViewportExt, **CDC::SetWindowExt**, **::SetMapMode**

CDC::SetMapperFlags

DWORD SetMapperFlags(DWORD *dwFlag* **);**

dwFlag Specifies whether the font mapper attempts to match a font's aspect height and width to the device. When this value is **ASPECT_FILTERING**, the mapper selects only fonts whose x-aspect and y-aspect exactly match those of the specified device.

Remarks

Changes the method used by the font mapper when it converts a logical font to a physical font. An application can use **SetMapperFlags** to cause the font mapper to attempt to choose only a physical font that exactly matches the aspect ratio of the specified device. An application that uses only raster fonts can use the **SetMapperFlags** function to ensure that the font selected by the font mapper is attractive and readable on the specified device. Applications that use scalable (TrueType) fonts typically do not use **SetMapperFlags**. If no physical font has an aspect ratio that matches the specification in the logical font, GDI chooses a new aspect ratio and selects a font that matches this new aspect ratio.

Return Value

The previous value of the font-mapper flag.

See Also

::SetMapperFlags

CDC::SetOutputDC

virtual void SetOutputDC(HDC *hDC* **);**

hDC A Windows device context.

Remarks

Call this member function to set the output device context, **m_hDC**. This member function can only be called when a device context has not been attached to the **CDC** object. This member function sets **m_hDC** but does not attach the device context to the **CDC** object.

See Also

CDC::SetAttribDC, **CDC::ReleaseAttribDC**, **CDC::ReleaseOutputDC**, **CDC::m_hDC**

CDC::SetPixel

COLORREF SetPixel(int *x*, **int** *y*, **COLORREF** *crColor* **);**

COLORREF SetPixel(POINT *point*, **COLORREF** *crColor* **);**

x Specifies the logical x-coordinate of the point to be set.

y Specifies the logical y-coordinate of the point to be set.

crColor Specifies the color used to paint the point.

point Specifies the logical x- and y-coordinates of the point to be set. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Sets the pixel at the point specified to the closest approximation of the color specified by *crColor*. The point must be in the clipping region. If the point is not in the clipping region, the function does nothing. Not all devices support the **SetPixel** function. To determine whether a device supports **SetPixel**, call the **GetDeviceCaps** member function with the **RASTERCAPS** index and check the return value for the **RC_BITBLT** flag.

Return Value

An RGB value for the color that the point is actually painted. This value can be different from that specified by *crColor* if an approximation of that color is used. If the function fails (if the point is outside the clipping region), the return value is –1.

See Also

CDC::GetDeviceCaps, **CDC::GetPixel**, **::SetPixel**, **POINT**, **CPoint**

CDC::SetPolyFillMode

int SetPolyFillMode(int *nPolyFillMode* **);**

nPolyFillMode Specifies the new filling mode. This value may be either **ALTERNATE** or **WINDING**. The default mode set in Windows is **ALTERNATE**.

Remarks

Sets the polygon-filling mode. When the polygon-filling mode is **ALTERNATE**, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on. This mode is the default. When the polygon-filling mode is **WINDING**, the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through a counterclockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

Return Value The previous filling mode, if successful; otherwise 0.

See Also **CDC::GetPolyFillMode**, **CDC::PolyPolygon**, **::SetPolyFillMode**

CDC::SetROP2

int SetROP2(int *nDrawMode* **);**

nDrawMode Specifies the new drawing mode. It can be any one of the following values, with the meanings as given:

- **R2_BLACK** Pixel is always black.
- **R2_WHITE** Pixel is always white.
- **R2_NOP** Pixel remains unchanged.
- **R2_NOT** Pixel is the inverse of the screen color.
- **R2_COPYPEN** Pixel is the pen color.
- **R2_NOTCOPYPEN** Pixel is the inverse of the pen color.
- **R2_MERGEPENNOT** Pixel is a combination of the pen color and the inverse of the screen color (final pixel = (NOT screen pixel) OR pen).
- **R2_MASKPENNOT** Pixel is a combination of the colors common to both the pen and the inverse of the screen (final pixel = (NOT screen pixel) AND pen).
- **R2_MERGENOTPEN** Pixel is a combination of the screen color and the inverse of the pen color (final pixel = (NOT pen) OR screen pixel).
- **R2_MASKNOTPEN** Pixel is a combination of the colors common to both the screen and the inverse of the pen (final pixel = (NOT pen) AND screen pixel).
- **R2_MERGEPEN** Pixel is a combination of the pen color and the screen color (final pixel = pen OR screen pixel).
- **R2_NOTMERGEPEN** Pixel is the inverse of the **R2_MERGEPEN** color (final pixel = NOT(pen OR screen pixel)).
- **R2_MASKPEN** Pixel is a combination of the colors common to both the pen and the screen (final pixel = pen AND screen pixel).

- **R2_NOTMASKPEN** Pixel is the inverse of the **R2_MASKPEN** color (final pixel = NOT(pen AND screen pixel)).
- **R2_XORPEN** Pixel is a combination of the colors that are in the pen or in the screen, but not in both (final pixel = pen XOR screen pixel).
- **R2_NOTXORPEN** Pixel is the inverse of the **R2_XORPEN** color (final pixel = NOT(pen XOR screen pixel)).

Remarks Sets the current drawing mode. The drawing mode specifies how the colors of the pen and the interior of filled objects are combined with the color already on the display surface. The drawing mode is for raster devices only; it does not apply to vector devices. Drawing modes are binary raster-operation codes representing all possible Boolean combinations of two variables, using the binary operators AND, OR, and XOR (exclusive OR), and the unary operation NOT.

Return Value The previous drawing mode. It can be any one of the values given in the Windows SDK documentation.

See Also **CDC::GetDeviceCaps**, **CDC::GetROP2**, **::SetROP2**

CDC::SetStretchBltMode

int SetStretchBltMode(int *nStretchMode* **);**

nStretchMode Specifies the new bitmap-stretching mode. It can be one of the following values, with the meaning as given:

- **STRETCH_ANDSCANS** Uses the AND operator to combine eliminated lines with the remaining lines. This mode preserves black pixels at the expense of colored or white pixels.
- **STRETCH_DELETESCANS** Deletes the eliminated lines. Information in the eliminated lines is not preserved.
- **STRETCH_ORSCANS** Uses the OR operator to combine eliminated lines with the remaining lines. This mode preserves colored or white pixels at the expense of black pixels.

Remarks Sets the bitmap-stretching mode for the **StretchBlt** member function. The bitmap-stretching mode defines how information is removed from bitmaps that are compressed by using the function. The default mode is **STRETCH_ANDSCANS**. The **STRETCH_ANDSCANS** and **STRETCH_ORSCANS** modes are typically used to preserve foreground pixels in monochrome bitmaps. The **STRETCH_DELETESCANS** mode is typically used to preserve color in color bitmaps.

Return Value The previous stretching mode. It can be **STRETCH_ANDSCANS**, **STRETCH_DELETESCANS**, or **STRETCH_ORSCRANS**.

See Also **CDC::GetStretchBltMode**, **CDC::StretchBlt**, **::SetStretchBltMode**

CDC::SetTextAlign

UINT SetTextAlign(UINT *nFlags* **);**

nFlags Specifies text-alignment flags. The flags specify the relationship between a point and a rectangle that bounds the text. The point can be either the current position or coordinates specified by a text-output function. The rectangle that bounds the text is defined by the adjacent character cells in the text string. The *nFlags* parameter can be one or more flags from the following three categories. Choose only one flag from each category. The first category affects text alignment in the x-direction:

- **TA_CENTER** Aligns the point with the horizontal center of the bounding rectangle.
- **TA_LEFT** Aligns the point with the left side of the bounding rectangle. This is the default setting.
- **TA_RIGHT** Aligns the point with the right side of the bounding rectangle.

The second category affects text alignment in the y-direction:

- **TA_BASELINE** Aligns the point with the baseline of the chosen font.
- **TA_BOTTOM** Aligns the point with the bottom of the bounding rectangle.
- **TA_TOP** Aligns the point with the top of the bounding rectangle. This is the default setting.

The third category determines whether the current position is updated when text is written:

- **TA_NOUPDATECP** Does not update the current position after each call to a text-output function. This is the default setting.
- **TA_UPDATECP** Updates the current x-position after each call to a text-output function. The new position is at the right side of the bounding rectangle for the text. When this flag is set, the coordinates specified in calls to the **TextOut** member function are ignored.

Remarks Sets the text-alignment flags. The **TextOut** and **ExtTextOut** member functions use these flags when positioning a string of text on a display or device. The flags specify the relationship between a specific point and a rectangle that bounds the text. The coordinates of this point are passed as parameters to the **TextOut** member function. The rectangle that bounds the text is formed by the adjacent character cells in the text string.

Return Value The previous text-alignment setting, if successful. The low-order byte contains the horizontal setting and the high-order byte contains the vertical setting; otherwise 0.

See Also **CDC::ExtTextOut**, **CDC::GetTextAlign**, **CDC::TabbedTextOut**, **CDC::TextOut**, **::SetTextAlign**

CDC::SetTextCharacterExtra

int SetTextCharacterExtra(int *nCharExtra* **);**

nCharExtra Specifies the amount of extra space (in logical units) to be added to each character. If the current mapping mode is not **MM_TEXT**, *nCharExtra* is transformed and rounded to the nearest pixel.

Remarks Sets the amount of intercharacter spacing. GDI adds this spacing to each character, including break characters, when it writes a line of text to the device context. The default value for the amount of intercharacter spacing is 0.

Return Value The amount of the previous intercharacter spacing.

See Also **CDC::GetTextCharacterExtra**, **::SetTextCharacterExtra**

CDC::SetTextColor

virtual COLORREF SetTextColor(COLORREF *crColor* **);**

crColor Specifies the color of the text as an RGB color value.

Remarks Sets the text color to the specified color. The system will use this text color when writing text to this device context and also when converting bitmaps between color and monochrome device contexts. If the device cannot represent the specified color, the system sets the text color to the nearest physical color. The background color for a character is specified by the **SetBkColor** and **SetBkMode** member functions.

Return Value An RGB value for the previous text color.

See Also **CDC::GetTextColor**, **CDC::BitBlt**, **CDC::SetBkColor**, **CDC::SetBkMode**, **::SetTextColor**

CDC::SetTextJustification

int SetTextJustification(**int** *nBreakExtra*, **int** *nBreakCount* **);**

nBreakExtra Specifies the total extra space to be added to the line of text (in logical units). If the current mapping mode is not **MM_TEXT**, the value given by this parameter is converted to the current mapping mode and rounded to the nearest device unit.

nBreakCount Specifies the number of break characters in the line.

Remarks Adds space to the break characters in a string. An application can use the **GetTextMetrics** member functions to retrieve a font's break character. After the **SetTextJustification** member function is called, a call to a text-output function (such as **TextOut**) distributes the specified extra space evenly among the specified number of break characters. The break character is usually the space character (ASCII 32), but may be defined by a font as some other character.

The member function **GetTextExtent** is typically used with **SetTextJustification**. **GetTextExtent** computes the width of a given line before alignment. An application can determine how much space to specify in the *nBreakExtra* parameter by subtracting the value returned by **GetTextExtent** from the width of the string after alignment.

The **SetTextJustification** function can be used to align a line that contains multiple runs in different fonts. In this case, the line must be created piecemeal by aligning and writing each run separately. Because rounding errors can occur during alignment, the system keeps a running error term that defines the current error. When aligning a line that contains multiple runs, **GetTextExtent** automatically uses this error term when it computes the extent of the next run. This allows the text-output function to blend the error into the new run. After each line has been aligned, this error term must be cleared to prevent it from being incorporated into the next line. The term can be cleared by calling **SetTextJustification** with *nBreakExtra* set to 0.

Return Value One if the function is successful; otherwise 0.

See Also **CDC::GetMapMode**, **CDC::GetTextExtent**, **CDC::GetTextMetrics**, **CDC::SetMapMode**, **CDC::TextOut**, **::SetTextJustification**

CDC::SetViewportExt

virtual CSize SetViewportExt(int *cx*, **int** *cy* **);**

virtual CSize SetViewportExt(SIZE *size* **);**

cx Specifies the x-extent of the viewport (in device units).

cy Specifies the y-extent of the viewport (in device units).

size Specifies the x- and y-extents of the viewport (in device units).

Remarks

Sets the x- and y-extents of the viewport of the device context. The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates. When the following mapping modes are set, calls to **SetWindowExt** and **SetViewportExt** are ignored:

MM_HIENGLISH	**MM_LOMETRIC**
MM_HIMETRIC	**MM_TEXT**
MM_LOENGLISH	**MM_TWIPS**

When **MM_ISOTROPIC** mode is set, an application must call the **SetWindowExt** member function before it calls **SetViewportExt**.

Return Value

The previous extents of the viewport as a **CSize** object. When an error occurs, the x- and y-coordinates of the returned **CSize** object are both set to 0.

See Also

CDC::SetWindowExt, **::SetViewportExt**, **CSize**, **CDC::GetViewportExt**

CDC::SetViewportOrg

virtual CPoint SetViewportOrg(int *x*, **int** *y* **);**

virtual CPoint SetViewportOrg(POINT *point* **);**

x Specifies the x-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

y Specifies the y-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

point Specifies the origin of the viewport. The values must be within the range of the device coordinate system. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks

Sets the viewport origin of the device context. The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates. The viewport origin marks the point in the device coordinate system to which GDI maps the window origin, a point in the logical coordinate system specified by the **SetWindowOrg** member function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Return Value

The previous origin of the viewport (in device coordinates) as a **CPoint** object.

See Also

CDC::SetWindowOrg, **::SetViewportOrg**, **CPoint**, **POINT**, **CDC::GetViewportOrg**

CDC::SetWindowExt

virtual CSize SetWindowExt(int *cx*, **int** *cy* **);**

virtual CSize SetWindowExt(SIZE *size* **);**

cx Specifies the x-extent (in logical units) of the window.

cy Specifies the y-extent (in logical units) of the window.

size Specifies the x- and y-extents (in logical units) of the window.

Remarks

Sets the x- and y-extents of the window associated with the device context. The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system. When the following mapping modes are set, calls to **SetWindowExt** and **SetViewportExt** functions are ignored:

- **MM_HIENGLISH**
- **MM_HIMETRIC**
- **MM_LOENGLISH**
- **MM_LOMETRIC**

- **MM_TEXT**
- **MM_TWIPS**

When **MM_ISOTROPIC** mode is set, an application must call the **SetWindowExt** member function before calling **SetViewportExt**.

Return Value The previous extents of the window (in logical units) as a **CSize** object. If an error occurs, the x- and y-coordinates of the returned **CSize** object are both set to 0.

See Also **CDC::GetWindowExt**, **CDC::SetViewportExt**, **::SetWindowExt**, **CSize**

CDC::SetWindowOrg

CPoint SetWindowOrg(int *x*, **int** *y* **);**

CPoint SetWindowOrg(POINT *point* **);**

x Specifies the logical x-coordinate of the new origin of the window.

y Specifies the logical y-coordinate of the new origin of the window.

point Specifies the logical coordinates of the new origin of the window. You can pass either a **POINT** structure or a **CPoint** object for this parameter.

Remarks Sets the window origin of the device context. The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system. The window origin marks the point in the logical coordinate system from which GDI maps the viewport origin, a point in the device coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Return Value The previous origin of the window as a **CPoint** object.

See Also **::SetWindowOrg**, **::SetViewportOrg**, **CPoint**, **POINT**, **CDC::GetWindowOrg**

CDC::StartDoc

int StartDoc(LPDOCINFO *lpDocInfo* **);**

lpDocInfo Points to a **DOCINFO** structure containing the name of the document file and the name of the output file.

Remarks

Informs the device driver that a new print job is starting and that all subsequent **StartPage** and **EndPage** calls should be spooled under the same job until an **EndDoc** call occurs. This ensures that documents longer than one page will not be interspersed with other jobs.

For Windows version 3.1, this function replaces the **STARTDOC** printer escape. Using this function ensures that documents containing more than one page are not interspersed with other print jobs.

When running under Windows version 3.0, this member function sends a **STARTDOC** printer escape.

StartDoc should not be used inside metafiles.

Return Value

The value –1 if there is an error such as insufficient memory or an invalid port specification occurs; otherwise a positive value.

DOCINFO Structure Windows 3.1 Only

A **DOCINFO** structure has this form:

```
typedef struct {    /* di */
    int     cbSize;
    LPCSTR  lpszDocName;
    LPCSTR  lpszOutput;
} DOCINFO;
```

The **DOCINFO** structure contains the input and output filenames used by the **StartDoc** function.

Members

cbSize Specifies the size of the structure, in bytes.

lpszDocName Points to a null-terminated string specifying the name of the document. This string must not be longer than 32 characters, including the null terminating character.

lpszOutput Points to a null-terminated string specifying the name of an output file. This allows a print job to be redirected to a file. If this value is **NULL**, output goes to the device for the specified device context. ♦

See Also

CDC::Escape, CDC::EndDoc, CDC::AbortDoc

CDC::StartPage

int StartPage( );

Remarks

Call this member function to prepare the printer driver to receive data. **StartPage** supersedes the **NEWFRAME** and **BANDINFO** escapes. For an overview of the sequence of printing calls, see the **StartDoc** member function.

The system disables the **ResetDC** member function between calls to **StartPage** and **EndPage**.

When running under Windows version 3.0, this member function does nothing.

See Also

CDC::Escape, **CDC::EndPage**

CDC::StretchBlt

BOOL StretchBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **CDC*** *pSrcDC*, **int** *xSrc*, **int** *ySrc*, **int** *nSrcWidth*, **int** *nSrcHeight*, **DWORD** *dwRop* **);**

x Specifies the x-coordinate (in logical units) of the upper-left corner of the destination rectangle.

y Specifies the y-coordinate (in logical units) of the upper-left corner of the destination rectangle.

nWidth Specifies the width (in logical units) of the destination rectangle.

nHeight Specifies the height (in logical units) of the destination rectangle.

pSrcDC Specifies the source device context.

xSrc Specifies the x-coordinate (in logical units) of the upper-left corner of the source rectangle.

ySrc Specifies the x-coordinate (in logical units) of the upper-left corner of the source rectangle.

nSrcWidth Specifies the width (in logical units) of the source rectangle.

nSrcHeight Specifies the height (in logical units) of the source rectangle.

dwRop Specifies the raster operation to be performed. Raster operation codes define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. This parameter may be one of the following values, as described below:

- **BLACKNESS** Turns all output black.
- **DSTINVERT** Inverts the destination bitmap.
- **MERGECOPY** Combines the pattern and the source bitmap using the Boolean AND operator.
- **MERGEPAINT** Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.
- **NOTSRCCOPY** Copies the inverted source bitmap to the destination.
- **NOTSRCERASE** Inverts the result of combining the destination and source bitmaps using the Boolean OR operator.
- **PATCOPY** Copies the pattern to the destination bitmap.
- **PATINVERT** Combines the destination bitmap with the pattern using the Boolean XOR operator.
- **PATPAINT** Combines the inverted source bitmap with the pattern using the Boolean OR operator. Combines the result of this operation with the destination bitmap using the Boolean OR operator.
- **SRCAND** Combines pixels of the destination and source bitmaps using the Boolean AND operator.
- **SRCCOPY** Copies the source bitmap to the destination bitmap.
- **SRCERASE** Inverts the destination bitmap and combines the result with the source bitmap using the Boolean AND operator.
- **SRCINVERT** Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
- **SRCPAINT** Combines pixels of the destination and source bitmaps using the Boolean OR operator.
- **WHITENESS** Turns all output white.

Remarks

Copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle. The function uses the stretching mode of the destination device context (set by **SetStretchBltMode**) to determine how to stretch or compress the bitmap.

The **StretchBlt** function moves the bitmap from the source device given by *pSrcDC* to the destination device represented by the device-context object whose member function is being called. The *xSrc*, *ySrc*, *nSrcWidth*, and *nSrcHeight* parameters define the upper-left corner and dimensions of the source rectangle. The *x*, *y*, *nWidth*, and *nHeight* parameters give the upper-left corner and dimensions of the destination rectangle. The raster operation specified by *dwRop* defines how the source bitmap and the bits already on the destination device are combined.

The **StretchBlt** function creates a mirror image of a bitmap if the signs of the *nSrcWidth* and *nWidth* or *nSrcHeight* and *nHeight* parameters differ. If *nSrcWidth* and *nWidth* have different signs, the function creates a mirror image of the bitmap along the x-axis. If *nSrcHeight* and *nHeight* have different signs, the function creates a mirror image of the bitmap along the y-axis.

The **StretchBlt** function stretches or compresses the source bitmap in memory and then copies the result to the destination. If a pattern is to be merged with the result, it is not merged until the stretched source bitmap is copied to the destination. If a brush is used, it is the selected brush in the destination device context. The destination coordinates are transformed according to the destination device context; the source coordinates are transformed according to the source device context.

If the destination, source, and pattern bitmaps do not have the same color format, **StretchBlt** converts the source and pattern bitmaps to match the destination bitmaps. The foreground and background colors of the destination device context are used in the conversion. If **StretchBlt** must convert a monochrome bitmap to color, it sets white bits (1) to the background color and black bits (0) to the foreground color. To convert color to monochrome, it sets pixels that match the background color to white (1) and sets all other pixels to black (0). The foreground and background colors of the device context with color are used.

Not all devices support the **StretchBlt** function. To determine whether a device supports **StretchBlt**, call the **GetDeviceCaps** member function with the **RASTERCAPS** index and check the return value for the **RC_STRETCHBLT** flag.

Return Value

Nonzero if the bitmap is drawn; otherwise 0.

See Also

CDC::BitBlt, **CDC::GetDeviceCaps**, **CDC::SetStretchBltMode**, **::StretchBlt**

CDC::TabbedTextOut

virtual CSize TabbedTextOut(int *x*, **int** *y*, **LPCSTR** *lpszString*, **int** *nCount*, **int** *nTabPositions*, **LPINT** *lpnTabStopPositions*, **int** *nTabOrigin* **);**

x Specifies the logical x-coordinate of the starting point of the string.

y Specifies the logical y-coordinate of the starting point of the string.

lpszString Points to the character string to draw. You can pass either a pointer to an array of characters or a **CString** object for this parameter.

nCount Specifies the number of characters in the string.

nTabPositions Specifies the number of values in the array of tab-stop positions.

lpnTabStopPositions Points to an array containing the tab-stop positions (in logical units). The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array.

nTabOrigin Specifies the x-coordinate of the starting position from which tabs are expanded (in logical units).

Remarks

Writes a character string at the specified location, expanding tabs to the values specified in the array of tab-stop positions. Text is written in the currently selected font. If *nTabPositions* is 0 and *lpnTabStopPositions* is **NULL**, tabs are expanded to eight times the average character width. If *nTabPositions* is 1, the tab stops are separated by the distance specified by the first value in the *lpnTabStopPositions* array. If the *lpnTabStopPositions* array contains more than one value, a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

The *nTabOrigin* parameter allows an application to call the **TabbedTextOut** function several times for a single line. If the application calls the function more than once with the *nTabOrigin* set to the same value each time, the function expands all tabs relative to the position specified by *nTabOrigin*.

By default, the current position is not used or updated by the function. If an application needs to update the current position when it calls the function, the application can call the **SetTextAlign** member function with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores the *x* and *y* parameters on subsequent calls to **TabbedTextOut**, using the current position instead.

Return Value

The dimensions of the string (in logical units) as a **CSize** object.

See Also

CDC::GetTabbedTextExtent, **CDC::SetTextAlign**, **CDC::TextOut**, **CDC::SetTextColor**, **::TabbedTextOut**, **CSize**

CDC::TextOut

virtual BOOL TextOut(int *x*, **int** *y*, **LPCSTR** *lpszString*, **int** *nCount* **);**

virtual BOOL TextOut(int *x*, **int** *y*, **const CString&** *str*);

x Specifies the logical x-coordinate of the starting point of the text.

y Specifies the logical y-coordinate of the starting point of the text.

lpszString Points to the character string to be drawn.

nCount Specifies the number of bytes in the string.

str A **CString** object that contains the characters to be drawn.

Remarks

Writes a character string at the specified location using the currently selected font. Character origins are at the upper-left corner of the character cell. By default, the current position is not used or updated by the function. If an application needs to update the current position when it calls **TextOut**, the application can call the **SetTextAlign** member function with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores the *x* and *y* parameters on subsequent calls to **TextOut**, using the current position instead.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CDC::ExtTextOut, **CDC::GetTextExtent**, **CDC::SetTextAlign**, **CDC::SetTextColor**, **CDC::TabbedTextOut**, **::TextOut**

CDC::UpdateColors

void UpdateColors( );

Remarks

Updates the client area of the device context by matching the current colors in the client area to the system palette on a pixel-by-pixel basis. An inactive window with a realized logical palette may call **UpdateColors** as an alternative to redrawing its client area when the system palette changes. For more information on using color palettes, see the Windows SDK documentation. The **UpdateColors** member function typically updates a client area faster than redrawing the area. However, because the function performs the color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy.

See Also

CDC::RealizePalette, **CPalette**, **::UpdateColors**

Data Members

CDC::m_hAttribDC

Remarks

The attribute device context for this **CDC** object. By default, this device context is equal to **m_hDC**. In general, **CDC** GDI calls that request information from the device context are directed to **m_hAttribDC**. See the **CDC** class description for more on the use of these two device contexts.

See Also

CDC::m_hDC, **CDC::SetAttribDC**, **CDC::ReleaseAttribDC**

CDC::m_hDC

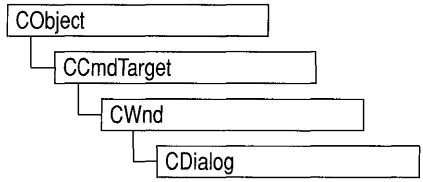
Remarks

The output device context for this **CDC** object. By default, **m_hDC** is equal to **m_hAttribDC**, the other device context wrapped by **CDC**. In general, **CDC** GDI calls that create output go to the **m_hDC** device context. You can initialize **m_hDC** and **m_hAttribDC** to point to different devices. See the **CDC** class description for more on the use of these two device contexts.

See Also

CDC::m_hAttribDC, **CDC::SetOutputDC**, **CDC::ReleaseOutputDC**

class CDialog : public CWnd



The **CDialog** class is the base class used for displaying dialog boxes on the screen. Dialog boxes are of two types: modal and modeless. A modal dialog box must be closed by the user before the application continues. A modeless dialog box allows the user to display the dialog box and return to another task without canceling or removing the dialog box.

A **CDialog** object is a combination of a dialog template and a **CDialog**-derived class. Use App Studio to create the dialog template and store it in a resource; then use ClassWizard to create a class derived from **CDialog**.

A dialog box, like any other window, receives messages from Windows. In a dialog box, you are particularly interested in handling notification messages from the dialog box's controls since that is how the user interacts with your dialog box. ClassWizard browses through the potential messages generated by each control in your dialog box, and you can select which messages you wish to handle. ClassWizard then adds the appropriate message-map entries and message-handler member functions to the new class for you. You only need to write application-specific code in the handler member functions.

If you prefer, you can always write message-map entries and member functions yourself instead of using ClassWizard.

In all but the most trivial dialog box, you add member variables to your derived dialog class to store data entered in the dialog box's controls by the user or to display data for the user. ClassWizard browses through those controls in your dialog box that can be mapped to data and prompts you to create a member variable for each control. At the same time, you choose a variable type and permissible range of values for each variable. ClassWizard adds the member variables to your derived dialog class.

ClassWizard then writes a data map to automatically handle the exchange of data between the member variables and the dialog box's controls. The data map provides functions that initialize the controls in the dialog box with the proper values, retrieve the data, and validate the data.

To create a modal dialog box, construct an object on the stack using the constructor for your derived dialog class and then call **DoModal** to create the dialog window and its controls. If you wish to create a modeless dialog, call **Create** in the constructor of your dialog class.

You can also create a template in memory by using a **DialogBoxResource** data structure as described in the *Windows Software Development Kit* documentation. After you construct a **CDialog** object, call **CreateIndirect** to create a modeless dialog box, or call **InitModalIndirect** and **DoModal** to create a modal dialog box.

ClassWizard writes the exchange and validation data map in an override of **CWnd::DoDataExchange** that ClassWizard adds to your new dialog class. See the **DoDataExchange** member function in **CWnd** for more on the exchange and validation functionality.

Both the programmer and the framework call **DoDataExchange** indirectly through a call to **CWnd::UpdateData**.

The framework calls **UpdateData** when the user clicks the OK button to close a modal dialog box. (The data is not retrieved if the Cancel button is clicked.) The default implementation of **OnInitDialog** also calls **UpdateData** to set the initial values of the controls. You typically override **OnInitDialog** to further initialize controls. **OnInitDialog** is called after all the dialog controls are created and just before the dialog box is displayed.

You can call **CWnd::UpdateData** at any time during the execution of a modal or modeless dialog box.

If you develop a dialog box by hand, you add the necessary member variables to the derived dialog-box class yourself, and you add member functions to set or get these values.

For more on App Studio, see the *App Studio User's Guide*. For more on ClassWizard, see Chapter 9 of the *App Studio User's Guide*, and Chapters 6 and 7 of the *Class Library User's Guide*.

Call **CWinApp::SetDialogBkColor** to set the background color for dialog boxes in your application.

A modal dialog box closes automatically when the user presses the OK or Cancel buttons or when your code calls the **EndDialog** member function.

When you implement a modeless dialog box, always override the **OnCancel** member function and call **DestroyWindow** from within it. Don't call the base class **CDialog::OnCancel**, because it calls **EndDialog**, which will make the dialog box invisible but will not destroy it. You should also override **PostNcDestroy** for modeless dialog boxes in order to delete **this**, since modeless dialog boxes are usually allocated with **new**. Modal dialog boxes are usually constructed on the frame and do not need **PostNcDestroy** cleanup.

#include <afxwin.h>

Construction/Destruction — Public Members

CDialog	Constructs a **CDialog** object.

Initialization — Public Members

InitModalIndirect	Creates a modal dialog box from a dialog-box template in memory (not resource-based). The parameters are stored until the function **DoModal** is called.

Operations — Public Members

DoModal	Invokes a modal dialog box and returns when done.
MapDialogRect	Converts the dialog-box units of a rectangle to screen units.
IsDialogMessage	Determines whether the given message is intended for the modeless dialog box and, if so, processes it.
NextDlgCtrl	Moves the focus to the next dialog-box control in the dialog box.
PrevDlgCtrl	Moves the focus to the previous dialog-box control in the dialog box.
GotoDlgCtrl	Moves the focus to a specified dialog-box control in the dialog box.
SetDefID	Changes the default pushbutton control for a dialog box to a specified pushbutton.
GetDefID	Gets the ID of the default pushbutton control for a dialog box.
SetHelpID	Sets a context-sensitive help ID for the dialog box.
EndDialog	Closes a modal dialog box.

Overridables — Public Members

OnInitDialog	Override to augment dialog-box initialization.
OnSetFont	Override to specify the font that a dialog-box control is to use when it draws text.
OnOK	Override to perform the OK button action in a modal dialog box. The default closes the dialog box and **DoModal** returns **IDOK**.
OnCancel	Override to perform the Cancel button or ESC key action. The default closes the dialog box and **DoModal** returns **IDCANCEL**.

Construction/Destruction—Protected Members

CDialog	Constructs a **CDialog** object.

Initialization—Protected Members

Create	Initializes the **CDialog** object. Creates a modeless dialog box and attaches it to the **CDialog** object.
CreateIndirect	Creates a modeless dialog box from a dialog-box template in memory (not resource-based).

Member Functions

CDialog::CDialog

CDialog(LPCSTR *lpszTemplateName*, **CWnd*** *pParentWnd* = **NULL);**

CDialog(UINT *nIDTemplate*, **CWnd*** *pParentWnd* = **NULL);**

Protected **CDialog( );** ♦

lpszTemplateName Contains a null-terminated string that is the name of a dialog-box template resource.

nIDTemplate Contains the ID number of a dialog-box template resource.

pParentWnd Points to the parent or owner window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

Remarks To construct a resource-based modal dialog box, invoke either public form of the constructor. One form of the constructor provides access to the dialog resource by template name. The other constructor provides access by template ID number, usually with an **IDD_** prefix (for example, IDD_DIALOG1).

To construct a modal dialog box from a template in memory, first invoke the parameterless, protected constructor and then call **InitModalIndirect**.

After you construct a modal dialog box with one of the above methods, call **DoModal**.

To construct a modeless dialog box, use the protected form of the **CDialog** constructor. The constructor is protected because you must derive your own dialog-box class to implement a modeless dialog box. Construction of a modeless dialog box is a two-step process. First invoke the constructor; then call the **Create** member function to create a resource-based dialog box, or call **CreateIndirect** to create the dialog box from a template in memory.

See Also **CDialog::Create**, **CWnd::DestroyWindow**, **CDialog::InitModalIndirect**, **CDialog::DoModal**, **::CreateDialog**

CDialog::Create

Protected **BOOL Create(LPCSTR** *lpszTemplateName*, **CWnd*** *pParentWnd* = **NULL);**

BOOL Create(UINT *nIDTemplate*, **CWnd*** *pParentWnd* = **NULL);** ♦

lpszTemplateName Contains a null-terminated string that is the name of a dialog-box template resource.

pParentWnd Points to the parent window object (of type **CWnd**) to which the dialog object belongs. If it is **NULL**, the dialog object's parent window is set to the main application window.

nIDTemplate Contains the ID number of a dialog-box template resource.

Remarks Call **Create** to create a modeless dialog box using a dialog-box template from a resource. You can put the call to **Create** inside the constructor or call it after the constructor is invoked.

Two forms of the **Create** member function are provided for access to the dialog-box template resource by either template name or template ID number (for example, IDD_DIALOG1).

For either form, pass a pointer to the parent window object. If *pParentWnd* is **NULL**, the dialog box will be created with its parent or owner window set to the main application window.

The **Create** member function returns immediately after it creates the dialog box.

Use the **WS_VISIBLE** style in the dialog-box template if the dialog box should appear when the parent window is created. Otherwise, you must call **ShowWindow**. For further dialog-box styles and their application, see the *Windows Software Development Kit* (SDK) documentation and App Studio documentation.

Use the **CWnd::DestroyWindow** function to destroy a dialog box created by the **Create** function.

Return Value Both forms return nonzero if dialog box creation and initialization was successful; otherwise 0.

See Also **CDialog::CDialog**, **CWnd::DestroyWindow**, **CDialog::InitModalIndirect**, **CDialog::DoModal**, **::CreateDialog**

CDialog::CreateIndirect

Protected **BOOL CreateIndirect(const void FAR*** *lpDialogTemplate*, **CWnd*** *pParentWnd* = **NULL);** ♦

lpDialogTemplate Points to memory that contains a dialog-box template used to create the dialog box. This template is in the form of a **DialogBoxHeader** structure and control information. For more information on this structure, see the *Software Development Kit* for Windows version 3.1.

pParentWnd Points to the dialog object's parent window object (of type **CWnd**). If it is **NULL**, the dialog object's parent window is set to the main application window.

Remarks Call this member function to create a modeless dialog box from a dialog-box template in memory.

The **CreateIndirect** member function returns immediately after it creates the dialog box.

Use the **WS_VISIBLE** style in the dialog-box template if the dialog box should appear when the parent window is created. Otherwise, you must call **ShowWindow** to cause it to appear. For more information on how you can specify other dialog-box styles in the template, see the *Windows SDK* documentation and the *App Studio User's Guide*.

Use the **CWnd::DestroyWindow** function to destroy a dialog box created by the **CreateIndirect** function.

Return Value Nonzero if the dialog was created and initialized successfully; otherwise 0.

See Also **CDialog::CDialog**, **CWnd::DestroyWindow**, **CDialog::Create**, **::CreateDialogIndirect**

CDialog::DoModal

virtual int DoModal( );

Remarks

Call this member function to invoke the modal dialog box and return the dialog box result when done. This member function handles all interaction with the user while the dialog box is active. This is what makes the dialog box modal; that is, the user cannot interact with other windows until the dialog box is closed.

If the user clicks one of the pushbuttons in the dialog box, such as OK or Cancel, a message-handler member function, such as **OnOK** or **OnCancel**, is called to attempt to close the dialog box. The default **OnOK** member function will validate and update the dialog-box data and close the dialog box with result **IDOK**, and the default **OnCancel** member function will close the dialog box with result **IDCANCEL** without validating or updating the dialog-box data. You can override these message-handler functions to alter their behavior.

Return Value

An **int** value that specifies the value of the *nResult* parameter that was passed to the **CDialog::EndDialog** member function, which is used to close the dialog box. The return value is –1 if the function could not create the dialog box, or **IDABORT** if some other error occurred.

See Also

::DialogBox

CDialog::EndDialog

void EndDialog(int *nResult* **);**

nResult Contains the value to be returned from the dialog box to the caller of **DoModal**.

Remarks

Call this member function to terminate a modal dialog box. This member function returns *nResult* as the return value of **DoModal**. You must use the **EndDialog** function to complete processing whenever a modal dialog box is created.

You can call **EndDialog** at any time, even in **OnInitDialog**, in which case you should close the dialog box before it is shown or before the input focus is set.

EndDialog does not close the dialog box immediately. Instead, it sets a flag that directs the dialog box to close as soon as the current message handler returns.

See Also

CDialog::DoModal, **CDialog::OnOK**, **CDialog::OnCancel**

CDialog::GetDefID

DWORD GetDefID( ) const;

Remarks

Call the **GetDefID** member function to get the ID of the default pushbutton control for a dialog box. This is usually an OK button.

Return Value

A 32-bit value (**DWORD**). If the default pushbutton has an ID value, the high-order word contains **DC_HASDEFID** and the low-order word contains the ID value. If the default pushbutton does not have an ID value, the return value is 0.

See Also

CDialog::SetDefID, **DM_GETDEFID**

CDialog::GotoDlgCtrl

void GotoDlgCtrl(CWnd* *pWndCtrl* **);**

pWndCtrl Identifies the window (control) that is to receive the focus.

Remarks

Moves the focus to the specified control in the dialog box.

To get a pointer to the control (child window) to pass as *pWndCtrl*, call the **CWnd::GetDlgItem** member function, which returns a pointer to a **CWnd** object.

See Also

CWnd::GetDlgItem, **CDialog::PrevDlgCtrl**, **CDialog::NextDlgCtrl**

CDialog::InitModalIndirect

BOOL InitModalIndirect(HGLOBAL *hDialogTemplate* **);**

hDialogTemplate Contains a handle to global memory containing a dialog-box template. This template is in the form of a **DialogBoxHeader** structure and data for each control in the dialog box. For more information on this structure, see the *Software Development Kit* for Windows version 3.1.

Remarks

Call this member function to initialize a modal dialog object using a dialog-box template that you construct in memory.

To create a modal dialog indirectly, first allocate a global block of memory and fill it with the dialog box template. Then call the empty **CDialog** constructor to construct the dialog-box object. Next, call **InitModalIndirect** to store your handle to the in-memory dialog-box template. The Windows dialog box is created and displayed later, when the **DoModal** member function is called.

Return Value

Nonzero if the dialog object was created and initialized successfully; otherwise 0.

See Also

::DialogBoxIndirect, **CDialog::DoModal**, **CWnd::DestroyWindow**, **DialogBoxResource**, **CDialog::CDialog**, **CDialog::DoModal**

CDialog::IsDialogMessage

BOOL IsDialogMessage(LPMSG *lpMsg* **);**

lpMsg Points to an **MSG** structure that contains the message to be checked.

Remarks

Call this member function to determine whether the given message is intended for a modeless dialog box; if it is, this function processes the message. When the **IsDialogMessage** function processes a message, it checks for keyboard messages and converts them to selection commands for the corresponding dialog box. For example, the TAB key selects the next control or group of controls, and the DOWN ARROW key selects the next control in a group.

You must not pass a message processed by **IsDialogMessage** to the **TranslateMessage** or **DispatchMessage** Windows functions because it has already been processed.

Return Value

Specifies whether the member function has processed the given message. It is nonzero if the message has been processed; otherwise 0. If the return is 0, call the **PreTranslateMessage** member function of the base class to process the message. In an override of the **CDialog::PreTranslateMessage** member function the code looks like this :

```
BOOL CMyDlg::PreTranslateMessage( msg )
{
    if( IsDialogMessage( msg ) )
        return TRUE;
    else
        return CDialog::PreTranslateMessage( msg );
}
```

See Also

::DispatchMessage, **::TranslateMessage**, **::GetMessage**, **CWnd::PreTranslateMessage**, **::IsDialogMessage**

CDialog::MapDialogRect

void MapDialogRect(LPRECT *lpRect* **) const;**

lpRect Points to a **RECT** structure or **CRect** object that contains the dialog-box coordinates to be converted.

Remarks

Call to convert the dialog-box units of a rectangle to screen units. Dialog-box units are stated in terms of the current dialog-box base unit derived from the average width and height of characters in the font used for dialog-box text. One horizontal unit is one-fourth of the dialog-box base-width unit, and one vertical unit is one-eighth of the dialog-box base height unit.

The **GetDialogBaseUnits** Windows function returns size information for the system font, but you can specify a different font for each dialog box if you use the **DS_SETFONT** style in the resource-definition file. The **MapDialogRect** Windows function uses the appropriate font for this dialog box.

The **MapDialogRect** member function replaces the dialog-box units in *lpRect* with screen units (pixels) so that the rectangle can be used to create a dialog box or position a control within a box.

See Also

::GetDialogBaseUnits, **::MapDialogRect**, **WM_SETFONT**

CDialog::NextDlgCtrl

void NextDlgCtrl( ) const;

Remarks

Moves the focus to the next control in the dialog box. If the focus is at the last control in the dialog box, it moves to the first control.

See Also

CDialog::PrevDlgCtrl, **CDialog::GotoDlgCtrl**

CDialog::OnCancel

Protected

virtual void OnCancel( ); ♦

Remarks

The framework calls this member function when the user clicks the Cancel button or presses the ESC key in a modal or modeless dialog box.

Override this member function to perform Cancel button action. The default simply terminates a modal dialog box by calling **EndDialog** and causing **DoModal** to return **IDCANCEL**.

If you implement the Cancel button in a modeless dialog box, you must override the **OnCancel** member function and call **DestroyWindow** from within it. Don't call the base-class member function, because it calls **EndDialog**, which will make the dialog box invisible but not destroy it.

See Also **CDialog::OnOK**, **CDialog::EndDialog**

CDialog::OnInitDialog

virtual BOOL OnInitDialog( );

Remarks

This member function is called in response to the **WM_INITDIALOG** message. This message is sent to the dialog box during the **Create, CreateIndirect**, or **DoModal** calls, which occur immediately before the dialog box is displayed.

Override this member function if you need to perform special processing when the dialog box is initialized. In the overridden version, first call the base class **OnInitDialog** but disregard its return value. You will normally return **TRUE** from your overridden member function.

Windows calls the **OnInitDialog** function via the standard global dialog-box procedure common to all Microsoft Foundation Class Library dialog boxes, rather than through your message map, so you do not need a message-map entry for this member function.

Return Value

Specifies whether the application has set the input focus to one of the controls in the dialog box. If **OnInitDialog** returns nonzero, Windows sets the input focus to the first control in the dialog box. The application can return 0 only if it has explicitly set the input focus to one of the controls in the dialog box.

See Also **CDialog::Create**, **CDialog::CreateIndirect**, **WM_INITDIALOG**

CDialog::OnOK

Protected **virtual void OnOK( );** ♦

Remarks Called when the user clicks the OK button (the button with an ID of **IDOK**).

Override this member function to perform the OK button action. If the dialog box includes automatic data validation and exchange, the default implementation of this member function validates the dialog-box data and updates the appropriate variables in your application.

If you implement the OK button in a modeless dialog box, you must override the **OnOK** member function and call **DestroyWindow** from within it. Don't call the base-class member function, because it calls **EndDialog**, which makes the dialog box invisible but does not destroy it.

See Also **CDialog::OnCancel**, **CDialog::EndDialog**

CDialog::OnSetFont

virtual void OnSetFont(CFont* *pFont* **);**

pFont Specifies a pointer to the font. Used as the default font for all controls in this dialog box.

Remarks Specifies the font a dialog-box control will use when drawing text. The dialog-box control will use the specified font as the default for all dialog-box controls. App Studio typically sets the dialog-box font as part of the dialog-box template resource.

See Also **WM_SETFONT**, **CWnd::SetFont**

CDialog::PrevDlgCtrl

void PrevDlgCtrl( ) const;

Remarks Sets the focus to the previous control in the dialog box. If the focus is at the first control in the dialog box, it moves to the last control in the box.

See Also **CDialog::NextDlgCtrl**, **CDialog::GotoDlgCtrl**

CDialog::SetDefID

void SetDefID(UINT *nID* **);**

nID Specifies the ID of the pushbutton control that will become the default.

Remarks Changes the default pushbutton control for a dialog box.

See Also **CDialog::GetDefID**

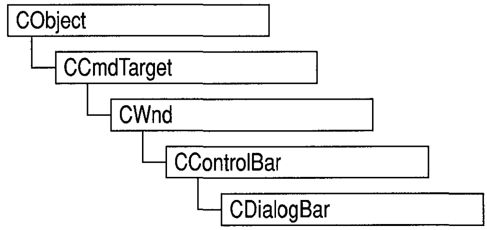
CDialog::SetHelpID

void SetHelpID(UINT *nIDR* **);**

nIDR Specifies the context-sensitive help ID.

Remarks Sets a context-sensitive help ID for the dialog box.

class CDialogBar : public CControlBar



The **CDialogBar** class provides the functionality of a Windows modeless dialog box in a control bar. A dialog bar resembles a dialog box in that it contains standard Windows controls that the user can tab between. Another similarity is that you create a dialog template to represent the dialog bar.

Creating and using a dialog bar is similar to creating and using a **CFormView** object (see *App Studio User's Guide*, Chapter 3). First, use App Studio to define a dialog template with the style **WS_CHILD** and no other style. The template must not have the style **WS_VISIBLE**. In your application code, call the constructor to construct the **CDialogBar** object, then call **Create** to create the dialog-bar window and attach it to the **CDialogBar** object.

#include <afxext.h>

See Also **CControlBar**, **CFormView**

Construction/Destruction—Public Members

CDialogBar	Constructs a **CDialogBar** object.
Create	Creates a Windows dialog bar and attaches it to the **CDialogBar** object.

Member Functions

CDialogBar::CDialogBar

CDialogBar( );

Remarks Constructs a **CDialogBar** object.

See Also **CControlBar**

CDialogBar::Create

BOOL Create(CWnd* *pParentWnd*, **LPCSTR** *lpszTemplateName*, **UINT** *nStyle*, **UINT** *nID* **);**

BOOL Create(CWnd* *pParentWnd*, **UINT** *nIDTemplate*, **UINT** *nStyle*, **UINT** *nID* **);**

pParentWnd A pointer to the parent **CWnd** object.

lpszTemplateName A pointer to the name of the **CDialogBar** object's dialog-box resource template.

nStyle The alignment style of the dialog bar. The styles supported and their meanings are as follows:

- **CBRS_BOTTOM** Control bar is at the bottom of the frame window.
- **CBRS_NOALIGN** Control bar is not repositioned when the parent is resized.
- **CBRS_LEFT** Control bar is at the left of the frame window.
- **CBRS_RIGHT** Control bar is at the right of the frame window.

nID The control ID of the dialog bar.

nIDTemplate The resource ID of the **CDialogBar** object's dialog-box template.

Remarks

Loads the dialog-box resource template specified by *lpszTemplateName* or *nIDTemplate*, creates the dialog-bar window, sets its style, and associates it with the **CDialogBar** object.

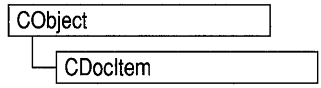
Return Value

Nonzero if successful; otherwise 0.

See Also

CDialogBar::CDialogBar

class CDocItem : public CObject



CDocItem is the base class for document items, which are components of a document's data. **CDocItem** objects are used to represent Object Linking and Embedding (OLE) items in both client and server documents.

Typically you do not use the **CDocItem** class directly. Instead, you use its derived classes **COleClientItem** or **COleServerItem**.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **COleDocument**, **COleServerItem**, **COleClientItem**

Operations—Public Members

GetDocument	Returns the document that contains the item.

Member Functions

CDocItem::GetDocument

CDocument* GetDocument( ) const;

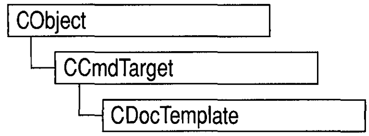
Remarks Call this function to get the document that contains the item. This function is overridden in the derived classes **COleClientItem** and **COleServerItem** to return pointers to **COleClientDoc** and **COleServerDoc**, respectively.

Return Value A pointer to the document that contains the item, or **NULL** if the item is not part of a document.

See Also **COleDocument**, **COleServerDoc**, **COleClientDoc**

class CDocTemplate : public CCmdTarget

CDocTemplate is an abstract base class that defines the basic functionality for document templates. A document template defines the relationship between three types of classes:



- A document class, which you derive from **CDocument**.
- A view class, which displays data from the document class listed above. You can derive this class from **CView**, **CScrollView**, **CFormView**, or **CEditView**. (You can also use **CEditView** directly.)
- A frame window class, which contains the view. For a single document interface (SDI) application, you derive this class from **CFrameWnd**. For a multiple document interface (MDI) application, you derive this class from **CMDIChildWnd**. If you don't need to customize the behavior of the frame window, you can use **CFrameWnd** or **CMDIChildWnd** directly without deriving your own class.

Your application has one document template for each type of document that it supports. For example, if your application supports both spreadsheets and text documents, the application has two document template objects. Each document template is responsible for creating and managing all the documents of its type.

The document template stores pointers to the **CRuntimeClass** objects for the document, view, and frame window classes. These **CRuntimeClass** objects are specified when constructing a document template.

The document template contains the ID of the resources used with the document type (such as menu, icon, or accelerator table resources). The document template also has strings containing additional information about its document type. These include the name of the document type (for example, "Worksheet"), the file extension (for example, ".xls"), and, optionally, other strings used by the application's user interface, the Windows File Manager, and Object Linking and Embedding (OLE) support.

Since **CDocTemplate** is an abstract class, you cannot use the class directly. A typical application uses one of the two **CDocTemplate**-derived classes that the Microsoft Foundation Class Library provides: **CSingleDocTemplate**, which implements SDI, and **CMultiDocTemplate**, which implements MDI. See those classes for more information on using document templates.

If your application requires a user-interface paradigm that is fundamentally different from SDI or MDI, you can derive your own class from **CDocTemplate**.

See Also **CSingleDocTemplate**, **CMultiDocTemplate**, **CDocument**, **CView**, **CScrollView**, **CEditView**, **CFormView**, **CFrameWnd**, **CMDIChildWnd**

Operations — Public Members

GetDocString	Retrieves a string describing the document type.

Member Functions

CDocTemplate::GetDocString

virtual BOOL GetDocString(CString& *rString*, **enum DocStringIndex** *index* **) const;**

rString A reference to a **CString** object that will contain the string when the function returns.

index An index of the substring being retrieved from the string describing the document type. This parameter can have one of the following values:

- **CDocTemplate::windowTitle** Name that appears in the application window's title bar (for example, "Microsoft Excel"). Present only in the document template for SDI applications.
- **CDocTemplate::docName** Root for the default document name (for example, "Sheet"). This root, plus a number, is used for the default name of a new document of this type whenever the user chooses the New command from the File menu (for example, "Sheet1" or "Sheet2"). If not specified, "Untitled" is used as the default.
- **CDocTemplate::fileNewName** Name of this document type. If the application supports more than one type of document, this string is displayed in the File New dialog box (for example, "Worksheet"). If not specified, the document type is inaccessible using the File New command.
- **CDocTemplate::filterName** Description of the document type and a wildcard filter matching documents of this type. This string is displayed in the List Files Of Type drop-down list in the File Open dialog box (for example, "Worksheets (*.xls)"). If not specified, the document type is inaccessible using the File Open command.

- **CDocTemplate::filterExt** Extension for documents of this type (for example, ".xls"). If not specified, the document type is inaccessible using the File Open command.
- **CDocTemplate::regFileTypeId** Identifier for the document type to be stored in the registration database maintained by Windows. This string is for internal use only (for example, "ExcelWorksheet"). If not specified, the document type cannot be registered with the Windows File Manager.
- **CDocTemplate::regFileTypeName** Name of the document type to be stored in the registration database. This string may be displayed in dialog boxes of applications that access the registration database (for example, "Microsoft Excel Worksheet").

If you are using AppWizard to create a set of starter files, the last four substrings are present only if you specify a filename extension for your application's documents when running AppWizard.

Remarks

Call this function to retrieve a specific substring describing the document type. The string containing these substrings is stored in the document template and is derived from a string in the resource file for the application. The framework calls this function to get the strings it needs for the application's user interface. If you have specified a filename extension for your application's documents, the framework also calls this function when adding an entry to the Windows registration database; this allows documents to be opened from the Windows File Manager.

Call this function only if you are deriving your own class from **CDocTemplate**.

Return Value

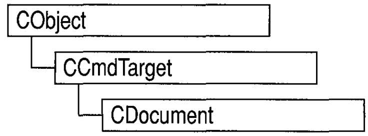
Nonzero if the specified substring was found; otherwise 0.

See Also

CMultiDocTemplate::CMultiDocTemplate, **CSingleDocTemplate::CSingleDocTemplate**, **CWinApp::RegisterShellFileTypes**

class CDocument : public CCmdTarget

The **CDocument** class provides the basic functionality for user-defined document classes. A document represents the unit of data that the user typically opens with the File Open command and saves with the File Save command.



CDocument supports standard operations such as creating a document, loading it, and saving it. The framework manipulates documents using the interface defined by **CDocument**.

An application can support more than one type of document; for example, an application might support both spreadsheets and text documents. Each type of document has an associated document template; the document template specifies what resources (for example, menu, icon, or accelerator table) are used for that type of document. Each document contains a pointer to its associated **CDocTemplate** object.

Users interact with a document through the **CView** object(s) associated with it. A view renders an image of the document in a frame window and interprets user input as operations on the document. A document can have multiple views associated with it. When the user opens a window on a document, the framework creates a view and attaches it to the document. The document template specifies what type of view and frame window are used to display each type of document.

Documents are part of the framework's standard command routing and consequently receive commands from standard user-interface components (such as the File Save menu item). A document receives commands forwarded by the active view. If the document doesn't handle a given command, it forwards the command to the document template that manages it.

When a document's data is modified, each of its views must reflect those modifications. **CDocument** provides the **UpdateAllViews** member function for you to notify the views of such changes, so the views can repaint themselves as necessary. The framework also prompts the user to save a modified file before closing it.

To implement documents in a typical application, you must do the following:

- Derive a class from **CDocument** for each type of document.
- Add member variables to store each document's data.
- Implement member functions for reading and modifying the document's data. The document's views are the most important users of these member functions.
- Override the **Serialize** member function in your document class to write and read the document's data to and from disk.

#include <afxwin.h>

See Also CCmdTarget, CView, CDocTemplate

Construction/Destruction — Public Members

CDocument	Constructs a **CDocument** object.

Operations — Public Members

AddView	Attaches a view to the document.
GetDocTemplate	Returns a pointer to the document template that describes the type of the document.
GetFirstViewPosition	Returns the position of the first in the list of views; used to begin iteration.
GetNextView	Iterates through the list of views associated with the document.
GetPathName	Returns the path of the document's data file.
GetTitle	Returns the document's title.
IsModified	Indicates whether the document has been modified since it was last saved.
RemoveView	Detaches a view from the document.
SetModifiedFlag	Sets a flag indicating that you have modified the document since it was last saved.
SetPathName	Sets the path of the document's data file.
SetTitle	Sets the document's title.
UpdateAllViews	Notifies all views that document has been modified.

Overridables — Public Members

CanCloseFrame	Advanced overridable; called before closing a frame window viewing this document.
DeleteContents	Called to perform cleanup of the document.
OnChangedViewList	Called after a view is added to or removed from the document.
OnCloseDocument	Called to close the document.
OnNewDocument	Called to create a new document.
OnOpenDocument	Called to open an existing document.
OnSaveDocument	Called to save the document to disk.

ReportSaveLoadException	Advanced overridable; called when an open or save operation cannot be completed because of an exception.
SaveModified	Advanced overridable; called to ask the user whether the document should be saved.

Member Functions

CDocument::AddView

void AddView(CView* *pView* **);**

pView Points to the view being added.

Remarks

Call this function to attach a view to the document. This function adds the specified view to the list of views associated with the document; the function also sets the view's document pointer to this document. The framework calls this function when attaching a newly created view object to a document; this occurs in response to a File New, File Open, or New Window command or when a splitter window is split.

Call this function only if you are manually creating and attaching a view. Typically you will let the framework connect documents and views by defining a **CDocTemplate** object to associate a document class, view class, and frame window class.

See Also

CDocTemplate, **CDocument::GetFirstViewPosition**, **CDocument::GetNextView**, **CDocument::RemoveView**, **CView::GetDocument**

CDocument::CanCloseFrame

virtual BOOL CanCloseFrame(CFrameWnd* *pFrame* **);**

pFrame Points to the frame window of a view attached to the document.

Remarks

Called by the framework before a frame window displaying the document is closed. The default implementation checks if there are other frame windows displaying the document. If the specified frame window is the last one that displays the document, the function prompts the user to save the document if it has been modified. Override this function if you want to perform special processing when a frame window is closed. This is an advanced overridable.

Return Value Nonzero if it is safe to close the frame window; otherwise 0.

See Also **CDocument::SaveModified**

CDocument::CDocument

CDocument( );

Remarks Constructs a **CDocument** object. The framework handles document creation for you. Override the **OnNewDocument** member function to perform initialization on a per-document basis; this is particularly important in single document interface (SDI) applications.

See Also **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**

CDocument::DeleteContents

virtual void DeleteContents( );

Remarks Called by the framework to delete the document's data without destroying the document object itself. It is called just before the document is to be destroyed. It is also called to ensure that a document is empty before it is reused. This is particularly important for an SDI application, which uses only one document object; the document object is reused whenever the user creates or opens another document. Call this function to implement an Edit Clear All or similar command that deletes all of the document's data. The default implementation of this function does nothing. Override this function to delete the data in your document.

See Also **CDocument::OnCloseDocument**, **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**

CDocument::GetDocTemplate

CDocTemplate* GetDocTemplate( ) const;

Remarks Call this function to get a pointer to the document template for this document type.

Return Value A pointer to the document template for this document type, or **NULL** if the document is not managed by a document template.

See Also **CDocTemplate**

CDocument::GetFirstViewPosition

virtual POSITION GetFirstViewPosition( ) const;

Remarks Call this function to get the position of the first view in the list of views associated with the document.

Return Value A **POSITION** value that can be used for iteration with the **GetNextView** member function.

See Also **CDocument::GetNextView**

Example To get the first view in the list of views:

```
POSITION pos = GetFirstViewPosition();
CView* pFirstView = GetNextView( pos );
```

CDocument::GetNextView

virtual CView* GetNextView(POSITION& *rPosition*) const;

rPosition A reference to a **POSITION** value returned by a previous call to the **GetNextView** or **GetFirstViewPosition** member functions. This value must not be **NULL**.

Remarks Call this function to iterate through all of the document's views. The function returns the view identified by *rPosition* and then sets *rPosition* to the **POSITION** value of the next view in the list. If the retrieved view is the last in the list, then *rPosition* is set to **NULL**.

Return Value A pointer to the view identified by *rPosition*.

See Also **CDocument::AddView**, **CDocument::GetFirstViewPosition**, **CDocument::RemoveView**, **CDocument::UpdateAllViews**

CDocument::GetPathName

const CString& GetPathName( ) const;

Remarks Call this function to get the fully qualified path of the document's disk file.

Return Value The document's fully qualified path. This string is empty if the document has not been saved or does not have a disk file associated with it.

See Also **CDocument::SetPathName**

CDocument::GetTitle

const CString& GetTitle( ) const;

Remarks Call this function to get the document's title, which is usually derived from the document's filename.

Return Value The document's title.

See Also **CDocument::SetTitle**

CDocument::IsModified

BOOL IsModified( );

Remarks Call this function to determine whether the document has been modified since it was last saved.

Return Value Nonzero if the document has been modified since it was last saved; otherwise 0.

See Also **CDocument::SetModifiedFlag**, **CDocument::SaveModified**

CDocument::OnChangedViewList

virtual void OnChangedViewList( );

Remarks

Called by the framework after a view is added to or removed from the document. The default implementation of this function checks whether the last view is being removed and, if so, deletes the document. Override this function if you want to perform special processing when the framework adds or removes a view. For example, if you want a document to remain open even when there are no views attached to it, override this function.

See Also

CDocument::AddView, **CDocument::RemoveView**

CDocument::OnCloseDocument

virtual void OnCloseDocument( );

Remarks

Called by the framework when the document is closed, typically as part of the File Close command. The default implementation of this function calls the **DeleteContents** member function to delete the document's data and then closes the frame windows for all the views attached to the document.

Override this function if you want to perform special cleanup processing when the framework closes a document. For example, if the document represents a record in a database, you may want to override this function to close the database. You should call the base class version of this function from your override.

See Also

CDocument::DeleteContents, **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**

CDocument::OnNewDocument

virtual BOOL OnNewDocument( );

Remarks

Called by the framework as part of the File New command. The default implementation of this function calls the **DeleteContents** member function to ensure that the document is empty and then marks the new document as clean. Override this function to initialize the data structure for a new document. You should call the base class version of this function from your override.

If the user chooses the File New command in an SDI application, the framework uses this function to reinitialize the existing document object, rather than creating a new one. If the user chooses File New in a multiple document interface (MDI) application, the framework creates a new document object each time and then calls this function to initialize it. You must place your initialization code in this function instead of in the constructor for the File New command to be effective in SDI applications.

Return Value

Nonzero if the document was successfully initialized; otherwise 0.

See Also

CDocument::CDocument, **CDocument::DeleteContents**, **CDocument::OnCloseDocument**, **CDocument::OnOpenDocument**, **CDocument::OnSaveDocument**

CDocument::OnOpenDocument

virtual BOOL OnOpenDocument(const char* *pszPathName* **);**

pszPathName Points to the path of the document to be opened.

Remarks

Called by the framework as part of the File Open command. The default implementation of this function opens the specified file, calls the **DeleteContents** member function to ensure that the document is empty, calls **Serialize** to read the file's contents, and then marks the document as clean. Override this function if you want to use something other than the archive mechanism or the file mechanism. For example, you might write an application where documents represent records in a database rather than separate files.

If the user chooses the File Open command in an SDI application, the framework uses this function to reinitialize the existing document object, rather than creating a new one. If the user chooses File Open in an MDI application, the framework constructs a new document object each time and then calls this function to initialize it. You must place your initialization code in this function instead of in the constructor for the File Open command to be effective in SDI applications.

Return Value

Nonzero if the document was successfully loaded; otherwise 0.

See Also

CDocument::DeleteContents, **CDocument::OnCloseDocument**, **CDocument::OnNewDocument**, **CDocument::OnSaveDocument**, **CDocument::ReportSaveLoadException**, **CObject::Serialize**

CDocument::OnSaveDocument

virtual BOOL OnSaveDocument(const char* *pszPathName* **);**

pszPathName Points to the fully qualified path that the file should be saved to.

Remarks

Called by the framework as part of the File Save or File Save As command. The default implementation of this function opens the specified file, calls **Serialize** to write the document's data to the file, and then marks the document as clean. Override this function if you want to perform special processing when the framework saves a document. For example, you might write an application where documents represent records in a database rather than separate files.

Return Value

Nonzero if the document was successfully saved; otherwise 0.

See Also

CDocument::OnCloseDocument, **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**, **CDocument::ReportSaveLoadException**, **CObject::Serialize**

CDocument::RemoveView

void RemoveView(CView* *pView* **);**

pView Points to the view being removed.

Remarks

Call this function to detach a view from a document. This function removes the specified view from the list of views associated with the document; it also sets the view's document pointer to **NULL**. This function is called by the framework when a frame window is closed or a pane of a splitter window is closed.

Call this function only if you are manually detaching a view. Typically you will let the framework detach documents and views by defining a **CDocTemplate** object to associate a document class, view class, and frame window class.

See Also

CDocument::AddView, **CDocument::GetFirstViewPosition**, **CDocument::GetNextView**

CDocument::ReportSaveLoadException

virtual void ReportSaveLoadException(const char* *pszPathName*, **CException*** *e*, **BOOL** *bSaving*, **UINT** *nIDPDefault* **);**

pszPathName Points to name of document that was being saved or loaded.

e Points to the exception that was thrown.

bSaving Flag indicating what operation was in progress; nonzero if the document was being saved, 0 if the document was being loaded.

nIDPDefault Identifier of the error message to be displayed if the function does not specify a more specific one.

Remarks

Called if an exception is thrown (typically a **CFileException** or **CArchiveException**) while saving or loading the document. The default implementation examines the exception object and looks for an error message that specifically describes the cause. If a specific message is not found, the general message specified by the *nIDPDefault* parameter is used. The function then displays a message box containing the error message. Override this function if you want to provide additional, customized failure messages. This is an advanced overridable.

See Also

CDocument::OnOpenDocument, **CDocument::OnSaveDocument**, **CFileException**, **CArchiveException**

CDocument::SaveModified

virtual BOOL SaveModified( );

Remarks

Called by the framework before a modified document is to be closed. The default implementation of this function displays a message box asking the user whether to save the changes to the document, if any have been made. Override this function if your program requires a different prompting procedure. This is an advanced overridable.

Return Value

Nonzero if it is safe to continue and close the document; 0 if the document should not be closed.

See Also

CDocument::CanCloseFrame, **CDocument::IsModified**, **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**, **CDocument::OnSaveDocument**

CDocument::SetModifiedFlag

void SetModifiedFlag(BOOL *bModified* **= TRUE);**

bModified Flag indicating whether the document has been modified.

Remarks

Call this function after you have made any modifications to the document. By calling this function consistently, you ensure that the framework prompts the user to save changes before closing a document. Typically you should use the default value of **TRUE** for the *bModified* parameter. To mark a document as clean (unmodified), call this function with a value of **FALSE**.

See Also

CDocument::IsModified, **CDocument::SaveModified**

CDocument::SetPathName

virtual void SetPathName(const char* *pszPathName* **);**

pszPathName Points to the string to be used as the document's path.

Remarks

Call this function to specify the fully qualified path of the document's disk file. The path is added to the most recently used (MRU) file list maintained by the application. Note that some documents are not associated with a disk file. Call this function only if you are overriding the framework's default implementation for opening and saving files.

See Also

CDocument::GetPathName, **CWinApp::AddToRecentFileList**

CDocument::SetTitle

virtual void SetTitle(const char* *pszTitle* **);**

szTitle Points to the string to be used as the document's title.

Remarks

Call this function to specify the document's title (the string displayed in the title bar of a frame window). Calling this function updates the titles of all frame windows that display the document.

See Also

CDocument::GetTitle

CDocument::UpdateAllViews

void UpdateAllViews(CView* *pSender*, **LPARAM** *lHint* = **0L, CObject*** *pHint* = **NULL);**

pSender Points to the view that modified the document, or **NULL** if all views are to be updated.

lHint Contains information about the modification.

pHint Points to an object storing information about the modification.

Remarks

Call this function after the document has been modified. You should call this function after you call the **SetModifiedFlag** member function. This function informs each view attached to the document, except for the view specified by *pSender*, that the document has been modified. You typically call this function from your view class after the user has changed the document through a view.

This function calls the **OnUpdate** member function for each of the document's views except the sending view, passing *pHint* and *lHint*. Use these parameters to pass information to the views about the modifications made to the document. You can encode information using *lHint* and/or you can define a **CObject**-derived class to store information about the modifications and pass an object of that class using *pHint*. Override the **OnUpdate** member function in your **CView**-derived class to optimize the updating of the view's display based on the information passed.

See Also

CDocument::SetModifiedFlag, CDocument::GetFirstViewPosition, CDocument::GetNextView, CView::OnUpdate

class CDumpContext

The **CDumpContext** class supports stream-oriented diagnostic output in the form of human-readable text. You can use **afxDump**, a predeclared **CDumpContext** object, for most of your dumping. The **afxDump** object is available only in the Debug version of the Microsoft Foundation Class Library. Several of the memory diagnostic functions use **afxDump** for their output. The predefined **afxDump** object, conceptually similar to the **cerr** stream, is connected to **stderr** under MS-DOS. Under the Windows environment, the output is routed to the debugger via the Windows function **OutputDebugString**.

The **CDumpContext** class has an overloaded insertion (<<) operator for **CObject** pointers that dumps the object's data. If you need a custom dump format for a derived object, override **CObject::Dump**. Most Microsoft Foundation classes implement an overridden **Dump** member function.

Classes that are not derived from **CObject**, such as **CString**, **CTime**, and **CTimeSpan**, have their own overloaded **CDumpContext** insertion operators, as do often-used structures such as **CFileStatus**, **CPoint**, and **CRect**.

If you use the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macros in the implementation of your class, then **CObject::Dump** will print the name of your **CObject**-derived class. Otherwise, it will print `CObject`.

The **CDumpContext** class is available with both the Debug and Release versions of the library, but the **Dump** member function is defined only in the Debug version. Use **#ifdef _DEBUG / #endif** statements to bracket your diagnostic code, including your custom **Dump** member functions.

Before you create your own **CDumpContext** object, you must create a **CFile** object that serves as the dump destination.

#define _DEBUG

#include <afx.h>

See Also **CFile**, **CObject**

Construction/Destruction—Public Members

CDumpContext	Constructs a **CDumpContext** object.

Basic Input/Output—Public Members

Flush	Flushes any data in the dump context buffer.
operator <<	Inserts variables and objects into the dump context.
HexDump	Dumps bytes in hexadecimal format.

Status—Public Members

GetDepth	Gets an integer corresponding to the depth of the dump.
SetDepth	Sets the depth of the dump.

Member Functions

CDumpContext::CDumpContext

CDumpContext(CFile* *pFile* **)**
throw(CMemoryException, CFileException);

pFile A pointer to the **CFile** object that is the dump destination.

Remarks

Constructs an object of class **CDumpContext**. The **afxDump** object is constructed automatically. The output from **afxDump** is sent to **stderr** in MS-DOS. Do not write to the underlying **CFile** while the dump context is active; otherwise, you will interfere with the dump. Under the Windows environment, the output is routed to the debugger via the Windows function **OutputDebugString**.

Example

```
extern char* pFileName;
CFile f;
if( !f.Open( pFileName, CFile::modeCreate | CFile::modeWrite ) ) {
   afxDump << "Unable to open file" << "\n";
   exit( 1 );
}
CDumpContext dc( &f );
```

CDumpContext::Flush

void Flush( )
throw(CFileException);

Remarks

Forces any data remaining in buffers to be written to the file attached to the dump context.

Example

```
afxDump.Flush();
```

CDumpContext::GetDepth

int GetDepth( ) const;

Remarks Determines if a deep or shallow dump is in process.

Return Value The depth of the dump as set by **SetDepth**.

See Also **CDumpContext::SetDepth**

Example See the example for **SetDepth.**

CDumpContext::HexDump

void HexDump(const char* *pszLine*, **BYTE*** *pby*, **int** *nBytes*, **int** *nWidth* **)**
throw(CFileException);

pszLine A string to output at the start of a new line.

pby A pointer to a buffer containing the bytes to dump.

nBytes The number of bytes to dump.

nWidth Maximum number of bytes dumped per line (not the width of the output line).

Remarks Dumps an array of bytes formatted as hexadecimal numbers.

Example

```
char test[] = "This is a test of CDumpContext::HexDump\n";
afxDump.HexDump( ".", (BYTE*) test, sizeof test, 20 );
```

The output from this program is:

```
. 54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 6F 66 20 43 44
. 75 6D 70 43 6F 6E 74 65 78 74 3A 3A 48 65 78 44 75 6D 70 0A
. 00
```

CDumpContext::SetDepth

void SetDepth(int *nNewDepth* **);**

nNewDepth The new depth value.

Remarks

Sets the depth for the dump. If you are dumping a primitive type or simple **CObject** that contains no pointers to other objects, then a value of 0 is sufficient. A value greater than 0 specifies a deep dump where all objects are dumped recursively. For example, a deep dump of a collection will dump all elements of the collection. You may use other specific depth values in your derived classes.

Note Circular references are not detected in deep dumps and can result in infinite loops.

See Also

CObject::Dump

Example

```
afxDump.SetDepth( 1 );  // Specifies deep dump
ASSERT( afxDump.GetDepth() == 1 );
```

Operators

CDumpContext::operator <<

CDumpContext& operator <<(const CObject* *pOb* **)**
throw(CFileException);

CDumpContext& operator <<(const char FAR* *lpsz* **)**
throw(CFileException);

CDumpContext& operator <<(const void FAR* *lp* **)**
throw(CFileException);

CDumpContext& operator <<(const void NEAR* *np* **)**
throw(CFileException);

CDumpContext& operator <<(BYTE *by* **)**
throw(CFileException);

CDumpContext& operator <<(WORD *w* **)**
throw(CFileException);

CDumpContext& operator <<(DWORD *dw* **)**
throw(CFileException);

CDumpContext& operator <<(int *n* **)**
throw(CFileException);

CDumpContext& operator <<(LONG *l* **)**
throw(CFileException);

CDumpContext& operator <<(UINT *n* **)**
throw(CFileException);

Remarks

Outputs the specified data to the dump context. The insertion operator is overloaded for **CObject** pointers as well as for most primitive types. A pointer to **char** results in a dump of string contents; a pointer to **void** results in a hexadecimal dump of the address only.

If you use the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macros in the implementation of your class, then the insertion operator, through **CObject::Dump**, will print the name of your **CObject**-derived class. Otherwise, it will print `CObject`. If you override the **Dump** function of the class, then you can provide a more meaningful output of the object's contents instead of a hexadecimal dump.

Return Value

A **CDumpContext** reference that enables multiple insertions on a single line.

Example

```
extern CObList li;
CString s = "test";
int i = 7;
long lo = 1000000000L;
afxDump << "list=" << &li << "string="
        << s << "int=" << i << "long=" << lo << "\n";
```

class CDWordArray : public CObject

The **CDWordArray** class supports arrays of 32-bit doublewords. The member functions of **CDWordArray** are similar to the member functions of class **CObArray**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **DWORD**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
DWORD CDWordArray::GetAt( int <nIndex> ) const;
```

CDWordArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of doublewords is stored to an archive, either with the overloaded insertion (<<) operator or with the **Serialize** member function, each element is, in turn, serialized. If you need debug output from individual elements in the array, you must set the depth of the **CDumpContext** object to 1 or greater.

#include <afxcoll.h>

See Also **CObArray**

Construction/Destruction—Public Members

CDWordArray	Constructs an empty array for doublewords.
~CDWordArray	Destroys a **CDWordArray** object.

Bounds—Public Members

GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations—Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; array not allowed to grow.
ElementAt	Returns a temporary reference to the doubleword within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

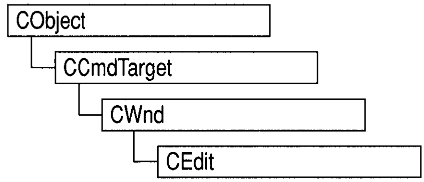
Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

class CEdit : public CWnd



The **CEdit** class provides the functionality of a Windows edit control. An edit control is a rectangular child window in which the user can enter text.

You can create an edit control either from a dialog template or directly in your code. In both cases, first call the constructor **CEdit** to construct the **CEdit** object, then call the **Create** member function to create the Windows edit control and attach it to the **CEdit** object. Construction can be a one-step process in a class derived from **CEdit**. Write a constructor for the derived class and call **Create** from within the constructor.

CEdit inherits significant functionality from **CWnd**. To set and retrieve text from a **CEdit** object, use the **CWnd** member functions **SetWindowText** and **GetWindowText**, which set or get the entire contents of an edit control, even if it is a multiline control. Also, if an edit control is multiline, get and set part of the control's text by calling the **CWnd** member functions **GetLine**, **SetSel**, **GetSel**, and **ReplaceSel**.

If you want to handle Windows notification messages sent by an edit control to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification(*id*, *memberFxn*)

where *id* specifies the child window ID of the edit control sending the notification, and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void memberFxn( );

Following is a list of potential message-map entries and a description of the cases in which they would be sent to the parent:

- **ON_EN_CHANGE** The user has taken an action that may have altered text in an edit control. Unlike the **EN_UPDATE** notification message, this notification message is sent after Windows updates the display.
- **ON_EN_ERRSPACE** The edit control cannot allocate enough memory to meet a specific request.

- **ON_EN_HSCROLL** The user clicks an edit control's horizontal scroll bar. The parent window is notified before the screen is updated.
- **ON_EN_KILLFOCUS** The edit control loses the input focus.
- **ON_EN_MAXTEXT** The current insertion has exceeded the specified number of characters for the edit control and has been truncated. Also sent when an edit control does not have the **ES_AUTOHSCROLL** style and the number of characters to be inserted would exceed the width of the edit control. Also sent when an edit control does not have the **ES_AUTOVSCROLL** style and the total number of lines resulting from a text insertion would exceed the height of the edit control.
- **ON_EN_SETFOCUS** Sent when an edit control receives the input focus.
- **ON_EN_UPDATE** The edit control is about to display altered text. Sent after the control has formatted the text but before it screens the text so that the window size can be altered, if necessary.
- **ON_EN_VSCROLL** The user clicks an edit control's vertical scroll bar. The parent window is notified before the screen is updated.

If you create a **CEdit** object within a dialog box, the **CEdit** object is automatically destroyed when the user closes the dialog box.

If you create a **CEdit** object from a dialog resource using App Studio, the **CEdit** object is automatically destroyed when the user closes the dialog box. If you create a **CEdit** object within a window, you may also need to destroy it. If you create the **CEdit** object on the stack, it is destroyed automatically. If you create the **CEdit** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user terminates the Windows edit control. If you allocate any memory in the **CEdit** object, override the **CEdit** destructor to dispose of the allocations.

#include <afxwin.h>

See Also **CWnd**, **CButton**, **CComboBox**, **CListBox**, **CScrollBar**, **CStatic**, **CDialog**

Construction/Destruction—Public Members

CEdit	Constructs a **CEdit** control object.

Initialization—Public Members

Create	Creates the Windows edit control and attaches it to the **CEdit** object.

Multiple-Line Operations — Public Members

GetLineCount	Retrieves the number of lines in a multiple-line edit control.
GetHandle	Retrieves a handle to the memory currently allocated for a multiple-line edit control.
SetHandle	Sets the handle to the local memory that will be used by a multiple-line edit control.
FmtLines	Sets the inclusion of soft line-break characters on or off within a multiple-line edit control.
LineIndex	Retrieves the character index of a line within a multiple-line edit control.
SetRect	Sets the formatting rectangle of a multiple-line edit control and updates the control.
SetRectNP	Sets the formatting rectangle of a multiple-line edit control without redrawing the control window.
SetTabStops	Sets the tab stops in a multiple-line edit control.

General Operations — Public Members

CanUndo	Determines if an edit-control operation can be undone.
GetModify	Determines if the contents of an edit control have been modified.
SetModify	Sets or clears the modification flag for an edit control.
SetReadOnly	Sets the read-only state of an edit control.
GetPasswordChar	Retrieves the password character displayed in an edit control when the user enters text.
GetRect	Gets the formatting rectangle of an edit control.
GetSel	Gets the starting and ending character positions of the current selection in an edit control.
GetLine	Retrieves a line of text from an edit control.
GetFirstVisibleLine	Determines the topmost visible line in an edit control.
EmptyUndoBuffer	Resets (clears) the undo flag of an edit control.
LimitText	Limits the length of the text that the user may enter into an edit control.

LineFromChar	Retrieves the line number of the line that contains the specified character index.
LineLength	Retrieves the length of a line in an edit control.
LineScroll	Scrolls the text of a multiple-line edit control.
ReplaceSel	Replaces the current selection in an edit control with the specified text.
SetPasswordChar	Sets or removes a password character displayed in an edit control when the user enters text.
SetSel	Selects a range of characters in an edit control.
Undo	Reverses the last edit-control operation.
Clear	Deletes (clears) the current selection (if any) in the edit control.
Copy	Copies the current selection (if any) in the edit control to the Clipboard in **CF_TEXT** format.
Cut	Deletes (cuts) the current selection (if any) in the edit control and copies the deleted text to the Clipboard in **CF_TEXT** format.
Paste	Inserts the data from the Clipboard into the edit control at the current cursor position. Data is inserted only if the Clipboard contains data in **CF_TEXT** format.

Member Functions

CEdit::CanUndo

BOOL CanUndo( ) const;

Return Value Nonzero if the last edit operation can be undone by a call to the **Undo** member function; 0 if it cannot be undone.

See Also **CEdit::Undo, EM_CANUNDO**

CEdit::CEdit

CEdit( );

Remarks Constructs a **CEdit** object.

See Also **CEdit::Create**

CEdit::Clear

void Clear( );

Remarks Deletes (clears) the current selection (if any) in the edit control. The deletion performed by **Clear** can be undone by calling the **Undo** member function. To delete the current selection and place the deleted contents into the Clipboard, call the **Cut** member function.

See Also **CEdit::CanUndo**, **CEdit::Undo**, **CEdit::Copy**, **CEdit::Cut**, **CEdit::Paste**, **WM_CLEAR**

CEdit::Copy

void Copy( );

Remarks Copies the current selection (if any) in the edit control to the Clipboard in **CF_TEXT** format.

See Also **CEdit::Clear**, **CEdit::Cut**, **CEdit::Paste**, **WM_COPY**

CEdit::Create

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

dwStyle Specifies the edit control's style.

rect Specifies the edit control's size and position. Can be a **CRect** object or **RECT** structure.

pParentWnd Specifies the edit control's parent window (usually a **CDialog** or **CModalDialog**). It must not be **NULL**.

nID Specifies the edit control's ID.

Remarks

You construct a **CEdit** object in two steps. First, call the **CEdit** constructor, then call **Create**, which creates the Windows edit control and attaches it to the **CEdit** object. When **Create** executes, Windows sends the **WM_NCCREATE**, **WM_NCCALCSIZE**, **WM_CREATE**, and **WM_GETMINMAXINFO** messages to the edit control. These messages are handled by default by the **OnNcCreate**, **OnNcCalcSize**, **OnCreate**, and **OnGetMinMaxInfo** member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CEdit**, add a message map to the new class, and override the above message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for the new class.

Apply the following window styles to an edit control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_GROUP** To group controls
- **WS_TABSTOP** To include edit control in the tabbing order

See **Create** in the **CWnd** base class for a full description of these window styles.

Return Value

Create returns nonzero if initialization is successful; 0 if unsuccessful.

Edit Styles

You can use any combination of the following edit-control styles for *dwStyle*:

- **ES_AUTOHSCROLL** Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position 0.
- **ES_AUTOVSCROLL** Automatically scrolls text up one page when the user presses ENTER on the last line.
- **ES_CENTER** Centers text in a multiline edit control.
- **ES_LEFT** Aligns text flush left.
- **ES_LOWERCASE** Converts all characters to lowercase as they are typed into the edit control.

- **ES_MULTILINE** Designates a multiple-line edit control. (The default is single line.) If the **ES_AUTOVSCROLL** style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If **ES_AUTOVSCROLL** is not given, the edit control shows as many lines as possible and beeps if ENTER is pressed when no more lines can be displayed. If the **ES_AUTOHSCROLL** style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press ENTER. If **ES_AUTOHSCROLL** is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if ENTER is pressed. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above and process any scroll messages sent by the parent window.
- **ES_NOHIDESEL** Normally, an edit control hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. Specifying **ES_NOHIDESEL** deletes this default action.
- **ES_OEMCONVERT** Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** Windows function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
- **ES_PASSWORD** Displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the **SetPasswordChar** member function to change the character that is displayed.
- **ES_RIGHT** Aligns text flush right in a multiline edit control.
- **ES_UPPERCASE** Converts all characters to uppercase as they are typed into the edit control.

Windows 3.1 Only

- **ES_READONLY** Prevents the user from entering or editing text in the edit control.
- **ES_WANTRETURN** Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiple-line edit control in a dialog box. Without this style, pressing the ENTER key has the same effect as pressing the dialog box's default pushbutton. This style has no effect on a single-line edit control. ♦

See Also **CEdit::CEdit**

CEdit::Cut

void Cut( );

Remarks

Deletes (cuts) the current selection (if any) in the edit control and copies the deleted text to the Clipboard in **CF_TEXT** format. The deletion performed by **Cut** can be undone by calling the **Undo** member function. To delete the current selection without placing the deleted text into the Clipboard, call the **Clear** member function.

See Also

CEdit::Undo, **CEdit::Clear**, **CEdit::Copy**, **CEdit::Paste**, **WM_CUT**

CEdit::EmptyUndoBuffer

void EmptyUndoBuffer( );

Remarks

Resets (clears) the undo flag of an edit control. The edit control will now be unable to undo the last operation. The undo flag is set whenever an operation within the edit control can be undone. The undo flag is automatically cleared whenever the **SetWindowText** or **SetHandle** member function is called.

See Also

CEdit::CanUndo, **CEdit::SetHandle**, **CEdit::Undo**, **CWnd::SetWindowText**, **EM_EMPTYUNDOBUFFER**

CEdit::FmtLines

BOOL FmtLines(BOOL *bAddEOL* **);**

bAddEOL Specifies whether soft line-break characters are to be inserted. A value of **TRUE** inserts the characters; a value of **FALSE** removes them.

Remarks

Sets the inclusion of soft line-break characters on or off within a multiple-line edit control. A soft line break consists of two carriage returns and a linefeed inserted at the end of a line that is broken because of word wrapping. A hard line break consists of one carriage return and a linefeed. Lines that end with a hard line break are not affected by **FmtLines**. Windows will only respond if the **CEdit** object is a multiple-line edit control. **FmtLines** only affects the buffer returned by **GetHandle** and the text returned by **WM_GETTEXT**. It has no impact on the display of the text within the edit control.

Return Value Nonzero if any formatting occurs; otherwise 0.

See Also **CEdit::GetHandle**, **CWnd::GetWindowText**, **EM_FMTLINES**

CEdit::GetFirstVisibleLine

Windows 3.1 Only **int GetFirstVisibleLine( ) const;** ♦

Remarks An application calls **GetFirstVisibleLine** to determine the topmost visible line in an edit control.

Return Value The zero-based index of the topmost visible line. For single-line edit controls, the return value is 0.

See Also **EM_GETFIRSTVISIBLELINE**

CEdit::GetHandle

HLOCAL GetHandle( ) const;

Remarks Retrieves a handle to the memory currently allocated for a multiple-line edit control. The handle is a local memory handle and may be used by any of the **Local** Windows memory functions that take a local memory handle as a parameter. **GetHandle** is processed only by multiple-line edit controls. Call **GetHandle** for a multiple-line edit control in a dialog box only if the dialog box was created with the **DS_LOCALEDIT** style flag set. If the **DS_LOCALEDIT** style is not set, you will still get a nonzero return value, but you will not be able to use the returned value.

Return Value A local memory handle that identifies the buffer holding the contents of the edit control. If an error occurs, such as sending the message to a single-line edit control, the return value is 0.

See Also **CEdit::SetHandle**, **EM_GETHANDLE**

CEdit::GetLine

int GetLine(int *nIndex*, **LPSTR** *lpszBuffer* **) const;**

int GetLine(int *nIndex*, **LPSTR** *lpszBuffer*, **int** *nMaxLength* **) const;**

nIndex Specifies the line number to retrieve from a multiple-line edit control. Line numbers are zero-based; a value of 0 specifies the first line. This parameter is ignored by a single-line edit control.

lpszBuffer Points to the buffer that receives a copy of the line. The first word of the buffer must specify the maximum number of bytes that can be copied to the buffer.

nMaxLength Specifies the maximum number of bytes that can be copied to the buffer. **GetLine** places this value in the first word of *lpszBuffer* before making the call to Windows.

Remarks Retrieves a line of text from an edit control and places it in *lpszBuffer*. This call is not processed for a single-line edit control. The copied line does not contain a null-termination character.

Return Value The number of bytes actually copied. The return value is 0 if the line number specified by *nIndex* is greater then the number of lines in the edit control.

See Also **CEdit::LineLength**, **CWnd::GetWindowText**, **EM_GETLINE**

CEdit::GetLineCount

int GetLineCount( ) const;

Remarks Retrieves the number of lines in a multiple-line edit control. **GetLineCount** is only processed by multiple-line edit controls.

Return Value An integer containing the number of lines in the multiple-line edit control. If no text has been entered into the edit control, the return value is 1.

See Also **EM_GETLINECOUNT**

CEdit::GetModify

BOOL GetModify( ) const;

Remarks Determines if the contents of an edit control have been modified. Windows maintains an internal flag indicating whether the contents of the edit control have been changed. This flag is cleared when the edit control is first created and may also be cleared by calling the **SetModify** member function.

Return Value Nonzero if the edit-control contents have been modified; 0 if they have remained unchanged.

See Also **CEdit::SetModify, EM_GETMODIFY**

CEdit::GetPasswordChar

Windows 3.1 Only **char GetPasswordChar( ) const;** ♦

Remarks An application calls the **GetPasswordChar** member function to retrieve the password character displayed in an edit control when the user enters text. If the edit control is created with the **ES_PASSWORD** style, the default password character is set to an asterisk (*).

Return Value Specifies the character to be displayed in place of the character typed by the user. The return value is **NULL** if no password character exists.

See Also **EM_GETPASSWORDCHAR, CEdit::SetPasswordChar**

CEdit::GetRect

void GetRect(LPRECT *lpRect* **) const;**

lpRect Points to the **RECT** structure that receives the formatting rectangle.

Remarks Gets the formatting rectangle of an edit control. The formatting rectangle is the limiting rectangle of the text, which is independent of the size of the edit-control window. The formatting rectangle of a multiple-line edit control can be modified by the **SetRect** and **SetRectNP** member functions.

See Also **CEdit::SetRect, CEdit::SetRectNP, EM_GETRECT**

CEdit::GetSel

DWORD GetSel( ) const;

void GetSel(int& *nStartChar*, **int&** *nEndChar* **) const;**

nStartChar Reference to an integer that will receive the position of the first character in the current selection.

nEndChar Reference to an integer that will receive the position of the first nonselected character past the end of the current selection.

Remarks Gets the starting and ending character positions of the current selection (if any) in an edit control, using either the return value or the parameters.

Return Value The version that returns a **DWORD** returns a value that contains the starting position in the low-order word and the position of the first nonselected character after the end of the selection in the high-order word.

See Also **CEdit::SetSel, EM_GETSEL**

CEdit::LimitText

void LimitText(int *nChars* = **0);**

nChars Specifies the length (in bytes) of the text that the user can enter. If this parameter is 0, the text length is set to **UINT_MAX** bytes. This is the default behavior.

Remarks Limits the length of the text that the user may enter into an edit control. **LimitText** limits only the text the user can enter. It has no effect on any text already in the edit control when the message is sent, nor does it affect the length of the text copied to the edit control by the **SetWindowText** member function in **CWnd**. If an application uses the **SetWindowText** function to place more text into an edit control than is specified in the call to **LimitText**, the user can edit the entire contents of the edit control.

See Also **CWnd::SetWindowText, EM_LIMITTEXT**

CEdit::LineFromChar

int LineFromChar(int *nIndex* **= –1) const;**

nIndex Contains the zero-based index value for the desired character in the text of the edit control, or contains –1. If *nIndex* is –1, it specifies the current line, that is, the line that contains the caret.

Remarks Retrieves the line number of the line that contains the specified character index. A character index is the number of characters from the beginning of the edit control. This member function is only used by multiple-line edit controls.

Return Value The zero-based line number of the line containing the character index specified by *nIndex*. If *nIndex* is –1, the number of the line that contains the first character of the selection is returned. If there is no selection, the current line number is returned.

See Also **CEdit::LineIndex**, **EM_LINEFROMCHAR**

CEdit::LineIndex

int LineIndex(int *nLine* **= –1) const;**

nLine Contains the index value for the desired line in the text of the edit control, or contains –1. If *nLine* is –1, it specifies the current line, that is, the line that contains the caret.

Remarks Retrieves the character index of a line within a multiple-line edit control. The character index is the number of characters from the beginning of the edit control to the specified line. This member function is only processed by multiple-line edit controls.

Return Value The character index of the line specified in *nLine* or –1 if the specified line number is greater than the number of lines in the edit control.

See Also **CEdit::LineFromChar**, **EM_LINEINDEX**

CEdit::LineLength

int LineLength(int *nLine* **= –1) const;**

nLine Specifies the character index of a character in the line whose length is to be retrieved. If this parameter is –1, the length of the current line (the line that contains the caret) is returned, not including the length of any selected text within the line. When **LineLength** is called for a single-line edit control, this parameter is ignored.

Remarks

Retrieves the length of a line in an edit control. Use the **LineIndex** member function to retrieve a character index for a given line number within a multiple-line edit control.

Return Value

When **LineLength** is called for a multiple-line edit control, the return value is the length (in bytes) of the line specified by *nLine*. When **LineLength** is called for a single-line edit control, the return value is the length (in bytes) of the text in the edit control.

See Also

CEdit::LineIndex, **EM_LINELENGTH**

CEdit::LineScroll

void LineScroll(int *nLines*, **int** *nChars* **= 0);**

nLines Specifies the number of lines to scroll vertically.

nChars Specifies the number of character positions to scroll horizontally. This value is ignored if the edit control has either the **ES_RIGHT** or **ES_CENTER** style.

Remarks

Scrolls the text of a multiple-line edit control. This member function is processed only by multiple-line edit controls. The edit control does not scroll vertically past the last line of text in the edit control. If the current line plus the number of lines specified by *nLines* exceeds the total number of lines in the edit control, the value is adjusted so that the last line of the edit control is scrolled to the top of the edit-control window. **LineScroll** can be used to scroll horizontally past the last character of any line.

See Also

EM_LINESCROLL

CEdit::Paste

void Paste( );

Remarks Inserts the data from the Clipboard into the edit control at the current cursor position. Data is inserted only if the Clipboard contains data in **CF_TEXT** format.

See Also **CEdit::Clear**, **CEdit::Copy**, **CEdit::Cut**, **WM_PASTE**

CEdit::ReplaceSel

void ReplaceSel(LPCSTR *lpszNewText* **);**

lpszNewText Points to a null-terminated string containing the replacement text.

Remarks Replaces the current selection in an edit control with the text specified by *lpszNewText*. Replaces only a portion of the text in an edit control. If you want to replace all of the text, use the **CWnd::SetWindowText** member function. If there is no current selection, the replacement text is inserted at the current cursor location.

See Also **CWnd::SetWindowText**, **EM_REPLACESEL**

CEdit::SetHandle

void SetHandle(HLOCAL *hBuffer* **);**

hBuffer Contains a handle to the local memory. This handle must have been created by a previous call to the **LocalAlloc** Windows function using the **LMEM_MOVEABLE** flag. The memory is assumed to contain a null-terminated string. If this is not the case, the first byte of the allocated memory should be set to 0.

Remarks Sets the handle to the local memory that will be used by a multiple-line edit control. The edit control will then use this buffer to store the currently displayed text instead of allocating its own buffer. This member function is processed only by multiple-line edit controls. Before an application sets a new memory handle, it should use the **GetHandle** member function to get the handle to the current memory buffer and free that memory using the **LocalFree** Windows function. **SetHandle** clears the undo buffer (the **CanUndo** member function then returns 0) and the internal modification flag (the **GetModify** member function then returns 0). The edit-control window is redrawn. You can use this member function in a multiple-line edit control in a dialog box only if you have created the dialog box with the **DS_LOCALEDIT** style flag set.

See Also **CEdit::CanUndo**, **CEdit::GetHandle**, **CEdit::GetModify**, **::LocalAlloc**, **::LocalFree**, **EM_SETHANDLE**

CEdit::SetModify

void SetModify(BOOL *bModified* **= TRUE);**

bModified A value of **TRUE** indicates that the text has been modified, and a value of **FALSE** indicates it is unmodified. By default, the modified flag is set.

Remarks Sets or clears the modified flag for an edit control. The modified flag indicates whether or not the text within the edit control has been modified. It is automatically set whenever the user changes the text. Its value may be retrieved with the **GetModify** member function.

See Also **CEdit::GetModify**, **EM_SETMODIFY**

CEdit::SetPasswordChar

void SetPasswordChar(char *ch* **);**

ch Specifies the character to be displayed in place of the character typed by the user. If *ch* is 0, the actual characters typed by the user are displayed.

Remarks Sets or removes a password character displayed in an edit control when the user types text. When a password character is set, that character is displayed for each character the user types. This member function has no effect on a multiple-line edit control. When the **SetPasswordChar** member function is called, **CEdit** will redraw all visible characters using the character specified by *ch*. If the edit control is created with the **ES_PASSWORD** style, the default password character is set to an asterisk (*). This style is removed if **SetPasswordChar** is called with *ch* set to 0.

See Also **CEdit::GetPasswordChar**, **EM_SETPASSWORDCHAR**

CEdit::SetReadOnly

Windows 3.1 Only **BOOL SetReadOnly(BOOL** *bReadOnly* **= TRUE);** ♦

bReadOnly Specifies whether to set or remove the read-only state of the edit control. A value of **TRUE** sets the state to read-only; a value of **FALSE** sets the state to read/write.

Remarks An application calls the **SetReadOnly** member function to set the read-only state of an edit control. The current setting can be found by testing the **ES_READONLY** flag in the return value of **CWnd::GetStyle**.

Return Value Nonzero if the operation is successful, or 0 if an error occurs.

See Also **EM_SETREADONLY**, **CWnd::GetStyle**

CEdit::SetRect

void SetRect(LPCRECT *lpRect* **);**

lpRect Points to the **RECT** structure or **CRect** object that specifies the new dimensions of the formatting rectangle.

Remarks Sets the dimensions of a rectangle using the specified coordinates. This member is processed only by multiple-line edit controls. Use **SetRect** to set the formatting rectangle of a multiple-line edit control. The formatting rectangle is the limiting rectangle of the text, which is independent of the size of the edit-control window. When the edit control is first created, the formatting rectangle is the same as the client area of the edit-control window. By using the **SetRect** member function, an application can make the formatting rectangle larger or smaller than the edit-control window. If the edit control has no scroll bar, text will be clipped, not wrapped, if the formatting rectangle is made larger than the window. If the edit control contains a border, the formatting rectangle is reduced by the size of the border. If you adjust the rectangle returned by the **GetRect** member function, you must remove the size of the border before you pass the rectangle to **SetRect**. When **SetRect** is called, the edit control's text is also reformatted and redisplayed.

See Also **CRect::CRect**, **CRect::CopyRect**, **CRect::operator =**, **CRect::SetRectEmpty**, **CEdit::GetRect**, **CEdit::SetRectNP**, **EM_SETRECT**

CEdit::SetRectNP

void SetRectNP(LPCRECT *lpRect* **);**

lpRect Points to a **RECT** structure or **CRect** object that specifies the new dimensions of the rectangle.

Remarks

Sets the formatting rectangle of a multiple-line edit control. The formatting rectangle is the limiting rectangle of the text, which is independent of the size of the edit-control window. **SetRectNP** is identical to the **SetRect** member function except that the edit-control window is not redrawn. When the edit control is first created, the formatting rectangle is the same as the client area of the edit-control window. By calling the **SetRectNP** member function, an application can make the formatting rectangle larger or smaller than the edit-control window. If the edit control has no scroll bar, text will be clipped, not wrapped, if the formatting rectangle is made larger than the window. This member is processed only by multiple-line edit controls.

See Also

CRect::CRect, **CRect::CopyRect**, **CRect::operator =**, **CRect::SetRectEmpty**, **CEdit::GetRect**, **CEdit::SetRect**, **EM_SETRECTNP**

CEdit::SetSel

void SetSel(DWORD *dwSelection*, **BOOL** *bNoScroll* **= FALSE);**

void SetSel(int *nStartChar*, **int** *nEndChar*, **BOOL** *bNoScroll* **= FALSE);**

dwSelection Specifies the starting position in the low-order word and the ending position in the high-order word. If the low-order word is 0 and the high-order word is –1, all the text in the edit control is selected. If the low-order word is –1, any current selection is removed.

Windows 3.1 Only

bNoScroll Indicates whether the caret should be scrolled into view. If **FALSE**, the caret is scrolled into view. If **TRUE**, the caret is not scrolled into view. ♦

nStartChar Specifies the starting position. If *nStartChar* is 0 and *nEndChar* is –1, all the text in the edit control is selected. If *nStartChar* is –1, any current selection is removed.

nEndChar Specifies the ending position.

Remarks Selects a range of characters in an edit control.

See Also **CEdit::GetSel**, **CEdit::ReplaceSel**, **EM_SETSEL**

CEdit::SetTabStops

void SetTabStops( );

BOOL SetTabStops(const int& *cxEachStop* **);**

BOOL SetTabStops(int *nTabStops*, **LPINT** *rgTabStops* **);**

cxEachStop Specifies that tab stops are to be set at every *cxEachStop* dialog units.

nTabStops Specifies the number of tab stops contained in *rgTabStops*. This number must be greater than 1.

rgTabStops Points to an array of unsigned integers specifying the tab stops in dialog units. A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit, and 1 vertical dialog unit is equal to one-eighth of the current dialog base height unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** Windows function returns the current dialog base units in pixels.

Remarks Sets the tab stops in a multiple-line edit control. When text is copied to a multiple-line edit control, any tab character in the text will cause space to be generated up to the next tab stop.

To set tab stops to the default size of 32 dialog units, call the parameterless version of this member function. To set tab stops to a size other than 32, call the version with the *cxEachStop* parameter. To set tab stops to an array of sizes, use the version with two parameters. This member function is only processed by multiple-line edit controls. **SetTabStops** does not automatically redraw the edit window. If you change the tab stops for text already in the edit control, call **CWnd::InvalidateRect** to redraw the edit window.

Return Value Nonzero if the tabs were set; otherwise 0.

See Also **::GetDialogBaseUnits**, **CWnd::InvalidateRect**, **EM_SETTABSTOPS**

CEdit::Undo

BOOL Undo( );

Remarks

Use to undo the last edit-control operation. An undo operation can also be undone. For example, you can restore deleted text with the first call to **Undo**. As long as there is no intervening edit operation, you can remove the text again with a second call to **Undo**.

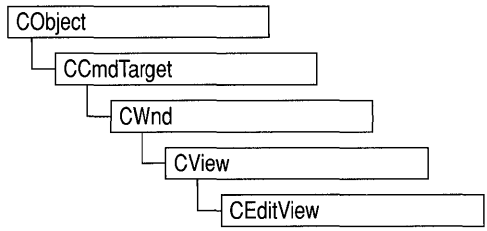
Return Value

For a single-line edit control, the return value is always nonzero. For a multiple-line edit control, the return value is nonzero if the undo operation is successful, or 0 if the undo operation fails.

See Also

CEdit::CanUndo, EM_UNDO

class CEditView : public CView



Like the **CEdit** class, the **CEditView** class provides the functionality of a Windows edit control. The **CEditView** class provides the following additional functions:

- Printing
- Find and replace
- Cut, copy, paste, clear, and undo

Because class **CEditView** is derived from class **CView**, objects of class **CEditView** can be used with documents and document templates.

Each **CEditView** control's text is kept in its own global memory object. Your application can have any number of **CEditView** controls.

Create objects of type **CEditView** if you want an edit control with the added functionality listed above. Derive your own classes from **CEditView** to add or modify the basic functionality, or to declare classes that can be added to a document template.

The default implementation of class **CEditView** handles the following commands: **ID_EDIT_CUT**, **ID_EDIT_COPY**, **ID_EDIT_PASTE**, **ID_EDIT_CLEAR**, **ID_EDIT_UNDO**, **ID_EDIT_SELECT_ALL**, **ID_EDIT_FIND**, **ID_EDIT_REPLACE**, **ID_EDIT_REPEAT**, and **ID_FILE_PRINT**.

Objects of type **CEditView** (or of types derived from **CEditView**) have the following limitations:

- **CEditView** does not implement true WYSIWYG (what you see is what you get) editing. Where there is a choice between readability on the screen and matching printed output, **CEditView** opts for screen readability.
- **CEditView** can display text in only a single font. No special character formatting is supported.
- The amount of text a **CEditView** can contain is limited. The limits are the same as for the **CEdit** control.

#include <afxext.h>

See Also **CEdit**, **CDocument**, **CDocTemplate**, **CView**

Data Members—Public Members

dwStyleDefault	Default style for objects of type **CEditView.**

Construction/Destruction—Public Members

CEditView	Constructs an object of type **CEditView**.

Attributes—Public Members

GetEditCtrl	Provides access to the **CEdit** portion of a **CEditView** object (the Windows edit control).
GetPrinterFont	Retrieves the current printer font.
GetSelectedText	Retrieves the current text selection.
SetPrinterFont	Sets a new printer font.
SetTabStops	Sets tab stops for both screen display and printing.

Operations—Public Members

FindText	Searches for a string within the text.
PrintInsideRect	Renders text inside a given rectangle.
SerializeRaw	Serializes a **CEditView** object to disk as raw text.

Overridables—Protected Members

OnFindNext	Finds next occurrence of a text string.
OnReplaceAll	Replaces all occurrences of a given string with a new string.
OnReplaceSel	Replaces current selection.
OnTextNotFound	Called when a find operation fails to match any further text.

Member Functions

CEditView::CEditView

CEditView( );

Remarks

Constructs an object of type **CEditView**. After constructing the object, you must call the **Create** function before the edit control is used. If you derive a class from

CEditView and add it to the template using **CWinApp::AddDocTemplate**, the framework calls both this constructor and the **Create** function.

See Also **CWnd::Create**, **CWinApp::AddDocTemplate**

CEditView::FindText

BOOL FindText(LPCSTR *lpszFind*, **BOOL** *bNext* = **TRUE,**
BOOL *bCase* = **TRUE);**

lpszFind The text to be found.

bNext Specifies the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

bCase Specifies whether the search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

Remarks Call the **FindText** function to search the **CEditView** object's text buffer. This function searches the text in the buffer for the text specified by *lpszFind*, starting at the current selection, in the direction specified by *bNext*, and with case sensitivity specified by *bCase*. If the text is found, it sets the selection to the found text and returns a nonzero value. If the text is not found, the function returns 0.

You normally do not need to call the **FindText** function unless you override **OnFindNext**, which calls **FindText**.

Return Value Nonzero if the search text is found; otherwise 0.

See Also **CEditView::OnFindNext**, **CEditView::OnReplaceAll**, **CEditView::OnReplaceSel**, **CEditView::OnTextNotFound**

CEditView::GetEditCtrl

CEdit& GetEditCtrl( ) const;

Remarks Call **GetEditCtrl** to get a reference to the edit control used by the edit view. This control is of type **CEdit**, so you can manipulate the Windows edit control directly using the **CEdit** member functions.

Warning Using the **CEdit** object can change the state of the underlying Windows edit control. For example, you should not change the tab settings using the **CEdit::SetTabStops** function because **CEditView** caches these settings for use both in the edit control and in printing. Instead, use **CEditView::SetTabStops**.

Return Value A reference to a **CEdit** object.

See Also **CEdit**, **CEditView::SetTabStops**

CEditView::GetPrinterFont

CFont* GetPrinterFont( ) const;

Remarks Call **GetPrinterFont** to get a pointer to a **CFont** object that describes the current printer font. If the printer font has not been set, the default printing behavior of the **CEditView** class is to print using the same font used for display.

Use this function to determine the current printer font. If it is not the desired printer font, use **CEditView::SetPrinterFont** to change it.

Return Value A pointer to a **CFont** object that specifies the current printer font; **NULL** if the printer font has not been set. The pointer may be temporary and should not be stored for later use.

See Also **CEditView::SetPrinterFont**

CEditView::GetSelectedText

void GetSelectedText(CString& *strResult* **) const;**

strResult A reference to the **CString** object that is to receive the selected text.

Remarks Call **GetSelectedText** to copy the selected text into a **CString** object, up to the end of the selection or the character preceding the first carriage-return character in the selection.

See Also **CEditView::OnReplaceSel**

CEditView::OnFindNext

Protected **virtual void OnFindNext(LPCSTR** *lpszFind*, **BOOL** *bNext*, **BOOL** *bCase* **);** ♦

lpszFind The text to be found.

bNext Specifies the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

bCase Specifies whether the search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

Remarks Searches the text in the buffer for the text specified by *lpszFind*, in the direction specified by *bNext*, with case sensitivity specified by *bCase*. The search starts at the beginning of the current selection and is accomplished through a call to **FindText**. In the default implementation, **OnFindNext** calls **OnTextNotFound** if the text is not found.

Override **OnFindNext** to change the way a **CEditView**-derived object searches text. **CEditView** calls **OnFindNext** when the user chooses the Find Next button in the standard Find dialog box.

See Also **CEditView::OnTextNotFound**, **CEditView::FindText**, **CEditView::OnReplaceAll**, **CEditView::OnReplaceSel**

CEditView::OnReplaceAll

Protected **virtual void OnReplaceAll(LPCSTR** *lpszFind*, **LPCSTR** *lpszReplace*, **BOOL** *bCase* **);** ♦

lpszFind The text to be found.

lpszReplace The text to replace the search text.

bCase Specifies whether search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

Remarks **CEditView** calls **OnReplaceAll** when the user selects the Replace All button in the standard Replace dialog box. **OnReplaceAll** searches the text in the buffer for the text specified by *lpszFind*, with case sensitivity specified by *bCase*. The search starts at the beginning of the current selection. Each time the search text is found, this function replaces that occurrence of the text with the text specified by *lpszReplace*. The search is accomplished through a call to **FindText**. In the default implementation, **OnTextNotFound** is called if the text is not found.

Override **OnReplaceAll** to change the way a **CEditView**-derived object replaces text.

See Also **CEditView::OnFindNext**, **CEditView::OnTextNotFound**, **CEditView::FindText**, **CEditView::OnReplaceSel**

CEditView::OnReplaceSel

Protected **virtual void OnReplaceSel(LPCSTR** *lpszFind*, **BOOL** *bNext*, **BOOL** *bCase*, **LPCSTR** *lpszReplace* **);** ♦

lpszFind The text to be found.

bNext Specifies the direction of the search. If **TRUE**, the search direction is toward the end of the buffer. If **FALSE**, the search direction is toward the beginning of the buffer.

bCase Specifies whether the search is case sensitive. If **TRUE**, the search is case sensitive. If **FALSE**, the search is not case sensitive.

lpszReplace The text to replace the found text.

Remarks **CEditView** calls **OnReplaceSel** when the user selects the Replace button in the standard Replace dialog box. After replacing the selection, this function searches the text in the buffer for the next occurrence of the text specified by *lpszFind*, in the direction specified by *bNext*, with case sensitivity specified by *bCase*. The search is accomplished through a call to **FindText**. If the text is not found, **OnTextNotFound** is called.

Override **OnReplaceSel** to change the way a **CEditView**-derived object replaces the selected text.

See Also **CEditView::OnFindNext**, **CEditView::OnTextNotFound**, **CEditView::FindText**, **CEditView::OnReplaceAll**

CEditView::OnTextNotFound

Protected **virtual void OnTextNotFound(LPCSTR** *lpszFind* **);** ♦

lpszFind The text to be found.

Remarks Override this function to change the default implementation, which calls the Windows function **MessageBeep**.

See Also **CEditView::FindText**, **CEditView::OnFindNext**, **CEditView::OnReplaceAll**, **CEditView::OnReplaceSel**

CEditView::PrintInsideRect

UINT PrintInsideRect(CDC **pDC*, **RECT&** *rectLayout*, **UINT** *nIndexStart*, **UINT** *nIndexStop* **);**

pDC Pointer to the printer device context.

rectLayout Reference to a **CRect** object or **RECT** structure specifying the rectangle in which the text is to be rendered.

nIndexStart Index within the buffer of the first character to be rendered.

nIndexStop Index within the buffer of the character following the last character to be rendered.

Remarks Call **PrintInsideRect** to print text in the rectangle specified by *rectLayout*.

If the **CEditView** control does not have the style **ES_AUTOHSCROLL**, text is wrapped within the rendering rectangle. If the control does have the style **ES_AUTOHSCROLL**, the text is clipped at the right edge of the rectangle.

The **rect.bottom** element of the *rectLayout* object is changed so that the rectangle's dimensions define the part of the original rectangle that is occupied by the text.

Return Value The index of the next character to be printed (i.e., the character following the last character rendered).

See Also **CEditView::SetPrinterFont**, **CEditView::GetPrinterFont**

CEditView::SerializeRaw

void SerializeRaw(CArchive& *ar* **);**

ar Reference to the **CArchive** object that stores the serialized text.

Remarks

Call **SerializeRaw** to have a **CArchive** object read or write the text in the **CEditView** object to a text file. **SerializeRaw** differs from **CEditView**'s internal implementation of **Serialize** in that it reads and writes only the text, without preceding object-description data.

See Also

CArchive, **CObject::Serialize**

CEditView::SetPrinterFont

void SetPrinterFont(CFont* *pFont* **);**

pFont A pointer to an object of type **CFont**. If **NULL**, the font used for printing is based on the display font.

Remarks

Call **SetPrinterFont** to set the printer font to the font specified by *pFont*.

If you want your view to always use a particular font for printing, include a call to **SetPrinterFont** in your class's **OnPreparePrinting** function. This virtual function is called before printing occurs, so the font change takes place before the view's contents are printed.

See Also

CWnd::SetFont, **CFont**, **CView::OnPreparePrinting**

CEditView::SetTabStops

void SetTabStops(int *nTabStops* **);**

nTabStops Width of each tab stop, in dialog units.

Remarks

Call this function to set the tab stops used for display and printing. Only a single tab-stop width is supported. (**CEdit** objects support multiple tab widths.) Widths are in dialog units, which equal one-fourth of the average character width (based on uppercase and lowercase alphabetic characters only) of the font used at the time of printing or displaying. You should not use **CEdit::SetTabStops** because **CEditView** must cache the tab-stop value.

This function modifies only the tabs of the object for which it is called. To change the tab stops for each **CEditView** object in your application, call each object's **SetTabStops** function. **dwStyleDefault** is a public member variable of type **DWORD**.

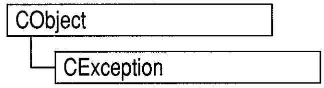
See Also **CWnd::SetFont**, **CEditView::SetPrinterFont**

Data Members

CEditView::dwStyleDefault

Remarks Pass this static member as the *dwStyle* parameter of the **Create** function to obtain the default style for the **CEditView** object. **dwStyleDefault** is a public member of type **DWORD**.

class CException : public CObject



CException is the base class for all exceptions in the Microsoft Foundation Class Library. The derived classes and their descriptions are listed below:

Class	Description
CMemoryException	Out-of-memory exception
CNotSupportedException	Request for an unsupported operation
CArchiveException	Archive-specific exceptions
CFileException	File-specific exceptions
CResourceException	Windows resource not found or not creatable
COleException	OLE (Object Linking and Embedding) exception

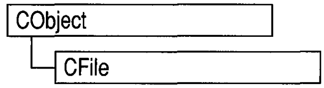
These exceptions are intended to be used with the **THROW**, **THROW_LAST**, **TRY**, **CATCH**, **AND_CATCH**, and **END_CATCH** macros. For more information on exceptions, see Chapter 16, "Exceptions," in the *Class Library User's Guide*.

Use the derived classes to catch specific exceptions. Use **CException** if you need to catch all types of exceptions (and then use **CObject::IsKindOf** to differentiate among **CException**-derived classes). All derived **CException** classes use the **IMPLEMENT_DYNAMIC** macro. **CException** objects are deleted automatically. Do not delete them yourself.

Because **CException** is an abstract base class, you cannot create **CException** objects; you must create objects of derived classes. If you need to create your own **CException** type, use one of the derived classes listed above as a model.

#include <afx.h>

class CFile : public CObject



CFile is the base class for Microsoft Foundation file classes. It directly provides unbuffered, binary disk input/output services, and it indirectly supports text files and memory files through its derived classes. **CFile** works in conjunction with the **CArchive** class to support serialization of Microsoft Foundation objects. The hierarchical relationship between this class and its derived classes allows your program to operate on all file objects through the polymorphic **CFile** interface. A memory file, for example, behaves like a disk file. Use **CFile** and its derived classes for general-purpose disk I/O. Use **ofstream** or other Microsoft iostream classes for formatted text sent to a disk file. Normally, a disk file is opened automatically on **CFile** construction and closed on destruction. Static member functions permit you to interrogate a file's status without opening the file.

#include <afx.h>

See Also **CStdioFile**, **CMemFile**

Data Members—Public Members

m_hFile	Usually contains the operating-system file handle.

Construction/Destruction—Public Members

CFile	Constructs a **CFile** object from a path or file handle.
Duplicate	Constructs a duplicate object based on this file.
Open	Safely opens a file with an error-testing option.
Close	Closes a file and deletes the object.

Input/Output—Public Members

Read	Reads (unbuffered) data from a file at the current file position.
Write	Writes (unbuffered) data in a file to the current file position.
Flush	Flushes any data yet to be written.

Position—Public Members

Seek	Positions the current file pointer.
SeekToBegin	Positions the current file pointer at the beginning of the file.
SeekToEnd	Positions the current file pointer at the end of the file.
GetLength	Obtains the length of the file.
SetLength	Changes the length of the file.

Locking—Public Members

LockRange	Locks a range of bytes in a file.
UnlockRange	Unlocks a range of bytes in a file.

Status—Public Members

GetPosition	Gets the current file pointer.
GetStatus	Obtains the status of this open file.

Static—Public Members

Rename	Renames the specified file (static function).
Remove	Deletes the specified file (static function).
GetStatus	Obtains the status of the specified file (static, virtual function).
SetStatus	Sets the status of the specified file (static, virtual function).

Member Functions

CFile::CFile

CFile( );

CFile(int *hFile* **);**

CFile(const char* *pszFileName*, **UINT** *nOpenFlags* **)**
throw(CFileException);

hFile The handle of a file that is already open.

pszFileName A string that is the path to the desired file. The path may be relative or absolute.

nOpenFlags Sharing and access mode. Specifies the action to take when opening the file. You can combine options listed below by using the bitwise-OR (|) operator. One access permission and one share option are required; the **modeCreate** and **modeNoInherit** modes are optional. The values and meanings are given below:

- **CFile::modeCreate** Directs the constructor to create a new file. If the file exists already, it is truncated to 0 length.

- **CFile::modeRead** Opens the file for reading only.
- **CFile::modeReadWrite** Opens the file for reading and writing.
- **CFile::modeWrite** Opens the file for writing only.
- **CFile::modeNoInherit** Prevents the file from being inherited by child processes.
- **CFile::shareDenyNone** Opens the file without denying other processes read or write access to the file. Create fails if the file has been opened in compatibility mode by any other process.
- **CFile::shareDenyRead** Opens the file and denies other processes read access to the file. Create fails if the file has been opened in compatibility mode or for read access by any other process.
- **CFile::shareDenyWrite** Opens the file and denies other processes write access to the file. Create fails if the file has been opened in compatibility mode or for write access by any other process.
- **CFile::shareExclusive** Opens the file with exclusive mode, denying other processes both read and write access to the file. Construction fails if the file has been opened in any other mode for read or write access, even by the current process.
- **CFile::shareCompat** Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. Construction fails if the file has been opened with any of the other sharing modes.
- **CFile::typeText** Sets text mode with special processing for carriage return– linefeed pairs (used in derived classes only).
- **CFile::typeBinary** Sets binary mode (used in derived classes only).

Remarks

The default constructor does not open a file but rather sets **m_hFile** to **CFile::hFileNull**. Because this constructor does not throw an exception, it does not make sense to use **TRY/CATCH** logic. Use the **Open** member function, then test directly for exception conditions. For a discussion of exception-processing strategy, see Chapter 16 in the *Class Library User's Guide*.

The constructor with one argument creates a **CFile** object that corresponds to an existing operating-system file identified by *hFile*. No check is made on the access mode or file type. When the **CFile** object is destroyed the operating-system file will not be closed. You must close the file yourself.

The constructor with two arguments creates a **CFile** object and opens the corresponding operating-system file with the given path. This constructor combines the functions of the first constructor and the **Open** member function. It throws an exception if there is an error while opening the file. Generally, this means that the error is unrecoverable and that the user should be alerted.

Example

```
char* pFileName = "test.dat";
TRY
{
    CFile f( pFileName, CFile::modeCreate | CFile::modeWrite );
}
CATCH( CFileException, e )
{
    #ifdef _DEBUG
        afxDump << "File could not be opened " << e->m_cause << "\n";
    #endif
}
END_CATCH
```

CFile::Close

virtual void Close( )
throw(CFileException);

Remarks

Closes the file associated with this object and makes the file unavailable for reading or writing. If you have not closed the file before destroying the object, the destructor closes it for you. If you used **new** to allocate the **CFile** object on the heap, then you must delete it after closing the file. **Close** sets **m_hFile** to **CFile::hFileNull**.

See Also

CFile::Open

CFile::Duplicate

virtual CFile* Duplicate( ) const
throw(CFileException);

Remarks

Constructs a duplicate **CFile** object for a given file. This is equivalent to the C run-time function **_dup**.

CFile::Flush

virtual void Flush( )
throw(CFileException);

Remarks Forces any data remaining in the file buffer to be written to the file. The use of **Flush** does not guarantee flushing of **CArchive** buffers. If you are using an archive, call **CArchive::Flush** first.

CFile::GetLength

virtual DWORD GetLength( ) const
throw(CFileException);

Remarks Obtains the current logical length of the file in bytes, not the amount physically allocated.

Return Value The length of the file.

See Also **CFile::SetLength**

CFile::GetPosition

virtual DWORD GetPosition( ) const
throw(CFileException);

Remarks Obtains the current value of the file pointer, which can be used in subsequent calls to **Seek**.

Return Value The file pointer as a 32-bit doubleword.

Example

```
extern CFile cfile;
DWORD dwPosition = cfile.GetPosition();
```

CFile::GetStatus

BOOL GetStatus(CFileStatus& *rStatus* **) const;**

static BOOL PASCAL GetStatus(const char* *pszFileName*, **CFileStatus&** *rStatus* **);**

rStatus A reference to a user-supplied **CFileStatus** structure that will receive the status information. The **CFileStatus** structure has the following fields with the meanings as given:

- **CTime m_ctime** The date and time the file was created
- **CTime m_mtime** The date and time the file was last modified
- **CTime m_atime** The date and time the file was last accessed for reading
- **LONG m_size** The logical size of the file in bytes, as reported by the MS-DOS command DIR
- **BYTE m_attribute** The MS-DOS attribute byte of the file
- **char m_szFullName[_MAX_PATH]** The absolute filename in the Windows character set. When running under MS-DOS only, **m_szFullName** is an OEM character string. (**_MAX_PATH** is defined in STDLIB.H.)

pszFileName A string in the Windows character set that is the path to the desired file. When running under MS-DOS only, *pszFileName* is an OEM character string. The path may be relative or absolute, but may not contain a network name.

Remarks

The virtual version of **GetStatus** retrieves the status of the open file associated with this **CFile** object. It does not insert a value into the **m_szFullName** structure member.

The static version gets the status of the named file and copies the filename to **m_szFullName**. This function obtains the file status from the directory entry without actually opening the file. It is useful for testing the existence and access rights of a file.

The **m_attribute** is the MS-DOS file attribute. The Microsoft Foundation classes provide an **enum** type attribute so that you can specify attributes symbolically:

```
enum Attribute {
   normal =    0x00,
   readOnly =  0x01,
   hidden =    0x02,
   system =    0x04,
   volume =    0x08,
   directory = 0x10,
   archive =   0x20
   };
```

Note This function is not available for the **CMemFile**-derived class.

See Also **CFile::UnlockRange**

Example

```
extern DWORD dwPos;
extern DWORD dwCount;
extern CFile cfile;
cfile.LockRange( dwPos, dwCount );
```

CFile::Open

virtual BOOL Open(const char* *pszFileName*, **UINT** *nOpenFlags*, **CFileException*** *pError* = **NULL);**

pszFileName A string that is the path to the desired file. The path may be relative or absolute but may not contain a network name.

nOpenFlags A **UINT** that defines the file's sharing and access mode. It specifies the action to take when opening the file. You can combine options by using the bitwise-OR (|) operator. One access permission and one share option are required; the **modeCreate** and **modeNoInherit** modes are optional. See the **CFile** constructor for a list of mode options.

pError A pointer to an existing file-exception object that indicates the completion status of the open operation.

Remarks **Open** is designed for use with the default **CFile** constructor. The two functions form a "safe" method for opening a file where a failure is a normal, expected condition. The constructor is guaranteed to succeed, and **Open** returns a pointer to an exception object, bypassing the **THROW/TRY/CATCH** mechanism.

Return Value **TRUE** if the open was successful; otherwise **FALSE**. The *pError* parameter is meaningful only if **FALSE** is returned.

See Also **CFile::CFile**, **CFile::Close**

Return Value

TRUE if no error, in which case *rStatus* is valid; otherwise **FALSE**. **FALSE** indicates that the file does not exist.

See Also

CFile::SetStatus, CTime

Example

```
CFileStatus status;
extern CFile cfile;
if( cfile.GetStatus( status ) )    // virtual member function
   {
       #ifdef _DEBUG
        afxDump << "File size = " << status.m_size << "\n";
       #endif
   }
char* pFileName = "test.dat";
if( CFile::GetStatus( pFileName, status ) )   // static function
   {
       #ifdef _DEBUG
        afxDump << "Full file name = " << status.m_szFullName << "\n";
       #endif
   }
```

CFile::LockRange

virtual void LockRange(DWORD *dwPos*, **DWORD** *dwCount* **)**
throw(CFileException);

dwPos The byte offset of the start of the byte range to lock.

dwCount The number of bytes in the range to lock.

Remarks

Locks a range of bytes in an open file, throwing an exception if the file is already locked. Locking bytes in a file prevents access to those bytes by other processes. You can lock more than one region of a file, but no overlapping regions are allowed.

When you unlock the region, using the **UnlockRange** member function, the byte range must correspond exactly to the region that was previously locked. The **LockRange** function does not merge adjacent regions; if two locked regions are adjacent, you must unlock each region separately.

Under MS-DOS, you must enable file sharing by running SHARE.EXE before running an application using this member function.

Example

```
CFile f;
CFileException e;
char* pFileName = "test.dat";
if( !f.Open( pFileName, CFile::modeCreate | CFile::modeWrite, &e ) )
    {
        #ifdef _DEBUG
            afxDump << "File could not be opened " << e.m_cause << "\n";
        #endif
    }
```

CFile::Read

virtual UINT Read(void FAR* *lpBuf*, **UINT** *nCount* **)**
throw(CFileException);

lpBuf Pointer to the user-supplied buffer that is to receive the data read from the file.

nCount The maximum number of bytes to be read from the file. For text-mode files, carriage return–linefeed pairs are counted as single characters.

Remarks

Reads data into a buffer from the file associated with the **CFile** object.

Return Value

The number of bytes transferred to the buffer. Note that for all **CFile** classes, the return value may be less than *nCount* if the end of file was reached.

See Also

CFile::Write

Example

```
extern CFile cfile;
char pbuf[100];
UINT nBytesRead = cfile.Read( pbuf, 100 );
```

CFile::Remove

static void PASCAL Remove(const char* *pszFileName* **)**
throw(CFileException);

pszFileName A string that is the path to the desired file. The path may be relative or absolute but may not contain a network name.

Remarks

This static function deletes the file specified by the path. It will not remove a directory. The **Remove** member function throws an exception if the connected file is open or if the file cannot be removed. This is equivalent to the MS-DOS DEL command.

Example

```
char* pFileName = "test.dat";
TRY
{
    CFile::Remove( pFileName );
}
CATCH( CFileException, e )
{
    #ifdef _DEBUG
        afxDump << "File " << pFileName << " cannot be removed\n";
    #endif
}
END_CATCH
```

CFile::Rename

static void PASCAL Rename(const char* *pszOldName*, **const char*** *pszNewName* **)**
throw(CFileException);

pszOldName The old path.

pszNewName The new path.

Remarks

This static function renames the specified file. Directories cannot be renamed. This is equivalent to the MS-DOS REN command.

Example

```
extern char* pOldName;
extern char* pNewName;
TRY
{
    CFile::Rename( pOldName, pNewName );
}
CATCH( CFileException, e )
{
    #ifdef _DEBUG
        afxDump << "File " << pOldName << " not found, cause = "
            << e->m_cause << "\n";
    #endif
}
END_CATCH
```

CFile::Seek

virtual LONG Seek(LONG *lOff*, **UINT** *nFrom*)
throw(CFileException);

lOff Number of bytes to move the pointer.

nFrom Pointer movement mode. Must be one of the following values, with the meaning as given:

- **CFile::begin** Move the file pointer *lOff* bytes forward from the beginning of the file.
- **CFile::current** Move the file pointer *lOff* bytes from the current position in the file.
- **CFile::end** Move the file pointer backward *lOff* bytes from the end of the file.

Remarks

Repositions the pointer in a previously opened file. The **Seek** function permits random access to a file's contents by moving the pointer a specified amount, absolutely or relatively. No data is actually read during the seek. When a file is opened, the file pointer is positioned at offset 0, the beginning of the file.

Return Value

If the requested position is legal, **Seek** returns the new byte offset from the beginning of the file. Otherwise, the return value is undefined and a **CFileException** object is thrown.

Example

```
extern CFile cfile;
LONG lOffset = 1000, lActual;
lActual = cfile.Seek( lOffset, CFile::begin );
```

CFile::SeekToBegin

void SeekToBegin( )
throw(CFileException);

Remarks

Sets the value of the file pointer to the beginning of the file. `SeekToBegin()` is equivalent to `Seek( 0L, CFile::begin )`.

Example

```
extern CFile cfile;
cfile.SeekToBegin();
```

CFile::SeekToEnd

DWORD SeekToEnd( )
throw(CFileException);

Remarks Sets the value of the file pointer to the logical end of the file. `SeekToEnd( )` is equivalent to `CFile::Seek( 0L, CFile::end )`.

Return Value The length of the file in bytes.

See Also **CFile::GetLength, CFile::Seek, CFile::SeekToBegin**

Example

```
extern CFile cfile;
DWORD dwActual = cfile.SeekToEnd();
```

CFile::SetLength

virtual void SetLength(const DWORD *dwNewLen* **)**
throw(CFileException);

dwNewLen Desired length of the file in bytes. This value may be larger or smaller than the current length of the file. The file will be extended or truncated as appropriate.

Remarks Changes the length of the file.

Note With **CMemFile**, this function could throw a **CMemoryException** object.

Example

```
extern CFile cfile;
DWORD dwNewLength = 10000;
cfile.SetLength( dwNewLength );
```

CFile::SetStatus

static void SetStatus(const char* *pszFileName***, const CFileStatus&** *status* **)**
throw(CFileException);

pszFileName A string that is the path to the desired file. The path may be relative or absolute but may not contain a network name.

status The buffer containing the new status information. Call the **GetStatus** member function to prefill the **CFileStatus** structure with current values, then make changes as required. If a value is 0, then the corresponding status item is not updated. See the **GetStatus** member function for a description of the **CFileStatus** structure.

Remarks

Sets the status of the file associated with this file location. Under MS-DOS, all times in the **CFileStatus** structure, as described in the **GetStatus** member function, contain the same value. To set the time, modify the **m_mtime** field of *status*. The **SetStatus** function will throw an exception under MS-DOS if the file's read-only attribute is set.

See Also

CFile::GetStatus

Example

```
char* pFileName = "test.dat";
extern BYTE newAttribute;
CFileStatus status;
CFile::GetStatus( pFileName, status );
status.m_attribute = newAttribute;
CFile::SetStatus( pFileName, status );
```

CFile::UnlockRange

virtual void UnlockRange(DWORD *dwPos*, **DWORD** *dwCount* **)**
throw(CFileException);

dwPos The byte offset of the start of the byte range to unlock.

dwCount The number of bytes in the range to unlock.

Remarks

Unlocks a range of bytes in an open file. See the description of the **LockRange** member function for details.

Under MS-DOS, you must load SHARE.EXE; otherwise, the function throws a **CFileException** object.

Note This function is not available for the **CMemFile**-derived class.

See Also

CFile::LockRange

Example

```
extern DWORD dwPos;
extern DWORD dwCount;
extern CFile cfile;
cfile.UnlockRange( dwPos, dwCount );
```

CFile::Write

virtual void Write(const void FAR* *lpBuf*, **UINT** *nCount* **)**
throw(CFileException);

lpBuf A pointer to the user-supplied buffer that contains the data to be written to the file.

nCount The number of bytes to be transferred from the buffer. For text-mode files, carriage return–linefeed pairs are counted as single characters.

Remarks

Writes data from a buffer to the file associated with the **CFile** object. **Write** throws an exception in response to several conditions including the disk-full condition.

See Also

CFile::Read, **CStdioFile::WriteString**

Example

```
extern CFile cfile;
char pbuf[100];
cfile.Write( pbuf, 100 );
```

Data Members

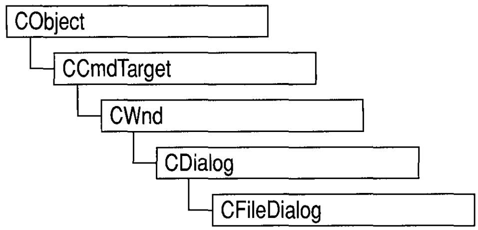
CFile::m_hFile

Remarks

Contains the operating-system file handle for an open file. **m_hFile** is a public variable of type **UINT**. It contains **CFile::m_hFileNull** (an operating-system-independent empty file indicator) if the handle has not been assigned.

Use of **m_hFile** is not recommended because the member's meaning depends on the derived class. **m_hFile** is made a public member to conveniently support nonpolymorphic use of the class.

class CFileDialog : public CDialog



The **CFileDialog** class encapsulates the Windows common file dialog box. Common file dialog boxes provide an easy way to implement File Open and File Save As dialog boxes (as well as other file-selection dialog boxes) in a manner consistent with Windows standards.

You can use **CFileDialog** "as is" with the constructor provided, or you can derive your own dialog class from **CFileDialog** and write a constructor to suit your needs. In either case, these dialog boxes will behave like standard Microsoft Foundation class dialog boxes because they are derived from the **CDialog** class.

To use a **CFileDialog** object, first create the object using the **CFileDialog** constructor. Once the dialog has been constructed, you can set or modify any values in the **m_ofn** structure to initialize the values or states of the dialog box's controls. The **m_ofn** structure is of type **OPENFILENAME**. For more information on this structure, see the *Windows Software Development Kit* (SDK) documentation.

After initializing the dialog box's controls, call the **DoModal** member function to display the dialog box and allow the user to enter the path and file. **DoModal** returns whether the user selected the OK (**IDOK**) or the Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CFileDialog**'s public member functions to retrieve the information input by the user.

CFileDialog includes several protected members that enable you to do custom handling of share violations, filename validation, and list-box change notification. These protected members are callback functions that most applications do not need to use, since default handling is done automatically. Message-map entries for these functions are not necessary because they are standard virtual functions.

You can use the Windows **CommDlgExtendedError** function to determine if an error occurred during initialization of the dialog box and to learn more about the error.

The destruction of **CFileDialog** objects is handled automatically. It is not necessary to call **CDialog::EndDialog**.

To allow the user to select multiple files, set the **OFN.ALLOW_MULTISELECT** flag before calling **DoModal**. You need to supply your own filename buffer to accommodate the returned list of multiple file names. Do this by replacing

m_ofn.lpstrFile with a pointer to a buffer you have allocated, after constructing the **CFileDialog**, but before calling **DoModal**.

CFileDialog relies on the COMMDLG.DLL file that ships with Windows version 3.1. For details about redistributing COMMDLG.DLL to Windows version 3.0 users, see the *Getting Started* manual in the Windows version 3.1 SDK.

If you derive a new class from **CFileDialog**, you can use a message map to handle any messages. To extend the default message handling, derive a class from **CWnd**, add a message map to the new class, and provide member functions for the new messages. You do not need to provide a hook function to customize the dialog box.

To customize the dialog box, derive a class from **CFileDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

#include <afxdlgs.h>

Data Members—Public Members

m_ofn	The Windows **OPENFILENAME** structure. Provides access to basic file dialog box parameters.

Construction/Destruction—Public Members

CFileDialog	Constructs a **CFileDialog** object.

Overridables—Public Members

DoModal	Displays the dialog box and allows the user to make a selection.
GetPathName	Returns the full path of the selected file.
GetFileName	Returns the filename of the selected file.
GetFileExt	Returns the file extension of the selected file.
GetFileTitle	Returns the title of the selected file.
GetReadOnlyPref	Returns the read-only status of the selected file.

Operations — Protected Members

OnShareViolation	Called when a share violation occurs.
OnFileNameOK	Called to validate the filename entered in the dialog box.
OnLBSelChangedNotify	Called when the list box selection changes.

Member Functions

CFileDialog::CFileDialog

CFileDialog(BOOL *bOpenFileDialog*, **LPCSTR** *lpszDefExt* = **NULL, LPCSTR** *lpszFileName* = **NULL, DWORD** *dwFlags* = **OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, LPCSTR** *lpszFilter* = **NULL, CWnd*** *pParentWnd* = **NULL);**

bOpenFileDialog Set to **TRUE** to construct a File Open dialog box or **FALSE** to construct a File Save As dialog box.

lpszDefExt The default filename extension. If the user does not include an extension in the Filename edit box, the extension specified by *lpszDefExt* is automatically appended to the filename. If this parameter is **NULL**, no file extension is appended.

lpszFileName The initial filename that appears in the filename edit box. If **NULL**, no filename initially appears.

dwFlags A combination of one or more flags that allow you to customize the dialog box. For a description of these flags, see the **OPENFILENAME** structure description in the Windows SDK documentation. If you modify the **m_ofn.Flags** structure member, use a bitwise-OR operator in your changes to keep the default behavior intact.

lpszFilter A series of string pairs that specify filters you can apply to the file. If you specify file filters, only selected files will appear in the Files list box. See the "Remarks" section below for more information on how to work with file filters.

pParentWnd A pointer to the file dialog-box object's parent or owner window.

Remarks

Call this function to construct a standard Windows file dialog box object. Either a File Open or File Save As dialog box is constructed, depending on the value of *bOpenFileDialog*.

The *lpszFilter* parameter is used to determine the type of filename a file must have to be displayed in the file list box. The first string in the string pair describes the filter; the second string indicates the file extension to use. Multiple extensions may be specified using ';' as the delimiter. The string ends with two '|' characters, followed by a **NULL** character. You can also use a **CString** object for this parameter.

For example, Microsoft Excel permits users to open files with extensions .XLC (chart) or .XLS (worksheet), among others. The filter for Excel could be written as:

```
static char BASED_CODE szFilter[] = "Chart Files (*.xlc) | *.xlc |
Worksheet Files (*.xls) | *.xls | Data Files (*.xlc;*.xls) | *.xlc;
*.xls | All Files (*.*) | *.* ||"
```

See Also **CFileDialog::DoModal**, **::GetOpenFileName**, **::GetSaveFileName**

CFileDialog::DoModal

virtual int DoModal( );

Remarks Call this function to display the Windows common file dialog box and allow the user to browse files and directories and enter a filename.

If you want to initialize the various file dialog-box options by setting members of the **m_ofn** structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

When the user clicks the dialog box's OK or Cancel buttons, or selects the Close option from the dialog box's control menu, control is returned to your application. You can then call other member functions to retrieve the settings or information the user inputs into the dialog box.

DoModal is a virtual function derived from class **CModalDialog**.

Return Value **IDOK** or **IDCANCEL** if the function is successful; otherwise 0. **IDOK** and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

If **IDCANCEL** is returned, you can call the Windows **CommDlgExtendedError** function to determine if an error occurred.

See Also **CDialog::DoModal**, **CFileDialog::CFileDialog**

CFileDialog::GetFileExt

CString GetFileExt( ) const;

Remarks

Call this function to retrieve the extension of the filename entered into the dialog box. For example, if the name of the file entered is DATA.TXT, **GetFileExt** returns "TXT".

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, then this member function only applies to the first name.

Return Value

The extension of the filename.

See Also

CFileDialog::GetPathName, **CFileDialog::GetFileName**, **CFileDialog::GetFileTitle**

CFileDialog::GetFileName

CString GetFileName( ) const;

Remarks

Call this function to retrieve the name of the file entered in the dialog box. The name of the file includes only its prefix, without the path or the extension. For example, **GetFileName** will return "TEXT" for the file C:\FILES\TEXT.DAT.

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, then this member function only applies to the first name.

Return Value

The name of the file.

See Also

CFileDialog::GetPathName, **CFileDialog::GetFileExt**, **CFileDialog::GetFileTitle**

CFileDialog::GetFileTitle

CString GetFileTitle( ) const;

Remarks

Call this function to retrieve the title of the filename entered in the dialog box. The title of the filename includes both the name and the extension. For example, **GetFileTitle** will return "TEXT.DAT" for the file C:\FILES\TEXT.DAT.

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, then this member function only applies to the first name.

Return Value The title of the file.

See Also **CFileDialog::GetPathName**, **CFileDialog::GetFileName**, **CFileDialog::GetFileExt**, **::GetFileTitle**

CFileDialog::GetPathName

CString GetPathName( ) const;

Remarks Call this function to retrieve the full path of the file entered in the dialog box. The path of the filename includes the file's title plus the entire directory path. For example, **GetPathName** will return "C:\FILES\TEXT.DAT" for the file C:\FILES\TEXT.DAT.

If **m_ofn.Flags** has the **OFN_ALLOWMULTISELECT** flag set, then this member function only applies to the first name.

Return Value The full path of the file.

See Also **CFileDialog::GetFileName**, **CFileDialog::GetFileExt**, **CFileDialog::GetFileTitle**

CFileDialog::GetReadOnlyPref

BOOL GetReadOnlyPref( ) const;

Remarks Call this function to determine whether the Read Only check box has been selected in the Windows standard File Open and File Save As dialog boxes. The Read Only check box can be hidden by setting the **OFN_HIDEREADONLY** style in the **CFileDialog** constructor.

Return Value Non-zero if the Read Only check box in the dialog box is selected; otherwise 0.

See Also **CFileDialog::CFileDialog**, **CFileDialog::GetPathName**, **CFileDialog::GetFileExt**

CFileDialog::OnFileNameOK

Protected

virtual BOOL OnFileNameOK( ); ♦

Remarks

Override this function only if you want to provide custom validation of filenames that are entered into a common file dialog box. This function allows you to reject a filename for any application-specific reason. Normally, you do not need to use this function because the framework provides default validation of filenames and displays a message box if an invalid filename is entered.

If a nonzero value is returned, the dialog box will remain displayed for the user to enter another filename.

Return Value

Nonzero if the filename is a valid MS-DOS filename; otherwise 0.

See Also

OPENFILENAME

CFileDialog::OnLBSelChangedNotify

Protected

virtual void OnLBSelChangedNotify(UINT *nIDBox*, **UINT** *iCurSel*, **UINT** *nCode*); ♦

nIDBox The ID of the list box or combo box in which the selection occurred.

iCurSel The index of the current selection.

nCode The control notification code.

This parameter must have one of the following values, with the meaning as given:

- **CD_LBSELCHANGE** Specifies *iCurSel* is the selected item in a single-selection list box.
- **CD_LBSELSUB** Specifies that *iCurSel* is no longer selected in a multiselection list box.
- **CD_LBSELADD** Specifies that *iCurSel* was selected in a multiselection list box.
- **CD_LBSELNOITEMS** Specifies that no selection exists in a multiselection list box.

For more information, see "Filename Dialog Boxes" in the Windows SDK Help.

Remarks

This function is called whenever the current selection in a list box is about to change. Override this function to provide custom handling of selection changes in the list box. For example, you can use this function to display the access rights or date-last-modified of each file the user selects.

CFileDialog::OnShareViolation

Protected

virtual UINT OnShareViolation(LPCSTR *lpszPathName* **);** ♦

lpszPathName The path of the file on which the share violation occurred.

Remarks

Override this function to provide custom handling of share violations. Normally, you do not need to use this function because the framework provides default checking of share violations and displays a message box if a share violation occurs.

If you want to disable share violation checking, use the bitwise-OR operator to combine the flag **OFN_SHAREAWARE** with **m_ofn.Flags**.

Return Value

One of the following values, with the meaning as given:

- **OFN_SHAREFALLTHROUGH** The filename is returned from the dialog box.
- **OFN_SHARENOWARN** No further action needs to be taken.
- **OFN_SHAREWARN** The user receives the standard warning message for this error.

See Also

CFileDialog::OnFileNameOK

Data Members

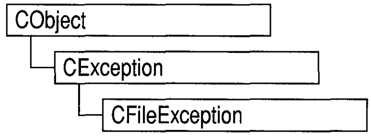
CFileDialog::m_ofn

Remarks

m_ofn is a structure of type **OPENFILENAME**. Use this structure to initialize the appearance of a File Open or File Save As dialog box after it is constructed but before it is displayed with the **DoModal** member function. For example, you can set the **lpszTitle** member of **m_ofn** to the caption you want the dialog box to have.

For more information on this structure, including a listing of its members, see **OPENFILENAME** in the Windows SDK documentation.

class CFileException : public CException



A **CFileException** object represents a file-related exception condition. The **CFileException** class includes public data members that hold the portable cause code and the operating-system-specific error number. The class also provides static member functions for throwing file exceptions and for returning cause codes for both operating-system errors and C run-time errors. **CFileException** objects are constructed and thrown in **CFile** member functions and in member functions of derived classes. You can access these objects within the scope of a **CATCH** expression. For portability, use only the cause code to get the reason for an exception. For more information about exceptions, see Chapter 16, “Exceptions,” in the *Class Library User’s Guide*.

#include <afx.h>

See Also **CFile**

Data Members—Public Members

m_cause	Contains portable code corresponding to the exception cause.
m_lOsError	Contains the related operating-system error number.

Construction/Destruction—Public Members

CFileException	Constructs a **CFileException** object.

Code Conversion—Public Members

OsErrorToException	Returns a cause code corresponding to an MS-DOS error code.
ErrnoToException	Returns cause code corresponding to a run-time error number.

Helper Functions—Public Members

ThrowOsError	Throws a file exception based on an operating-system error number.
ThrowErrno	Throws a file exception based on a run-time error number.

Member Functions

CFileException::CFileException

CFileException(int *cause* **= CFileException::none, LONG** *lOsError* **= –1);**

cause An enumerated type variable that indicates the reason for the exception. See **CFileException::m_cause** for a list of the possible values.

lOsError An operating-system-specific reason for the exception, if available. The *lOsError* parameter provides more information than *cause* does.

Remarks

Constructs a **CFileException** object that stores the cause code and the operating-system code in the object. Do not use this constructor directly, but rather call the global function **AfxThrowFileException**.

Note The variable *lOsError* applies only to **CFile** and **CStdioFile** objects. The **CMemFile** class does not handle this error code. More information specifically about the operating system is available through the run-time function **_dosexterr** (MS-DOS only).

See Also

AfxThrowFileException

CFileException::ErrnoToException

static int PASCAL ErrnoToException(int *nErrno* **);**

nErrno An integer error code as defined in the run-time include file ERRNO.H.

Remarks

Converts a given run-time library error value to a **CFileException** enumerated error value. See **CFileException::m_cause** for a list of the possible enumerated values.

Return Value

Enumerated value that corresponds to a given run-time library error value.

See Also

CFileException::OsErrorToException

Example

```
#include <errno.h>
ASSERT( CFileException::ErrnoToException( EACCES ) ==
                    CFileException::accessDenied );
```

CFileException::OsErrorToException

static int PASCAL OsErrorToException(LONG *lOsError* **);**

lOsError An operating-system-specific error code.

Remarks

Returns an enumerator that corresponds to a given *lOsError* value. If the error code is unknown, then the function returns **CFileException::generic**.

Return Value

Enumerated value that corresponds to a given operating-system error value.

See Also

CFileException::ErrnoToException

Example

```
ASSERT( CFileException::OsErrorToException( 5 ) ==
                    CFileException::accessDenied );
```

CFileException::ThrowErrno

static void PASCAL ThrowErrno(int *nErrno* **);**

nErrno An integer error code as defined in the run-time include file ERRNO.H.

Remarks

Constructs a **CFileException** object corresponding to a given *nErrno* value, then throws the exception.

See Also

CFileException::ThrowOsError

Example

```
#include <errno.h>
CFileException::ThrowErrno( EACCES );  // "access denied"
```

CFileException::ThrowOsError

static void PASCAL ThrowOsError(LONG *lOsError* **);**

lOsError An operating-system-specific error code.

Remarks

Throws a **CFileException** corresponding to a given *lOsError* value. If the error code is unknown, then the function throws an exception coded as **CFileException::generic**.

See Also **CFileException::ThrowErrno**

Example

```
FileException::ThrowOsError( 5 );  // "access denied"
```

Data Members

CFileException::m_cause

Remarks

Contains values defined by a **CFileException** enumerated type. This data member is a public variable of type **int**. The enumerators and their meanings are as follows:

- **CFileException::none** No error occurred.
- **CFileException::generic** An unspecified error occurred.
- **CFileException::fileNotFound** The file could not be located.
- **CFileException::badPath** All or part of the path is invalid.
- **CFileException::tooManyOpenFiles** The permitted number of open files was exceeded.
- **CFileException::accessDenied** The file could not be accessed.
- **CFileException::invalidFile** There was an attempt to use an invalid file handle.
- **CFileException::removeCurrentDir** The current working directory cannot be removed.
- **CFileException::directoryFull** There are no more directory entries.
- **CFileException::badSeek** There was an error trying to set the file pointer.
- **CFileException::hardIO** There was a hardware error.
- **CFileException::sharingViolation** SHARE.EXE was not loaded, or a shared region was locked.
- **CFileException::lockViolation** There was an attempt to lock a region that was already locked.
- **CFileException::diskFull** The disk is full.
- **CFileException::endOfFile** The end of file was reached.

Note These **CFileException** cause enumerators are distinct from the **CArchiveException** cause enumerators.

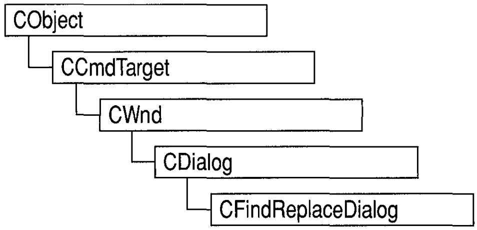
Example

```
extern char* pFileName;
TRY
{
   CFile f( pFileName, CFile::modeCreate | CFile::modeWrite );
}
CATCH( CFileException, e)
{
    if( e->m_cause == CFileException::fileNotFound )
        printf( "ERROR: File not found\n");
}
```

CFileException::m_lOsError

Remarks Contains the operating-system error code for this exception. See your operating-system technical manual for a listing of error codes. This data member is a public variable of type **LONG**.

class CFindReplaceDialog : public CDialog



The **CFindReplaceDialog** class allows you to implement standard string Find/Replace dialog boxes in your application. Unlike the other Windows common dialog boxes, **CFindReplaceDialog** objects are modeless, allowing users to interact with other windows while they are on screen. There are two kinds of **CFindReplaceDialog** objects: Find dialog boxes and Find/Replace dialog boxes. Although the dialog boxes allow the user to input search and search/replace strings, they do not perform any of the searching or replacing functions. You must add these to the application.

To construct a **CFindReplaceDialog** object, use the provided constructor (which has no arguments). Since this is a modeless dialog box, allocate the object on the heap using the **new** operator, rather than on the stack.

Once a **CFindReplaceDialog** object has been constructed, you must call the **Create** member function to create and display the dialog box.

Use the **m_fr** structure to initialize the dialog box before calling **Create**. The **m_fr** structure is of type **FINDREPLACE**. For more information on this structure, see the *Windows Software Development Kit* (SDK) documentation.

In order for the parent window to be notified of find/replace requests, you must use the Windows **RegisterMessage** function and use the **ON_REGISTERED_MESSAGE** message-map macro in your frame window that handles this registered message. You can call any of the member functions listed in the following "Operations–Public Members" section from the frame window's callback function.

You can determine if the user has decided to terminate the dialog box with the **IsTerminating** member function.

CFindReplaceDialog relies on the COMMDLG.DLL file that ships with Windows version 3.1. For details about redistributing COMMDLG.DLL to Windows version 3.0 users, see the *Getting Started* manual in the Windows version 3.1 SDK.

To customize the dialog box, derive a class from **CFindReplaceDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

#include <afxdlgs.h>

Data Members—Public Members

m_fr	A structure used to customize a **CFindReplaceDialog** object.

Construction/Destruction—Public Members

CFindReplaceDialog	Call this function to construct a **CFindReplaceDialog** object.
Create	Creates and displays a **CFindReplaceDialog** dialog box.

Operations—Public Members

FindNext	Call this function to determine whether the user wants to find the next occurrence of the find string.
GetNotifier	Call this function to retrieve the **FINDREPLACE** structure in your registered message handler.
GetFindString	Call this function to retrieve the current find string.
GetReplaceString	Call this function to retrieve the current replace string.
IsTerminating	Call this function to determine whether the dialog box is terminating.
MatchCase	Call this function to determine if the user wants to match the case of the find string exactly.
MatchWholeWord	Call this function to determine whether the user wants to match entire words only.
ReplaceAll	Call this function to determine whether the user wants all occurrences of the string to be replaced.
ReplaceCurrent	Call this function to determine whether the user wants the current word to be replaced.
SearchDown	Call this function to determine whether the user wants the search to proceed in a downward direction.

Member Functions

CFindReplaceDialog::CFindReplaceDialog

CFindReplaceDialog( );

Remarks Constructs a **CFindReplaceDialog** object. **CFindReplaceDialog** objects are constructed on the heap with the **new** operator. See the class description above for more information on the construction of **CFindReplaceDialog** objects. Use the **Create** member function to display the dialog box.

See Also **CFindReplaceDialog::Create**

CFindReplaceDialog::Create

BOOL Create(BOOL *bFindDialogOnly*, **LPCSTR** *lpszFindWhat*, **LPCSTR** *lpszReplaceWith* = **NULL, DWORD** *dwFlags* = **FR_DOWN, CWnd*** *pParentWnd* = **NULL);**

bFindDialogOnly Set this parameter to **TRUE** to display the standard Windows Find dialog box. Set it to **FALSE** to display the Windows Find/Replace dialog box.

lpszFindWhat Specifies the string to search for.

lpszReplaceWith Specifies the default string to replace found strings with.

dwFlags One or more flags you can use to customize the settings of the dialog box, combined using the bitwise-OR operator. The default value is **FR_DOWN**, which specifies that the search is to proceed in a downward direction. See the **FINDREPLACE** structure in the Windows SDK for more information on these flags.

pParentWnd A pointer to the dialog box's parent or owner window. This is the window that will receive the special message indicating that a find/replace action is requested. If **NULL**, the application's main window is used.

Remarks

Creates and displays either a Find or Find/Replace dialog box object, depending on the value of *bFindDialogOnly*.

In order for the parent window to be notified of find/replace requests, you must use the Windows **RegisterMessage** function whose return value is a message number unique to the application's instance. Your frame window should have a message map entry that declares the callback function (**OnFindReplace** in the example that follows) that handles this registered message. The following code fragment is an example of how to do this for a frame window class named `CMyFrameWnd`:

```
class CMyFrameWnd : public CFrameWnd
{
protected:
    afx_msg LONG LRESULT OnFindReplace(WPARAM wParam, LPARAM
lParam);

    DECLARE_MESSAGE_MAP()
};
static UINT NEAR WM_FINREPLACE = ::RegisterMessage(FINDMSGSTRING);

BEGIN_MESSAGE_MAP( CMyFrameWnd, CFrameWnd )
   //Normal message map entries here.
   ON_REGISTERED_MESSAGE( WM_FINDREPLACE, OnFindReplace )
END_MESSAGE_MAP
```

Within your **OnFindReplace** function, you interpret the intentions of the user and create the code for the find/replace operations.

See Also

CFindReplaceDialog::CFindReplaceDialog

CFindReplaceDialog::FindNext

BOOL FindNext( ) const;

Remarks

Call this function from your callback function to determine whether the user wants to find the next occurrence of the search string.

Return Value

Nonzero if the user wants to find the next occurrence of the search string; otherwise 0.

See Also

CFindReplaceDialog::GetFindString, **CFindReplaceDialog::SearchDown**

CFindReplaceDialog::GetFindString

CString GetFindString( ) const;

Remarks Call this function from your callback function to retrieve the default string to find.

Return Value The default string to find.

See Also **CFindReplaceDialog::FindNext**, **CFindReplaceDialog::GetReplaceString**

CFindReplaceDialog::GetNotifier

static CFindReplaceDialog* PASCAL GetNotifier(LPARAM *lParam* **);**

lParam The **lparam** value passed to the frame window's **OnFindReplace** member function.

Remarks Call this function to retrieve a pointer to the current Find Replace dialog box. It should be used within your callback function to access the current dialog box, call its member functions, and access the **m_fr** structure.

Return Value A pointer to the current dialog box.

CFindReplaceDialog::GetReplaceString

CString GetReplaceString( ) const;

Return Value The default string to replace found strings with.

See Also **CFindReplaceDialog::GetFindString**

CFindReplaceDialog::IsTerminating

BOOL IsTerminating( ) const;

Remarks Call this function within your callback function to determine whether the user has decided to terminate the dialog box. If this function returns nonzero, you should call the **DestroyWindow** member function of the current dialog box and set any dialog box pointer variable to **NULL**. Optionally, you can also store the find/replace text last entered and use it to initialize the next find/replace dialog box.

Return Value Nonzero if the user has decided to terminate the dialog box; otherwise 0.

CFindReplaceDialog::MatchCase

BOOL MatchCase( ) const;

Return Value Nonzero if the user wants to find occurrences of the search string that exactly match the case of the search string; otherwise 0.

See Also **CFindReplaceDialog::MatchWholeWord**

CFindReplaceDialog::MatchWholeWord

BOOL MatchWholeWord( ) const;

Return Value Nonzero if the user wants to match only the entire words of the search string; otherwise 0.

See Also **CFindReplaceDialog::MatchCase**

CFindReplaceDialog::ReplaceAll

BOOL ReplaceAll( ) const;

Return Value Nonzero if the user has requested that all strings matching the replace string be replaced; otherwise 0.

See Also **CFindReplaceDialog::ReplaceCurrent**

CFindReplaceDialog::ReplaceCurrent

BOOL ReplaceCurrent( ) const;

Return Value Nonzero if the user has requested that the currently selected string be replaced with the replace string; otherwise 0.

See Also **CFindReplaceDialog::ReplaceAll**

CFindReplaceDialog::SearchDown

BOOL SearchDown( ) const;

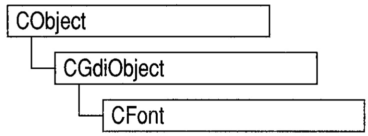
Return Value Nonzero if the user wants the search to proceed in a downward direction; 0 if the user wants the search to proceed in an upward direction.

Data Members

CFindReplaceDialog::m_fr

Remarks **m_fr** is a structure of type **FINDREPLACE**. Its members store the characteristics of the dialog-box object. After constructing a **CFindReplaceDialog** object, you can use **m_fr** to initialize various values in the dialog box. You must initialize the dialog box's values before calling the **Create** member function. For more information on this structure, see the **FINDREPLACE** structure in the Windows SDK documentation.

class CFont : public CGdiObject



The **CFont** class encapsulates a Windows graphics device interface (GDI) font and provides member functions for manipulating the font. To use a **CFont** object, construct a **CFont** object and attach a Windows font to it with **CreateFont** or **CreateFontIndirect**, and then use the object's member functions to manipulate the font.

#include <afxwin.h>

Construction/Destruction—Public Members

CFont	Constructs a **CFont** object.

Initialization—Public Members

CreateFontIndirect	Initializes a **CFont** object with the characteristics given in a **LOGFONT** structure.
CreateFont	Initializes a **CFont** with the specified characteristics.

Operations—Public Members

FromHandle	Returns a pointer to a **CFont** object when given a Windows **HFONT**.

Member Functions

CFont::CFont

CFont( );

Remarks Constructs a **CFont** object. The resulting object must be initialized with **CreateFont** or **CreateFontIndirect** before it can be used.

See Also **CFont::CreateFontIndirect**, **CFont::CreateFont**, **::EnumFonts**

CFont::CreateFont

BOOL CreateFont(int *nHeight*, **int** *nWidth*, **int** *nEscapement*, **int** *nOrientation*, **int** *nWeight*, **BYTE** *bItalic*, **BYTE** *bUnderline*, **BYTE** *cStrikeOut*, **BYTE** *nCharSet*, **BYTE** *nOutPrecision*, **BYTE** *nClipPrecision*, **BYTE** *nQuality*, **BYTE** *nPitchAndFamily*, **LPCSTR** *lpszFacename* **);**

nHeight Specifies the desired height (in logical units) of the font. The font height can be specified in the following ways:

- Greater than 0, in which case the height is transformed into device units and matched against the cell height of the available fonts.
- Equal to 0, in which case a reasonable default size is used.
- Less than 0, in which case the height is transformed into device units and the absolute value is matched against the character height of the available fonts.

The absolute value of *nHeight* must not exceed 16,384 device units after it is converted. For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size or the smallest font if all the fonts exceed the requested size.

nWidth Specifies the average width (in logical units) of characters in the font. If *nWidth* is 0, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts to find the closest match, which is determined by the absolute value of the difference.

nEscapement Specifies the angle (in 0.1-degree units) between the escapement vector and the x-axis of the display surface. The escapement vector is the line through the origins of the first and last characters on a line. The angle is measured counterclockwise from the x-axis.

nOrientation Specifies the angle (in 0.1-degree units) between the baseline of a character and the x-axis. The angle is measured counterclockwise from the x-axis for coordinate systems in which the y-direction is down and clockwise from the x-axis for coordinate systems in which the y-direction is up.

nWeight Specifies the font weight (in inked pixels per 1000). The common constants are as follows (*nWeight* can be any integer value from 0 to 1000):

Constant	Value
FW_DONTCARE	0
FW_THIN	100

Constant	Value
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

These values are approximate; the actual appearance depends on the typeface. Some fonts have only **FW_NORMAL**, **FW_REGULAR**, and **FW_BOLD** weights. If **FW_DONTCARE** is specified, a default weight is used.

bItalic Specifies whether the font is italic.

bUnderline Specifies whether the font is underlined.

cStrikeOut Specifies whether characters in the font are struck out. Specifies a strikeout font if set to a nonzero value.

nCharSet Specifies the font's character set. The following constants and values are predefined:

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

The OEM character set is system-dependent.

Fonts with other character sets may exist in the system. An application that uses a font with an unknown character set must not attempt to translate or interpret strings that are to be rendered with that font. Instead, the strings should be passed directly to the output device driver.

The font mapper does not use the **DEFAULT_CHARSET** value. An application can use this value to allow the name and size of a font to fully describe the logical font. If a font with the specified name does not exist, a font from any character set can be substituted for the specified font. To avoid unexpected results, applications should use the **DEFAULT_CHARSET** value sparingly.

nOutPrecision   Specifies the desired output precision. The output precision defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. It can be any one of the following values:

OUT_CHARACTER_PRECIS
OUT_DEFAULT_PRECIS
OUT_DEVICE_PRECIS
OUT_RASTER_PRECIS
OUT_STRING_PRECIS
OUT_STROKE_PRECIS
OUT_TT_PRECIS

Applications can use the **OUT_DEVICE_PRECIS**, **OUT_RASTER_PRECIS**, and **OUT_TT_PRECIS** values to control how the font mapper chooses a font when the system contains more than one font with a given name. For example, if a system contains a font named Symbol in raster and TrueType form, specifying **OUT_TT_PRECIS** forces the font mapper to choose the TrueType version. (Specifying **OUT_TT_PRECIS** forces the font mapper to choose a TrueType font whenever the specified font name matches a device or raster font, even when there is no TrueType font of the same name.)

nClipPrecision   Specifies the desired clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. It can be any one of the following values:

CLIP_CHARACTER_PRECIS
CLIP_DEFAULT_PRECIS
CLIP_ENCAPSULATE
CLIP_LH_ANGLES
CLIP_MASK
CLIP_STROKE_PRECIS
CLIP_TT_ALWAYS

To use an embedded read-only font, an application must specify **CLIP_ENCAPSULATE**.

To achieve consistent rotation of device, TrueType, and vector fonts, an application can use the OR operator to combine the **CLIP_LH_ANGLES** value with any of the other *nClipPrecision* values. If the **CLIP_LH_ANGLES** bit is set, the rotation for all fonts depends on whether the orientation of the coordinate system is left-handed or right-handed. (For more information about the orientation of coordinate systems, see the description of the *nOrientation* parameter.) If **CLIP_LH_ANGLES** is not set, device fonts always rotate counterclockwise, but the rotation of other fonts is dependent on the orientation of the coordinate system.

nQuality Specifies the font's output quality, which defines how carefully the GDI must attempt to match the logical-font attributes to those of an actual physical font. It can be one of the following values, with the meaning as given:

- **DEFAULT_QUALITY** Appearance of the font does not matter.
- **DRAFT_QUALITY** Appearance of the font is less important than when **PROOF_QUALITY** is used. For GDI raster fonts, scaling is enabled. Bold, italic, underline, and strikeout fonts are synthesized if necessary.
- **PROOF_QUALITY** Character quality of the font is more important than exact matching of the logical-font attributes. For GDI raster fonts, scaling is disabled and the font closest in size is chosen. Bold, italic, underline, and strikeout fonts are synthesized if necessary.

nPitchAndFamily Specifies the pitch and family of the font. The two low-order bits specify the pitch of the font and can be any one of the following values:

DEFAULT_PITCH	**VARIABLE_PITCH**
FIXED_PITCH	

Applications can add **TMPF_TRUETYPE** to the *nPitchAndFamily* parameter to choose a TrueType font. The four high-order bits of the parameter specify the font family and can be one of the following values, with the meaning as given:

- **FF_DECORATIVE** Novelty fonts. Old English, for example.
- **FF_DONTCARE** Don't care or don't know.
- **FF_MODERN** Fonts with constant stroke width (fixed-pitch), with or without serifs. Fixed-pitch fonts are usually modern faces. Pica, Elite, and Courier New are examples.
- **FF_ROMAN** Fonts with variable stroke width (proportionally spaced) and with serifs. Times New Roman and Century Schoolbook are examples.
- **FF_SCRIPT** Fonts designed to look like handwriting. Script and Cursive are examples.
- **FF_SWISS** Fonts with variable stroke width (proportionally spaced) and without serifs. MS Sans Serif is an example.

An application can specify a value for *nPitchAndFamily* by using the Boolean OR operator to join a pitch constant with a family constant.

Font families describe the look of a font in a general way. They are intended for specifying fonts when the exact typeface desired is not available.

lpszFacename A **CString** or pointer to a null-terminated string that specifies the typeface name of the font. The length of this string must not exceed 30 characters. The Windows **EnumFontFamilies** function can be used to enumerate all currently available fonts. If *lpszFacename* is **NULL**, the GDI uses a device-independent typeface.

Remarks

Initializes a **CFont** object with the specified characteristics. The font can subsequently be selected as the font for any device context. The **CreateFont** function does not create a new Windows GDI font. It merely selects the closest match from the fonts available in the GDI's pool of physical fonts. Applications can use the default settings for most of these parameters when creating a logical font. The parameters that should always be given specific values are *nHeight* and *lpszFacename*. If *nHeight* and *lpszFacename* are not set by the application, the logical font that is created is device-dependent.

When you finish with the **CFont** object created by the **CreateFont** function, first select the font out of the device context, then delete the **CFont** object.

Return Value

Nonzero if successful; otherwise 0.

See Also

CFont::CreateFontIndirect, **::CreateFont**, **::EnumFontFamilies**, **::EnumFonts**

CFont::CreateFontIndirect

BOOL CreateFontIndirect(const LOGFONT FAR* *lpLogFont* **);**

lpLogFont Points to a **LOGFONT** structure that defines the characteristics of the logical font.

Remarks

Initializes a **CFont** object with the characteristics given in a **LOGFONT** structure pointed to by *lpLogFont*. The font can subsequently be selected as the current font for any device. This font has the characteristics specified in the **LOGFONT** structure. When the font is selected by using the **CDC::SelectObject** or **CMetaFileDC::SelectObject** member function, the GDI's font mapper attempts to match the logical font with an existing physical font. If it fails to find an exact match for the logical font, it provides an alternative whose characteristics match as many of the requested characteristics as possible.

When you finish with the **CFont** object created by the **CreateFontIndirect** function, first select the font out of the device context, then delete the **CFont** object.

Return Value

Nonzero if successful; otherwise 0.

LOGFONT Structure

The **LOGFONT** structure has the following form:

```
typedef struct tagLOGFONT {
    int   lfHeight;
    int   lfWidth;
    int   lfEscapement;
    int   lfOrientation;
    int   lfWeight;
    BYTE  lfItalic;
    BYTE  lfUnderline;
    BYTE  lfStrikeOut;
    BYTE  lfCharSet;
    BYTE  lfOutPrecision;
    BYTE  lfClipPrecision;
    BYTE  lfQuality;
    BYTE  lfPitchAndFamily;
    BYTE  lfFaceName[LF_FACESIZE];
} LOGFONT;
```

For more complete information about this structure see **LOGFONT** in the *Microsoft Windows Software Development Kit* documentation.

See Also

CFont::CreateFont, **CDC::SelectObject**, **CGdiObject::DeleteObject**, **CMetaFileDC::SelectObject**, **::CreateFontIndirect**

CFont::FromHandle

static CFont* PASCAL FromHandle(HFONT *hFont* **);**

hFont An **HFONT** handle to a Windows font.

Remarks

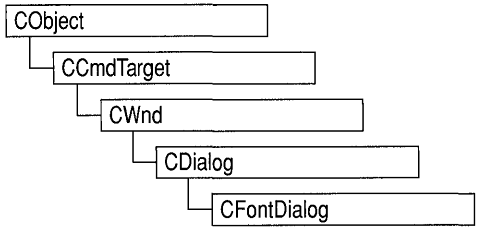
Returns a pointer to a **CFont** object when given an **HFONT** handle to a Windows GDI font object. If a **CFont** object is not already attached to the handle, a temporary **CFont** object is created and attached. This temporary **CFont** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

Return Value

A pointer to a **CFont** object if successful; otherwise **NULL**.

class CFontDialog : public CDialog

The **CFontDialog** class allows you to incorporate a font-selection dialog box into your application. A **CFontDialog** object is a dialog box with a list of fonts that are currently installed in the system. The user can select a particular font from the list, and this selection is then reported back to the application.



To construct a **CFontDialog** object, use the provided constructor or derive a new subclass and use your own custom constructor.

Once a **CFontDialog** object has been constructed, you can use the **m_cf** structure to initialize the values or states of controls in the dialog box. The **m_cf** structure is of type **CHOOSEFONT**. For more information on this structure, see the *Windows Software Development Kit* (SDK) documentation.

After initializing the dialog object's controls, call the **DoModal** member function to display the dialog box and allow the user to select a font. **DoModal** returns whether the user selected the OK (**IDOK**) or Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CFontDialog**'s member functions to retrieve the information input by the user.

You can use the Windows **CommDlgExtendedError** function to determine if an error occurred during initialization of the dialog box to learn more about the error. For more information on this function, see the Windows SDK documentation.

CFontDialog relies on the COMMDLG.DLL file that ships with Windows version 3.1. For details about redistributing COMMDLG.DLL to Windows version 3.0 users, see the *Getting Started* manual for the Windows version 3.1 SDK.

To customize the dialog box, derive a class from **CFontDialog**, provide a custom dialog template, and add a message-map to process the notification messages from the extended controls. Any unprocessed messages should be passed to the base class.

Customizing the hook function is not required.

#include <afxdlgs.h>

Data Members — Public Members

m_cf	A structure used to customize a **CFontDialog** object.

Construction/Destruction — Public Members

CFontDialog	Constructs a **CFontDialog** object.

Operations — Public Members

DoModal	Displays the dialog box and allows the user to make a selection.
GetCurrentFont	Retrieves the name of the currently selected font.
GetFaceName	Returns the face name of the selected font.
GetStyleName	Returns the style name of the selected font.
GetSize	Returns the point size of the selected font.
GetColor	Returns the color of the selected font.
GetWeight	Returns the weight of the selected font.
IsStrikeOut	Determines if the font is displayed with strikeout.
IsUnderline	Determines if the font is underlined.
IsBold	Determines if the font is bold.
IsItalic	Determines if the font is italic.

Member Functions

CFontDialog::CFontDialog

CFontDialog(LPLOGFONT *lplfInitial* **= NULL,**
DWORD *dwFlags* **= CF_EFFECTS | CF_SCREENFONTS,**
CDC* *pdcPrinter* **= NULL,**
CWnd* *pParentWnd* **= NULL);**

lplfInitial A pointer to a **LOGFONT** data structure that allows you to set some of the font's characteristics. The **LOGFONT** type is defined in WINDOWS.H as follows:

```
typedef struct tagLOGFONT
  {
    int       lfHeight;
    int       lfWidth;
    int       lfEscapement;
    int       lfOrientation;
    int       lfWeight;
    BYTE      lfItalic;
    BYTE      lfUnderline;
    BYTE      lfStrikeOut;
    BYTE      lfCharSet;
    BYTE      lfOutPrecision;
    BYTE      lfClipPrecision;
    BYTE      lfQuality;
    BYTE      lfPitchAndFamily;
    BYTE      lfFaceName[LF_FACESIZE];
  } LOGFONT;
```

For more information on the **LOGFONT** structure, see the Windows SDK documentation.

dwFlags Specifies one or more choose-font flags. One or more preset values can be combined using the bitwise-OR operator. If you modify the **m_ofn.Flags** structure member, be sure to use a bitwise-OR operator in your changes to keep the default behavior intact. For details on each of these flags, see the description of the **CHOOSEFONT** structure in the Windows SDK documentation.

pdcPrinter A pointer to a printer-device context. If supplied, this parameter points to a printer-device context for the printer on which the fonts are to be selected.

pParentWnd A pointer to the font dialog box's parent or owner window.

Remarks Constructs a **CFontDialog** object.

See Also **CFontDialog::DoModal**

CFontDialog::DoModal

virtual int DoModal( );

Remarks

Call this function to display the Windows common font dialog box and allow the user to choose a font.

If you want to initialize the various font dialog controls by setting members of the **m_cf** structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

If **DoModal** returns **IDOK**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

Return Value

IDOK or **IDCANCEL** if the function is successful; otherwise 0. **IDOK** and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

If **IDCANCEL** is returned, you can call the Windows **CommDlgExtendedError** function to determine if an error occurred.

See Also

CDialog::DoModal, **CFontDialog::CFontDialog**

CFontDialog::GetColor

COLORREF GetColor( ) const;

Return Value

The color of the selected font.

See Also

CFontDialog::GetCurrentFont

CFontDialog::GetCurrentFont

void GetCurrentFont(LPLOGFONT *lplf* **);**

lplf A pointer to a **LOGFONT** structure.

Remarks

Assigns the characteristics of the currently selected font to the members of a **LOGFONT** structure. For more information on the **LOGFONT** structure, see the

Windows SDK documentation. Other **CFontDialog** member functions are provided to access individual characteristics of the current font.

See Also **CFontDialog::GetFaceName**, **CFontDialog::GetStyleName**

CFontDialog::GetFaceName

CString GetFaceName( ) const;

Return Value The face name of the font selected in the **CFontDialog** dialog box.

See Also **CFontDialog::GetCurrentFont**, **CFontDialog::GetStyleName**

CFontDialog::GetSize

int GetSize( ) const;

Return Value The font's point size.

See Also **CFontDialog::GetWeight**, **CFontDialog::GetCurrentFont**

CFontDialog::GetStyleName

CString GetStyleName( ) const;

Return Value The style name of the font.

See Also **CFontDialog::GetFaceName**, **CFontDialog::GetCurrentFont**

CFontDialog::GetWeight

int GetWeight( ) const;

Return Value The weight of the selected font.

See Also **CFontDialog::GetCurrentFont**, **CFontDialog::IsBold**

CFontDialog::IsBold

BOOL IsBold( ) const;

Return Value Nonzero if the selected font has the Bold characteristic enabled; otherwise 0.

See Also **CFontDialog::GetCurrentFont**

CFontDialog::IsItalic

BOOL IsItalic( ) const;

Return Value Nonzero if the selected font has the Italic characteristic enabled; otherwise 0.

See Also **CFontDialog::GetCurrentFont**

CFontDialog::IsStrikeOut

BOOL IsStrikeOut( ) const;

Return Value Nonzero if the selected font has the Strikeout characteristic enabled; otherwise 0.

See Also **CFontDialog::GetCurrentFont**

CFontDialog::IsUnderline

BOOL IsUnderline( ) const;

Return Value Nonzero if the selected font has the Underline characteristic enabled; otherwise 0.

See Also **CFontDialog::GetCurrentFont**

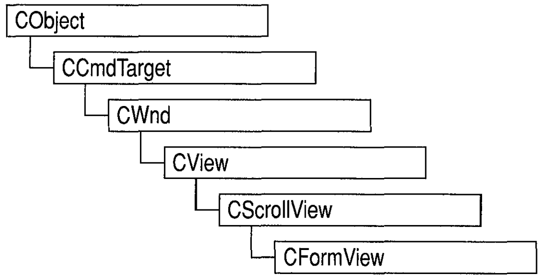
Data Members

CFontDialog::m_cf

Remarks

A structure whose members store the characteristics of the dialog object. After constructing a **CFontDialog** object, you can use **m_cf** to initialize various values in the dialog box. You must initialize the dialog box's values before calling the **Create** member function. For more information on this structure, see **CHOOSEFONT** in the Windows SDK documentation.

class CFormView : public CScrollView



The **CFormView** class is the base class used for views containing controls. These controls are laid out based on a dialog-template resource. Use **CFormView** if you want form-based documents in your application. These views support scrolling, as needed, using the **CScrollView** functionality.

Creating a view based on **CFormView** is similar to creating a dialog box. To use **CFormView**, take the following steps:

1. Design a dialog template.

   Use the App Studio dialog editor to design the dialog box. Then, in the Styles property page, set the following properties:

   - In the Style box, select Child (**WS_CHILD** on).
   - In the Border box, select None (**WS_BORDER** off).
   - Clear the Visible check box (**WS_VISIBLE** off).
   - Clear the Titlebar check box (**WS_CAPTION** off).

   These steps are necessary because a form view is not a true dialog box. For more information about creating a dialog-box resource using App Studio, see Chapter 3, "Using the Dialog Editor," in the *App Studio User's Guide*.

2. Create a view class.

   With your dialog template open, invoke ClassWizard and choose **CFormView** as the class type when you are filling in the Add Class dialog box. ClassWizard creates a **CFormView**-derived class and connects it to the dialog template you just designed. This connection is established in the constructor for your class; ClassWizard generates a call to the base-class constructor, **CFormView::CFormView**, and passes the resource ID of your dialog template. For example:

```
CMyFormView::CMyFormView()
    : CFormView(CMyFormView::IDD)
{
    //{{AFX_DATA_INIT(CMyFormView)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT

    // Other construction code, such as data initialization
}
```

Note If you choose not to use ClassWizard, you must define the appropriate ID you supply to the **CFormView** constructor (that is, `CMyFormView::IDD` is not predefined). ClassWizard declares `IDD` as an **enum** value in the class it creates for you.

If you want to define member variables in your view class that correspond to the controls in your form view, use the Edit Variables button in the ClassWizard dialog box. This allows you to use the dialog data exchange (DDX) mechanism. If you want to define message handlers for control-notification messages, use the Add Function button in the ClassWizard dialog box. For more information on using ClassWizard, see Chapters 6 and 7 of the *Class Library User's Guide* or Chapter 9 of the *App Studio User's Guide*.

3. Override the **OnUpdate** member function.

   The **OnUpdate** member function is defined by **CView** and is called to update the form view's appearance. Override this function to update the member variables in your view class with the appropriate values from the current document. Then, if you are using DDX, use the **UpdateData** member function defined by **CWnd** to update the controls in your form view.

   The **OnInitialUpdate** member function (also defined by **CView**) is called to perform one-time initialization of the view. **CFormView** overrides this function to use DDX to set the initial values of the controls you have mapped using ClassWizard. Override **OnInitialUpdate** if you want to perform custom initialization.

4. Implement a member function to move data from your view to your document.

   This member function is typically a message handler for a control-notification message or for a menu command. If you are using DDX, call the **UpdateData** member function to update the member variables in your view class. Then move their values to the document associated with the form view.

5. Override the **OnPrint** member function (optional).

   The **OnPrint** member function is defined by **CView** and prints the view. By default, printing and print preview are not supported by the **CFormView** class. To add printing support, override the **OnPrint** function in your derived class. See the VIEWEX sample for more information about how to add printing capabilities to a view derived from **CFormView**.

6. Associate your view class with a document class and a frame-window class using a document template.

Unlike ordinary views, form views do not require you to override the **OnDraw** member function defined by **CView**. This is because controls are able to paint themselves. Only if you want to customize the display of your form view (for example, to provide a background for your view) should you override **OnDraw**. If you do so, be careful that your updating does not conflict with the updating done by the controls.

If the view becomes smaller than the dialog template, scroll bars appear automatically. Views derived from **CFormView** support only the **MM_TEXT** mapping mode.

If you are not using DDX, use the **CWnd** dialog functions to move data between the member variables in your view class and the controls in your form view.

For more information about DDX, see Chapter 7 of the *Class Library User's Guide* or Chapter 5 in this manual.

#include <afxext.h>

See Also **CDialog**, **CScrollView**, **CView::OnUpdate**, **CView::OnInitialUpdate**, **CView::OnPrint**, **CWnd::UpdateData**, **CScrollView::ResizeParentToFit**

Construction/Destruction—Protected Members

CFormView	Constructs a **CFormView** object.

Member Functions

CFormView::CFormView

CFormView(LPCSTR *lpszTemplateName* **);**

CFormView(UINT *nIDTemplate* **);**

lpszTemplateName Contains a null-terminated string that is the name of a dialog-template resource.

nIDTemplate Contains the ID number of a dialog-template resource.

Remarks

When you create an object of a type derived from **CFormView**, invoke one of the constructors to create the view object and identify the dialog resource on which the view is based. You can either identify the resource by name (pass a string as the argument to the constructor) or by its ID (pass an unsigned integer as the argument).

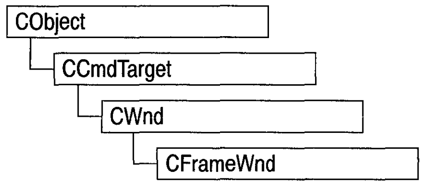
The form-view window and child controls are not created until **CWnd::Create** is called. **CWnd::Create** is called by the framework as part of the document and view creation process, which is driven by the document template.

Note Your derived class *must* supply its own constructor. In the constructor, invoke the constructor, **CFormView::CFormView**, with the resource name or ID as an argument as shown in the preceding class overview.

See Also

CWnd::Create

class CFrameWnd : public CWnd



The **CFrameWnd** class provides the functionality of a Windows single document interface (SDI) overlapped or pop-up frame window, along with members for managing the window. To create a useful frame window for your application, derive a class from **CFrameWnd**. Add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window. There are three ways to construct a frame window:

- Directly construct it using **Create**.
- Directly construct it using **LoadFrame**.
- Indirectly construct it using a document template.

Before you call either **Create** or **LoadFrame**, you must construct the frame-window object on the heap using the C++ **new** operator. Before calling **Create**, you may also register a window class with the **AfxRegisterWndClass** global function to set the icon and class styles for the frame.

Use the **Create** member function to pass the frame's creation parameters as immediate arguments.

LoadFrame requires fewer arguments than **Create**, and instead retrieves most of its default values from resources, including the frame's caption, icon, accelerator table, and menu. To be accessible by **LoadFrame**, all these resources must have the same resource ID (for example, **IDR_MAINFRAME**).

When a **CFrameWnd** object contains views and documents, they are created indirectly by the framework instead of directly by the programmer. The **CDocTemplate** object orchestrates the creation of the frame, the creation of the containing views, and the connection of the views to the appropriate document. The parameters of the **CDocTemplate** constructor specify the **CRuntimeClass** of the three classes involved (document, frame, and view). A **CRuntimeClass** object is used by the framework to dynamically create new frames when specified by the user (for example, by using the File New command or the multiple document interface [MDI] Window New command).

A frame-window class derived from **CFrameWnd** must be declared with **DECLARE_DYNCREATE** in order for the above **RUNTIME_CLASS** mechanism to work correctly.

A **CFrameWnd** contains default implementations to perform the following functions of a main window in a typical application for Windows:

- A **CFrameWnd** frame window keeps track of a currently active view that is independent of the Windows active window or the current input focus. When the frame is reactivated, the active view is notified by calling **CView::OnActivateView**.
- Command messages and many common frame-notification messages, including those handled by the **OnSetFocus**, **OnHScroll**, and **OnVScroll** functions of **CWnd**, are delegated by a **CFrameWnd** frame window to the currently active view.
- The currently active view (or currently active MDI child frame window in the case of an MDI frame) can determine the caption of the frame window. This feature can be disabled by turning off the **FWS_ADDTOTITLE** style bit of the frame window.
- A **CFrameWnd** frame window manages the positioning of the control bars, views, and other child windows inside the frame window's client area. A frame window also does idle-time updating of toolbar and other control-bar buttons. A **CFrameWnd** frame window also has default implementations of commands for toggling on and off the toolbar and status bar.
- A **CFrameWnd** frame window manages the main menu bar. When a pop-up menu is displayed, the frame window uses the **UPDATE_COMMAND_UI** mechanism to determine which menu items should be enabled, disabled, or checked. When the user selects a menu item, the frame window updates the status bar with the message string for that command.
- A **CFrameWnd** frame window has an optional accelerator table that automatically translates keyboard accelerators.
- A **CFrameWnd** frame window has an optional help ID set with **LoadFrame** that is used for context-sensitive help. A frame window is the main orchestrator of semimodal states such as context-sensitive help (SHIFT+F1) and print-preview modes.
- A **CFrameWnd** frame window will open a file dragged from the File Manager and dropped on the frame window. If a file extension is registered and associated with the application, the frame window responds to the dynamic data exchange (DDE) open request that occurs when the user opens a data file in the File Manager or when the **ShellExecute** Windows function is called.
- If the frame window is the main application window (that is, **CWinApp::m_pMainWnd**), when the user closes the application, the frame window prompts the user to save any modified documents (for **OnClose** and **OnQueryEndSession**).

- If the frame window is the main application window, the frame window is the context for running WinHelp. Closing the frame window will shut down WINHELP.EXE if it was launched for help for this application.

Do not use the C++ **delete** operator to destroy a frame window. Use **CWnd::DestroyWindow** instead. The **CFrameWnd** implementation of **PostNcDestroy** will delete the C++ object when the window is destroyed. When the user closes the frame window, the default **OnClose** handler will call **DestroyWindow**.

#include <afxwin.h>

See Also **CWnd**, **CMDIFrameWnd**, **CMDIChildWnd**

Data Members—Public Members

m_bAutoMenuEnable	Controls automatic enable and disable functionality for menu items.
rectDefault	Pass this static **CRect** as a parameter when creating a **CFrameWnd** object to allow Windows to choose the window's initial size and position.

Construction/Destruction—Public Members

CFrameWnd	Constructs a **CFrameWnd** object.

Initialization—Public Members

Create	Call to create and initialize the Windows frame window associated with the **CFrameWnd** object.
LoadFrame	Call to dynamically create a frame window from resource information.
LoadAccelTable	Call to load an accelerator table.

Operations—Public Members

ActivateFrame	Makes the frame visible and available to the user.
SetActiveView	Sets the active **CView** object.
GetActiveView	Returns the active **CView** object.
GetActiveDocument	Returns the active **CDocument** object.
RecalcLayout	Repositions control bars.

Overridables—Public Members

OnSetPreviewMode	Sets the application's main frame window into and out of print-preview mode.

Overridables—Protected Members

OnCreateClient	Creates a client window for the frame.

Member Functions

CFrameWnd::ActivateFrame

virtual void ActivateFrame(int *nCmdShow* = **–1);**

nCmdShow Specifies the parameter to pass to **CWnd::ShowWindow**. By default, the frame is shown and correctly restored.

Remarks

Call this member function to activate and restore the frame window so that it is visible and available to the user. This member function is usually called after a non-user interface event such as a DDE, Object Linking and Embedding (OLE), or other event that may show the frame window or its contents to the user.

The default implementation activates the frame and brings it to the top of the Z-order and, if necessary, carries out the same steps for the application's main frame window.

Override this member function to change how a frame is activated. For example, you can force MDI child windows to be maximized. Add the appropriate functionality, then call the base class version with an explicit *nCmdShow*.

CFrameWnd::CFrameWnd

CFrameWnd( );

Remarks

Constructs a **CFrameWnd** object, but doesn't create the visible frame window. Call **Create** to create the visible window.

See Also

CFrameWnd::Create, **CFrameWnd::LoadFrame**

CFrameWnd::Create

BOOL Create(LPCSTR *lpszClassName*, **LPCSTR** *lpszWindowName*,
DWORD *dwStyle* = **WS_OVERLAPPEDWINDOW,**
const RECT& *rect* = **rectDefault, CWnd*** *pParentWnd* = **NULL,**
LPCSTR *lpszMenuName* = **NULL, DWORD** *dwExStyle* = **0,**
CCreateContext* *pContext* = **NULL);**

lpszClassName Points to a null-terminated character string that names the Windows class. The class name can be any name registered with the **AfxRegisterWndClass** global function or the **RegisterClass** Windows function. If **NULL**, uses the predefined default **CFrameWnd** attributes.

lpszWindowName Points to a null-terminated character string that represents the window name. Used as text for the title bar.

dwStyle Specifies the window style attributes. Include the **FWS_ADDTOTITLE** style if you want the title bar to automatically display the name of the document represented in the window.

See the **CWnd::Create** member function on page 904 for a full list of window styles.

rect Specifies the size and position of the window. The **rectDefault** value allows the Windows operating system to specify the size and position of the new window.

pParentWnd Specifies the parent window of this frame window. This parameter should be **NULL** for top-level frame windows.

lpszMenuName Identifies the name of the menu resource to be used with the window. Use **MAKEINTRESOURCE** if the menu has an integer ID instead of a string. This parameter can be **NULL**.

dwExStyle Specifies the window extended style attributes.

See the **CWnd::CreateEx** member function on page 907 for a list of extended window styles.

pContext Specifies a pointer to a **CCreateContext** structure. This parameter can be **NULL**.

Remarks

Construct a **CFrameWnd** object in two steps. First invoke the constructor, which constructs the **CFrameWnd** object, then call **Create**, which creates the Windows frame window and attaches it to the **CFrameWnd** object. **Create** initializes the window's class name and window name and registers default values for its style, parent, and associated menu.

Use **LoadFrame** rather than **Create** to load the frame window from a resource instead of specifying its arguments.

Return Value Nonzero if initialization is successful; otherwise 0.

See Also **CFrameWnd::CFrameWnd**, **CFrameWnd::LoadFrame**, **CCreateContext**, **CWnd::Create**, **CWnd::PreCreateWindow**

CFrameWnd::GetActiveDocument

virtual CDocument* GetActiveDocument( );

Remarks Call this member function to obtain a pointer to the current **CDocument** attached to the current active view.

Return Value A pointer to the current **CDocument**. If there is no current document, returns **NULL**.

See Also **CFrameWnd::GetActiveView**

CFrameWnd::GetActiveView

CView* GetActiveView( ) const;

Remarks Call this member function to obtain a pointer to the active view.

Return Value A pointer to the current **CView**. If there is no current view, returns **NULL**.

See Also **CFrameWnd::SetActiveView**, **CFrameWnd::GetActiveDocument**

CFrameWnd::LoadAccelTable

BOOL LoadAccelTable(LPCSTR *lpszResourceName* **);**

lpszResourceName Identifies the name of the accelerator resource. Use **MAKEINTRESOURCE** if the resource is identified with an integer ID.

Remarks

Call to load the specified accelerator table. Only one table may be loaded at a time. Accelerator tables loaded from resources are freed automatically when the application terminates.

If you call **LoadFrame** to create the frame window, the framework loads an accelerator table along with the menu and icon resources, and a subsequent call to this member function is then unnecessary.

Return Value

Nonzero if the accelerator table was successfully loaded; otherwise 0.

See Also

CFrameWnd::LoadFrame, **::LoadAccelerators**

CFrameWnd::LoadFrame

virtual BOOL LoadFrame(UINT *nIDResource*, **DWORD** *dwDefaultStyle* = **WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE, CWnd*** *pParentWnd* = **NULL, CCreateContext*** *pContext* = **NULL);**

nIDResource The ID of shared resources associated with the frame window.

dwDefaultStyle The frame's style. Include the **FWS_ADDTOTITLE** style if you want the title bar to automatically display the name of the document represented in the window.

See the **CWnd::Create** member function on page 904 for a full list of window styles.

pParentWnd A pointer to the frame's parent.

pContext A pointer to a **CCreateContext** structure. This parameter can be **NULL**.

Remarks

Construct a **CFrameWnd** object in two steps. First invoke the constructor, which constructs the **CFrameWnd** object, then call **LoadFrame**, which loads the Windows frame window and associated resources and attaches the frame window to the **CFrameWnd** object. The *nIDResource* parameter specifies the menu, the accelerator table, the icon, and the string resource of the title for the frame window.

Use the **Create** member function rather than **LoadFrame** when you want to specify all of the frame window's creation parameters.

The framework calls **LoadFrame** when it creates a frame window using a document template object.

The framework uses the *pContext* argument to specify the objects to be connected to the frame window, including any contained view objects. You can set the *pContext* argument to **NULL** when you call **LoadFrame**.

See Also **CDocTemplate**, **CFrameWnd::Create**, **CFrameWnd::CFrameWnd**, **CWnd::PreCreateWindow**

CFrameWnd::OnCreateClient

Protected **virtual BOOL OnCreateClient(LPCREATESTRUCT** *lpcs*, **CCreateContext*** *pContext* **);** ♦

lpcs A pointer to a Windows **CREATESTRUCT** structure.

pContext A pointer to a **CCreateContext** structure.

Remarks Called by the framework during the execution of **OnCreate**. Never call this function.

The default implementation of this function creates a **CView** object from the information provided in *pContext*, if possible.

Override this function to override values passed in the **CCreateContext** object or to change the way controls in the main client area of the frame window are created. The **CCreateContext** members you can override are described in the **CCreateContext** class.

Note Do not replace values passed in the **CREATESTRUCT** structure. They are for informational use only. If you want to override the initial window rectangle, for example, override the **CWnd** member function **PreCreateWindow**.

CFrameWnd::OnSetPreviewMode

virtual void OnSetPreviewMode(BOOL *bPreview*, **CPrintPreviewState*** *pModeStuff* **);**

bPreview Specifies whether or not to place the application in print-preview mode. Set to **TRUE** to place in print preview, **FALSE** to restore to cancel the preview mode.

pModeStuff A pointer to a **CPrintPreviewState** structure.

Remarks

Call this member function to set the application's main frame window into and out of print-preview mode.

The default implementation disables all standard toolbars and hides the main menu and the main client window. This turns MDI frame windows into temporary SDI frame windows.

Override this member function to customize the hiding and showing of control bars and other frame window parts during print preview. Call the base class implementation from within the overridden version.

CFrameWnd::RecalcLayout

virtual void RecalcLayout( );

Remarks

Call this member function to reposition control bars after changing the layout of the frame window. For example, call it when you turn on or off control bars or add another control bar. Called by the framework when the standard control bars are toggled on or off or when the frame window is resized. The default implementation of this member function calls the **CWnd** member function **RepositionBars** to reposition all the control bars in the frame as well as the main client window (usually a **CView** or **MDICLIENT**).

See Also

CWnd::RepositionBars

CFrameWnd::SetActiveView

void SetActiveView(CView* *pViewNew* **);**

pViewNew Specifies a pointer to a **CView** object, or **NULL** for no active view.

Remarks

Call this member function to set the active view. The framework will call this function automatically as the user changes the focus to a view within the frame window. You may explictly call **SetActiveView** to change the focus to the specified view.

See Also

CFrameWnd::GetActiveView, **CView::OnActivateView**, **CFrameWnd::GetActiveDocument**

Data Members

CFrameWnd::m_bAutoMenuEnable

Remarks

When this data member is enabled (which is the default), menu items that don't have **ON_UPDATE_COMMAND_UI** or **ON_COMMAND** handlers will be automatically disabled when the user pulls down a menu. Menu items that have an **ON_COMMAND** handler but no **ON_UPDATE_COMMAND_UI** handler will be automatically enabled. When this data member is set, menu items are automatically enabled in the same way that toolbar buttons are enabled.

This data member simplifies the implementation of optional commands based on the current selection and reduces the need for an application to write **ON_UPDATE_COMMAND_UI** handlers for enabling and disabling menu items.

See Also

CCmdUI, **CCmdTarget**

CFrameWnd::rectDefault

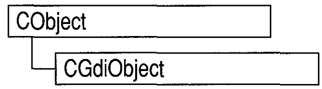
Remarks

Pass this static **CRect** as a parameter when creating a window to allow Windows to choose the window's initial size and position.

See Also

CW_USEDEFAULT

class CGdiObject : public CObject



The **CGdiObject** class provides a base class for various kinds of Windows graphics device interface (GDI) objects such as bitmaps, regions, brushes, pens, palettes, and fonts. You never create a **CGdiObject** directly. Rather, you create an object from one of its derived classes, such as **CPen** or **CBrush**.

#include <afxwin.h>

See Also **CBitmap**, **CBrush**, **CFont**, **CPalette**, **CPen**, **CRgn**

Data Members—Public Members

m_hObject	A **HANDLE** containing the **HBITMAP**, **HPALETTE**, **HRGN**, **HBRUSH**, **HPEN**, or **HFONT** attached to this object.

Construction/Destruction—Public Members

CGdiObject	Constructs a **CGdiObject** object.

Operations—Public Members

GetSafeHandle	Returns **m_hObject** unless **this** is **NULL**, in which case **NULL** is returned.
FromHandle	Returns a pointer to a **CGdiObject** object given a handle to a Windows GDI object.
Attach	Attaches a Windows GDI object to a **CGdiObject** object.
Detach	Detaches a Windows GDI object from a **CGdiObject** object and returns a handle to the Windows GDI object.
DeleteObject	Deletes the Windows GDI object attached to the **CGdiObject** object from memory by freeing all system storage associated with the object.
DeleteTempMap	Deletes any temporary **CGdiObject** objects created by **FromHandle**.
GetObject	Fills a buffer with data that describes the Windows GDI object attached to the **CGdiObject** object.
CreateStockObject	Retrieves a handle to one of the Windows predefined stock pens, brushes, or fonts.
UnrealizeObject	Resets the origin of a brush or resets a logical palette.

Member Functions

CGdiObject::Attach

BOOL Attach(HGDIOBJ *hObject* **);**

hObject A **HANDLE** to a Windows GDI object (for example, **HPEN** or **HBRUSH**).

Remarks Attaches a Windows GDI object to a **CGdiObject** object.

Return Value Nonzero if attachment is successful; otherwise 0.

See Also **CGdiObject::Detach**

CGdiObject::CGdiObject

CGdiObject( );

Remarks Constructs a **CGdiObject** object. You never create a **CGdiObject** directly. Rather, you create an object from one of its derived classes, such as **CPen** or **CBrush**.

See Also **CPen**, **CBrush**, **CFont**, **CBitmap**, **CRgn**, **CPalette**

CGdiObject::CreateStockObject

BOOL CreateStockObject(int *nIndex* **);**

nIndex A constant specifying the type of stock object desired. It can be one of the following values, with the meanings as given:

- **BLACK_BRUSH** Black brush.
- **DKGRAY_BRUSH** Dark gray brush.
- **GRAY_BRUSH** Gray brush.

- **HOLLOW_BRUSH** Hollow brush.
- **LTGRAY_BRUSH** Light gray brush.
- **NULL_BRUSH** Null brush.
- **WHITE_BRUSH** White brush.
- **BLACK_PEN** Black pen.
- **NULL_PEN** Null pen.
- **WHITE_PEN** White pen.
- **ANSI_FIXED_FONT** ANSI fixed system font.
- **ANSI_VAR_FONT** ANSI variable system font.
- **DEVICE_DEFAULT_FONT** Device-dependent font.
- **OEM_FIXED_FONT** OEM-dependent fixed font.
- **SYSTEM_FONT** The system font. By default, Windows uses the system font to draw menus, dialog-box controls, and other text. In Windows versions 3.0 and later, the system font is proportional width; earlier versions of Windows use a fixed-width system font.
- **SYSTEM_FIXED_FONT** The fixed-width system font used in Windows prior to version 3.0. This object is available for compatibility with earlier versions of Windows.
- **DEFAULT_PALETTE** Default color palette. This palette consists of the 20 static colors in the system palette.

Remarks Retrieves a handle to one of the predefined stock Windows GDI pens, brushes, or fonts, and attaches the GDI object to the **CGdiObject** object. Call this function with one of the derived classes that corresponds to the Windows GDI object type, such as **CPen** for a stock pen.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CPen::CPen**, **CBrush::CBrush**, **CFont::CFont**, **CPalette::CPalette**

CGdiObject::DeleteObject

BOOL DeleteObject( );

Remarks Deletes the attached Windows GDI object from memory by freeing all system storage associated with the Windows GDI object. The storage associated with the **CGdiObject** object is not affected by this call. An application should not call

DeleteObject on a **CGdiObject** object that is currently selected into a device context. When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted independently.

See Also **CGdiObject::Detach**

CGdiObject::DeleteTempMap

static void PASCAL DeleteTempMap( );

Remarks Called automatically by the **CWinApp** idle-time handler, **DeleteTempMap** deletes any temporary **CGdiObject** objects created by **FromHandle**. **DeleteTempMap** detaches the Windows GDI object attached to a temporary **CGdiObject** object before deleting the **CGdiObject** object.

See Also **CGdiObject::Detach**, **CGdiObject::FromHandle**

CGdiObject::Detach

HGDIOBJ Detach( );

Remarks Detaches a Windows GDI object from a **CGdiObject** object and returns a handle to the Windows GDI object.

Return Value A **HANDLE** to the Windows GDI object detached; otherwise **NULL** if no GDI object is attached.

See Also **CGdiObject::Attach**

CGdiObject::FromHandle

static CGdiObject* PASCAL FromHandle(HGDIOBJ *hObject* **);**

hObject A **HANDLE** to a Windows GDI object.

Remarks Returns a pointer to a **CGdiObject** object given a handle to a Windows GDI object. If a **CGdiObject** object is not already attached to the Windows GDI object, a temporary **CGdiObject** object is created and attached. This temporary

CGdiObject object is only valid until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

Return Value A pointer to a **CGdiObject** that may be temporary or permanent.

See Also **CGdiObject::DeleteTempMap**

CGdiObject::GetObject

int GetObject(int *nCount*, **LPVOID** *lpObject* **) const;**

nCount Specifies the number of bytes to copy into the *lpObject* buffer.

lpObject Points to a user-supplied buffer that is to receive the information.

Remarks Fills a buffer with data that defines a specified object. The function retrieves a data structure whose type depends on the type of graphic object, as shown by the following list:

Object	Buffer type
CPen	**LOGPEN**
CBrush	**LOGBRUSH**
CFont	**LOGFONT**
CBitmap	**BITMAP**
CPalette	**int**
CRgn	Not supported

If the object is a **CBitmap** object, **GetObject** returns only the width, height, and color format information of the bitmap. The actual bits can be retrieved by using **CBitmap::GetBitmapBits**. If the object is a **CPalette** object, **GetObject** retrieves an integer that specifies the number of entries in the palette. The function does not retrieve the **LOGPALETTE** structure that defines the palette. An application can get information on palette entries by calling **CPalette::GetPaletteEntries**.

Return Value The number of bytes retrieved; otherwise 0 if an error occurs.

See Also **CBitmap::GetBitmapBits**, **CPalette::GetPaletteEntries**

CGdiObject::GetSafeHandle

HGDIOBJ GetSafeHandle( ) const;

Remarks

Returns **m_hObject** unless **this** is **NULL**, in which case **NULL** is returned. This is part of the general handle interface paradigm and is useful when **NULL** is a valid or special value for a handle.

Return Value

A **HANDLE** to the attached Windows GDI object; **NULL** if no object is attached.

CGdiObject::UnrealizeObject

BOOL UnrealizeObject( );

Remarks

Resets the origin of a brush or resets a logical palette. While **UnrealizeObject** is a member function of the **CGdiObject** class, it should be invoked only on **CBrush** or **CPalette** objects. For **CBrush** objects, **UnrealizeObject** directs the system to reset the origin of the given brush the next time it is selected into a device context. If the object is a **CPalette** object, **UnrealizeObject** directs the system to realize the palette as though it had not previously been realized. The next time the application calls the **CDC::RealizePalette** function for the specified palette, the system completely remaps the logical palette to the system palette. The **UnrealizeObject** function should not be used with stock objects. The **UnrealizeObject** function must be called whenever a new brush origin is set (by means of the **CDC::SetBrushOrg** function). The **UnrealizeObject** function must not be called for the currently selected brush or currently selected palette of any display context.

Return Value

Nonzero if successful; otherwise 0.

See Also

CDC::RealizePalette, CDC::SetBrushOrg

Data Members

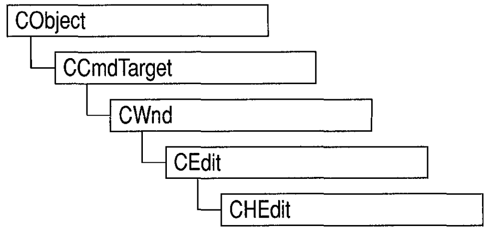
CGdiObject::m_hObject

Remarks

A **HANDLE** containing the **HBITMAP**, **HRGN**, **HBRUSH**, **HPEN**, **HPALETTE**, or **HFONT** attached to this object.

class CHEdit : public CEdit

The **CHEdit** class encapsulates the functionality of the handwriting edit, or "hedit," control in Microsoft Windows for Pen Computing. This control has all the functionality of a normal keyboard-based edit control. It also allows for handwriting recognition.



An application built with the application framework detects pen-equipped systems and, by default, registers them as pen enabled. When your application starts up on one of these systems, all edit controls support general handwriting recognition.

If you have information—such as the type of input expected—that can simplify the handwriting recognizer's task, you should use **CHEdit** controls, then set the alphabet code (ALC) style for the kind of data you are expecting. The more narrowly you define the type of data expected, the better the recognition algorithms work. Note that if you have a fixed-length entry field, **CBEdit** controls can help the recognizer understand *where* to expect the user to input data.

Take the following steps to create a **CHEdit** control using App Studio:

1. Create a user-defined control in your dialog box.
2. In the Caption field, enter **ALC<*x*>**, where *x* is a number obtained by combining the desired ALC styles using the bitwise-OR operator.

   The following table shows the values and corresponding common ALC styles allowed for **CHEdit** controls:

Value	ALC Style
1	Lowercase
2	Uppercase
3	Uppercase or Lowercase
4	Numeric
8	Punctuation
16	Mathematical symbols
32	Monetary symbols
64	Other

3. In the Class field, enter "hedit" (or "bedit" if you are creating a boxed edit control).

4. In the Style field, enter the hexadecimal number obtained by combining the desired edit styles from the table below using the bitwise-OR operator. The four most-significant hexadecimal digits should remain 0x5001 for a visible child window with the tab-stop property set.

   The following table shows a subset of the edit-control styles allowed for **CHEdit** controls (for a complete set of styles, see "Edit Styles" in **CEdit::Create**):

Hexadecimal Value	Meaning
0x0001	Center text in control
0x0002	Right align text in control
0x0004	Multiline edit control
0x0008	Uppercase text only
0x0010	Lowercase text only

If you want to handle Windows notification messages sent by a **CHEdit** control to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler function to the parent class for each message.

You will typically add entries for the notifications generated by a standard **CEdit** object. These notification handlers are identical to **CEdit** notification handlers.

Each message-map entry takes the following form:

ON_CONTROL(*notification-message*, *id*, *memberFxn*)

where *notification-message* specifies the notification message you want to handle, *id* specifies the child-window ID of the control sending the notification, and *memberFxn* specifies the name of the parent member function you have written to handle the notification.

The *memberFxn* prototype for these notification handlers is as follows:

afx_msg void memberFxn( );

The following is a list of applicable notification messages specific to **CHEdit** objects:

- **HN_ENDREC** The current recognition context was closed. The call to the recognizer for recognition has terminated.
- **HN_DELAYEDRECOGFAIL** Delayed recognition has failed. The attempted recognition was initiated by an application through the member function **StopInkMode**, or by the user's tapping on a control.
- **HN_RCRESULT** The hedit control has received a **WM_RCRESULT** message from the recognizer.

#include <afxpen.h>

Construction/Destruction — Public Members

CHEdit	Constructs a **CHEdit** object.
Create	Creates a **CHEdit** control.

Operations — Public Members

GetInflate	Gets the inflation rectangle (the rectangle in which handwriting is recognized).
GetInkHandle	Gets a handle to captured ink.
GetRC	Gets a pointer to a recognition context.
GetUnderline	Returns the state of the underline mode.
SetInflate	Sets the inflation rectangle (the rectangle in which handwriting is recognized).
SetInkMode	Starts the collection of inking.
SetRC	Sets a pointer to a recognition context.
SetUnderline	Sets the underline mode.
StopInkMode	Stops the collection of ink.

Member Functions

CHEdit::CHEdit

CHEdit( );

Remarks Constructs a **CHEdit** object.

See Also **CHEdit::Create**

CHEdit::Create

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **)**

dwStyle Specifies the hedit control's style. See **CEdit::Create** for a list of these styles.

rect Specifies the hedit control's boxed rectangle. Note that the area sensitive to pen gestures and inking can be modified using **SetInflate**.

pParentWnd Specifies the hedit control's parent window (usually derived from **CDialog**). It must not be **NULL**.

nID Specifies the edit control ID.

Remarks

You construct a **CHEdit** object in two steps. First, construct the **CHEdit** object, then call **Create**, which creates the Windows hedit control and attaches it to the **CHEdit** object. To extend the default message handling, derive a class from **CHEdit**, add a message map to the new class, and override the appropriate message-handler member functions.

Return Value

Nonzero if initialization is successful; otherwise 0.

See Also

CEdit::Create, **CHEdit::CHEdit**, **CHEdit::SetInflate**

CHEdit::GetInflate

BOOL GetInflate(LPRECTOFS *lpRectOfs* **);**

lpRectOfs A far pointer to a **RECTOFS** structure object that receives the inflation offsets. This structure is described in the "RECTOFS Structure" section that follows.

Remarks

The returned structure contains offsets from the top, left, bottom, and right sides of the client rectangle rather than the location or dimensions of the rectangle. Both positive and negative values are legal for the members of the *lpRectOfs* argument.

Return Values

Nonzero if successful; otherwise 0.

RECTOFS Structure

A **RECTOFS** structure has this form:

```
typedef struct tagRECTOFS
{
    int dLeft;
    int dTop;
    int dRight;
    int dBottom;
} RECTOFS;
```

A **RECTOFS** structure contains a list of offsets from the top, left, bottom, and right boundaries of the client area of the control. Handwriting is recognized in the rectangle defined by the client rectangle and modified by these offsets. Positive values for any member indicate that the rectangle should be enlarged (or inflated), and negative values indicate that the rectangle should be reduced.

Members

dLeft Offset from left side of client rectangle.

dTop Offset from top of client rectangle.

dRight Offset from right side of client rectangle.

dBottom Offset from bottom of client rectangle.

Comments

In addition to having the basic characteristics of an edit control, the hedit or bedit control must make allowances for the input of handwriting. The client rectangle often needs to be adjusted to a larger size to allow for easier writing.

For example, the Delete gesture typically extends above the selected text it is deleting. If the gesture is arbitrarily clipped off at the edge of the client window, recognition accuracy suffers. Likewise, restricting handwriting input to stay within the lines can also hinder recognition accuracy. To correct this, rectangle offsets are used in the hedit and bedit controls to make the writing area slightly larger than the client window size of a normal edit control. The **GetInflate** and **SetInflate** member functions are used to get and set the inflation rectangle.

The inflation need not be symmetrical in every direction (that is, you can inflate one side of the rectangle more than another).

See Also

CHEdit::SetInflate, WM_HEDITCTL

CHEdit::GetInkHandle

HPENDATA GetInkHandle( );

Remarks Obtains a handle to captured ink. If you expect to use this data after the hedit control is destroyed, you must duplicate this handle because the control's copy is invalidated on destruction.

Return Value A handle to the ink entered by the user. If the control is not in ink mode, **GetInkHandle** returns **NULL**.

See Also **::GetPenDataInfo**, **WM_HEDITCTL**

CHEdit::GetRC

BOOL GetRC(LPRC *lpRC* **);**

lpRC A far pointer to an **RC** structure. For a detailed description of the **RC** structure, see *Microsoft Windows for Pen Computing: Programmer's Reference*.

Remarks Retrieves the current recognition context.

Return Value Nonzero if successful; otherwise 0.

See Also **CHEdit::SetRC**, **WM_HEDITCTL**

CHEdit::GetUnderline

BOOL GetUnderline( );

Remarks Gets the underline mode.

Return Value Nonzero if underline mode is set; 0 if underline mode is not set.

See Also **CHEdit::SetUnderline**, **WM_HEDITCTL**

CHEdit::SetInflate

BOOL SetInflate(LPRECTOFS *lpRectOfs* **);**

lpRectOfs A far pointer to a **RECTOFS** structure object that specifies the inflation offsets. See **GetInflate** for a description of the **RECTOFS** structure.

Remarks The structure specifies offsets from the top, left, bottom, and right sides of the client rectangle rather than the location or dimensions of the rectangle. Both positive and negative values are legal for the members of the *lpRectOfs* parameter.

Return Value Nonzero if successful; otherwise 0.

See Also **CHEdit::GetInflate**, **WM_HEDITCTL**

CHEdit::SetInkMode

BOOL SetInkMode(HPENDATA *hPenDataInitial* **= NULL);**

hPenDataInitial A handle to the initial pen data.

Remarks Starts the collection of inking. You can specify *hPenDataInitial* or allow it to default to **NULL**. If you specify this data, all offsets must be relative to the top-left corner of the client rectangle of the hedit control.

Return Value Nonzero if successful; otherwise 0.

See Also **CHEdit::GetInkHandle**, **CHEdit::StopInkMode**, **WM_HEDITCTL**

CHEdit::SetRC

BOOL SetRC(LPRC *lpRC* **);**

lpRC A far pointer to an **RC** structure. For a detailed description of this structure, see *Microsoft Windows for Pen Computing: Programmer's Reference.*

Remarks Sets a new recognition context. You might, for example, change the recognition context to specify numeric values and gestures only (which allows it to ignore the difference between the letter "O" and the number "0"). The **SetRC** function can be used in conjunction with the **GetRC** function to change one member of the recognition context.

Return Value Nonzero if successful; otherwise 0.

See Also **CHEdit::GetRC**, **WM_HEDITCTL**

CHEdit::SetUnderline

BOOL SetUnderline(BOOL *bUnderline* **= TRUE);**

bUnderline If **TRUE**, underline mode is turned on.

Remarks Sets the underline mode. Note that to use the underline mode, the hedit control's border must be off. That is, the **WS_BORDER** bit of the hedit control must be off.

Return Value Nonzero if successful; otherwise 0.

See Also **CHEdit::GetUnderline**, **WM_HEDITCTL**

CHEdit::StopInkMode

BOOL StopInkMode(UINT *hep* **);**

hep The action to take after stopping the collection of ink. These actions can be:

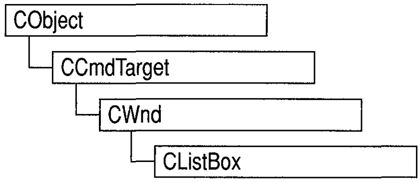
- **HEP_RECOG** Perform recognition and display the text
- **HEP_NORECOG** Remove the ink without performing the recognition
- **HEP_WAITFORTAP** Perform recognition on next tap in the control

Remarks Stops the collection of ink and specifies the next action for the recognizer.

Return Value Nonzero if successful; otherwise 0.

See Also **CHEdit::SetInkMode**, **CHEdit::GetInkHandle**, **WM_HEDITCTL**

class CListBox : public CWnd



The **CListBox** class provides the functionality of a Windows list box. A list box displays a list of items, such as filenames, that the user can view and select. In a single-selection list box, the user can select only one item. In a multiple-selection list box, a range of items can be selected. When the user selects an item, it is highlighted and the list box sends a notification message to the parent window. The list box itself automatically displays horizontal or vertical scroll bars if the list within the box is too large for the list-box window.

You can create a list box either from a dialog template or directly in your code. In both cases, call the constructor **CListBox** to construct the **CListBox** object, then call the **Create** member function to create the Windows list-box control and attach it to the **CListBox** object. Construction can be a one-step process in a class derived from **CListBox**. Write a constructor for the derived class and call **Create** from within the constructor. If you want to handle Windows notification messages sent by a list box to its parent (usually a class derived from **CDialog**), add a message-map entry and message-handler member function to the parent class for each message.

Each message-map entry takes the following form:

ON_Notification(*id*, *memberFxn*)

where *id* specifies the child window ID of the list-box control sending the notification and *memberFxn* is the name of the parent member function you have written to handle the notification.

The parent's function prototype is as follows:

afx_msg void memberFxn( );

Following is a list of potential message-map entries and a description of the cases in which they would be sent to the parent:

- **ON_LBN_DBLCLK**   The user double-clicks a string in a list box. Only a list box that has the **LBS_NOTIFY** style will send this notification message.
- **ON_LBN_ERRSPACE**   The list box cannot allocate enough memory to meet the request.
- **ON_LBN_KILLFOCUS**   The list box is losing the input focus.

Windows 3.1 Only

- **ON_LBN_SELCANCEL**   The current list-box selection is cancelled. This message is only sent when a list box has the **LBS_NOTIFY** style. ♦

- **ON_LBN_SELCHANGE** The selection in the list box is about to change. This notification is not sent if the selection is changed by the **CListBox::SetCurSel** member function. This notification applies only to a list box that has the **LBS_NOTIFY** style. The **LBN_SELCHANGE** notification message is sent for a multiple-selection list box whenever the user presses an arrow key, even if the selection does not change.
- **ON_LBN_SETFOCUS** The list box is receiving the input focus.

If you create a **CListBox** object within a dialog box (through a dialog resource), the **CListBox** object is automatically destroyed when the user closes the dialog box. If you create a **CListBox** object within a window, you may need to destroy the **CListBox** object. If you create the **CListBox** object on the stack, it is destroyed automatically. If you create the **CListBox** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user terminates the Windows list box. If you allocate any memory in the **CListBox** object, override the **CListBox** destructor to dispose of the allocations.

#include <afxwin.h>

See Also **CWnd**, **CButton**, **CComboBox**, **CEdit**, **CScrollBar**, **CStatic**, **CDialog**

Construction/Destruction—Public Members

CListBox	Constructs a **CListBox** object.

Initialization—Public Members

Create	Creates the Windows list box and attaches it to the **CListBox** object.

General Operations—Public Members

GetCount	Returns the number of strings in a list box.
GetHorizontalExtent	Returns the width in pixels that a list box can be scrolled horizontally.
SetHorizontalExtent	Sets the width in pixels that a list box can be scrolled horizontally.
GetTopIndex	Returns the index of the first visible string in a list box.
SetTopIndex	Sets the zero-based index of the first visible string in a list box.
GetItemData	Returns the 32-bit value associated with the list-box item.
GetItemDataPtr	Returns a pointer to a list-box item.
SetItemData	Sets the 32-bit value associated with the list-box item.

SetItemDataPtr	Sets a pointer to the list-box item.
GetItemRect	Returns the bounding rectangle of the list-box item as it is currently displayed.
SetItemHeight	Sets the height of items in a list box.
GetItemHeight	Determines the height of items in a list box.
GetSel	Returns the selection state of a list-box item.
GetText	Copies a list-box item into a buffer.
GetTextLen	Returns the length in bytes of a list-box item.
SetColumnWidth	Sets the column width of a multicolumn list box.
SetTabStops	Sets the tab-stop positions in a list box.

Single-Selection Operations—Public Members

GetCurSel	Returns the zero-based index of the currently selected string in a list box.
SetCurSel	Selects a list-box string.

Multiple-Selection Operations—Public Members

SetSel	Selects or deselects a list-box item in a multiple-selection list box.
GetCaretIndex	Determines the index of the item that has the focus rectangle in a multiple-selection list box.
SetCaretIndex	Set the focus rectangle to the item at the specified index in a multiple-selection list box.
GetSelCount	Returns the number of strings currently selected in a multiple-selection list box.
GetSelItems	Returns the indices of the strings currently selected in a list box.
SelItemRange	Selects or deselects a range of strings in a multiple-selection list box.

String Operations—Public Members

AddString	Adds a string to a list box.
DeleteString	Deletes a string from a list box.
InsertString	Inserts a string at a specific location in a list box.
ResetContent	Clears all the entries from a list box.
Dir	Adds filenames from the current directory to a list box.
FindString	Searches for a string in a list box.

FindStringExact	Finds the first list-box string that matches a specified string.
SelectString	Searches for and selects a string in a single-selection list box.

Overridables—Public Members

DrawItem	Called by the framework when a visual aspect of an owner-draw list box changes.
MeasureItem	Called by the framework when an owner-draw list box is created to determine list-box dimensions.
CompareItem	Called by the framework to determine the position of a new item in a sorted owner-draw list box.
DeleteItem	Called by the framework when the user deletes an item from an owner-draw list box.

Member Functions

CListBox::AddString

int AddString(LPCSTR *lpszItem* **);**

lpszItem Points to the null-terminated string that is to be added.

Remarks

Call this member function to add a string to a list box. If the list box was not created with the **LBS_SORT** style, the string is added to the end of the list. Otherwise, the string is inserted into the list, and the list is sorted. If the list box was created with the **LBS_SORT** style but not the **LBS_HASSTRINGS** style, the framework sorts the list by one or more calls to the **CompareItem** member function. Use **InsertString** to insert a string into a specific location within the list box.

Return Value

The zero-based index to the string in the list box. The return value is **LB_ERR** if an error occurs; the return value is **LB_ERRSPACE** if insufficient space is available to store the new string.

See Also

CListBox::InsertString, **CListBox::CompareItem**, **LB_ADDSTRING**

CListBox::CListBox

CListBox( );

Remarks

You construct a **CListBox** object in two steps. First call the constructor **CListBox**, then call **Create**, which initializes the Windows list box and attaches it to the **CListBox**.

See Also

CListBox::Create

CListBox::CompareItem

virtual int CompareItem(LPCOMPAREITEMSTRUCT *lpCompareItemStruct* **);**

lpCompareItemStruct A long pointer to a **COMPAREITEMSTRUCT** structure.

Remarks

Called by the framework to determine the relative position of a new item in a sorted owner-draw list box. By default, this member function does nothing. If you create an owner-draw list box with the **LBS_SORT** style, you must override this member function to assist the framework in sorting new items added to the list box.

Return Value

Indicates the relative position of the two items described in the **COMPAREITEMSTRUCT** structure. It may be any of the following values:

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

See **CWnd::OnCompareItem** on page 956 for a description of the **COMPAREITEMSTRUCT** structure.

See Also

WM_COMPAREITEM, **CWnd::OnCompareItem**, **CListBox::DrawItem**, **CListBox::MeasureItem**, **CListBox::DeleteItem**

CListBox::Create

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

dwStyle Specifies the style of the list box.

rect Specifies the list-box size and position. Can be either a **CRect** object or a **RECT** structure.

pParentWnd Specifies the list box's parent window (usually a **CDialog** or **CModalDialog** object). It must not be **NULL**.

nID Specifies the list box's control ID.

Remarks

You construct a **CListBox** object in two steps. First call the constructor, then call **Create**, which initializes the Windows list box and attaches it to the **CListBox** object. When **Create** executes, Windows sends the **WM_NCCREATE**, **WM_CREATE**, **WM_NCCALCSIZE**, and **WM_GETMINMAXINFO** messages to the list-box control. These messages are handled by default by the **OnNcCreate**, **OnCreate**, **OnNcCalcSize**, and **OnGetMinMaxInfo** member functions in the **CWnd** base class. To extend the default message handling, derive a class from **CListBox**, add a message map to the new class, and override the preceding message-handler member functions. Override **OnCreate**, for example, to perform needed initialization for a new class.

Apply the following window styles to a list-box control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_VSCROLL** To add a vertical scroll bar
- **WS_HSCROLL** To add a horizontal scroll bar
- **WS_GROUP** To group controls
- **WS_TABSTOP** To allow tabbing to this control

See the **Create** member function in the **CWnd** base class for a full description of these window styles.

Return Value

Nonzero if successful; otherwise 0.

List-Box Styles

You can use any combination of the following list-box styles for *dwStyle*:

- **LBS_EXTENDEDSEL** The user can select multiple items using the SHIFT key and the mouse or special key combinations.

- **LBS_HASSTRINGS** Specifies an owner-draw list box that contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the **GetText** member function to retrieve the text for a particular item.
- **LBS_MULTICOLUMN** Specifies a multicolumn list box that is scrolled horizontally. The **SetColumnWidth** member function sets the width of the columns.
- **LBS_MULTIPLESEL** String selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
- **LBS_NOINTEGRALHEIGHT** The size of the list box is exactly the size specified by the application when it created the list box. Usually, Windows sizes a list box so that the list box does not display partial items.
- **LBS_NOREDRAW** List-box display is not updated when changes are made. This style can be changed at any time by sending a **WM_SETREDRAW** message.
- **LBS_NOTIFY** Parent window receives an input message whenever the user clicks or double-clicks a string.
- **LBS_OWNERDRAWFIXED** The owner of the list box is responsible for drawing its contents; the items in the list box are the same height.
- **LBS_OWNERDRAWVARIABLE** The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
- **LBS_SORT** Strings in the list box are sorted alphabetically.
- **LBS_STANDARD** Strings in the list box are sorted alphabetically, and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.
- **LBS_USETABSTOPS** Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** Windows function returns the current dialog base units in pixels.)
- **LBS_WANTKEYBOARDINPUT** The owner of the list box receives **WM_VKEYTOITEM** or **WM_CHARTOITEM** messages whenever the user presses a key while the list box has input focus. This allows an application to perform special processing on the keyboard input.

Windows 3.1 Only

- **LBS_DISABLENOSCROLL** The list box shows a disabled vertical scroll bar when the list box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items. ♦

See Also **CListBox::CListBox**

CListBox::DeleteItem

virtual void DeleteItem(LPDELETEITEMSTRUCT *lpDeleteItemStruct* **);**

lpDeleteItemStruct A long pointer to a Windows **DELETEITEMSTRUCT** structure that contains information about the deleted item.

Remarks Called by the framework when the user deletes an item from an owner-draw **CListBox** object or destroys the list box. The default implementation of this function does nothing. Override this function to redraw an owner-draw list box as needed.

See **CWnd::OnDeleteItem** on page 961 for a description of the **DELETEITEMSTRUCT** structure.

See Also **CListBox::CompareItem**, **CWnd::OnDeleteItem**, **CListBox::DrawItem**, **CListBox::MeasureItem**, **::DeleteItem**

CListBox::DeleteString

int DeleteString(UINT *nIndex* **);**

nIndex Specifies the zero-based index of the string to be deleted.

Remarks Deletes an item in a list box.

Return Value A count of the strings remaining in the list. The return value is **LB_ERR** if *nIndex* specifies an index greater then the number of items in the list.

See Also **LB_DELETESTRING**, **CListBox::AddString**, **CListBox::InsertString**

CListBox::Dir

int Dir(UINT *attr*, **LPCSTR** *lpszWildCard* **);**

attr Can be any combination of the **enum** values described in **CFile::GetStatus**, or any combination of the following values:

Value	Meaning
0x0000	File can be read from or written to.
0x0001	File can be read from but not written to.
0x0002	File is hidden and does not appear in a directory listing.
0x0004	File is a system file.
0x0010	The name specified by *lpszWildCard* specifies a directory.
0x0020	File has been archived.
0x4000	Include all drives that match the name specified by *lpszWildCard*.
0x8000	Exclusive flag. If the exclusive flag is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to "normal" files.

lpszWildCard Points to a file-specification string. The string can contain wildcards (for example, *.*).

Remarks Adds a list of filenames and/or drives to a list box.

Return Value The zero-based index of the last filename added to the list. The return value is **LB_ERR** if an error occurs; the return value is **LB_ERRSPACE** if insufficient space is available to store the new strings.

See Also **CWnd::DlgDirList**, **LB_DIR**, **CFile::GetStatus**

CListBox::DrawItem

virtual void DrawItem(LPDRAWITEMSTRUCT *lpDrawItemStruct* **);**

lpDrawItemStruct A long pointer to a **DRAWITEMSTRUCT** structure that contains information about the type of drawing required.

Remarks Called by the framework when a visual aspect of an owner-draw list box changes. The member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed.

By default, this member function does nothing. Override this member function to implement drawing for an owner-draw **CListBox** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before this member function terminates.

See **CWnd::OnDrawItem** on page 964 for a description of the **DRAWITEMSTRUCT** structure.

See Also **CListBox::CompareItem**, **CWnd::OnDrawItem**, **::DrawItem**, **CListBox::MeasureItem**, **CListBox::DeleteItem**

CListBox::FindString

int FindString(int *nStartAfter*, **LPCSTR** *lpszItem* **) const;**

nStartAfter Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If *nStartAfter* is –1, the entire list box is searched from the beginning.

lpszItem Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string may contain any combination of uppercase and lowercase letters.

Remarks Finds the first string in a list box that contains the specified prefix without changing the list-box selection. Use the **SelectString** member function to both find and select a string.

Return Value The zero-based index of the matching item, or **LB_ERR** if the search was unsuccessful.

See Also **CListBox::SelectString**, **CListBox::AddString**, **CListBox::InsertString**, **LB_FINDSTRING**

CListBox::FindStringExact

Windows 3.1 Only **int FindStringExact(int** *nIndexStart*, **LPCSTR** *lpszFind* **) const;** ♦

nIndexStart Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nIndexStart*. If *nIndexStart* is –1, the entire list box is searched from the beginning.

lpszFind Points to the null-terminated string to search for. This string can contain a complete filename, including the extension. The search is not case sensitive, so the string can contain any combination of uppercase and lowercase letters.

Remarks An application calls the **FindStringExact** member function to find the first list-box string that matches the string specified in *lpszFind*. If the list box was created with an owner-draw style but without the **LBS_HASSTRINGS** style, the **FindStringExact** member function attempts to match the doubleword value against the value of *lpszFind*.

Return Value The index of the matching item, or **LB_ERR** if the search was unsuccessful.

See Also **CListBox::FindString**, **LB_FINDSTRING**, **LB_FINDSTRINGEXACT**

CListBox::GetCaretIndex

Windows 3.1 Only **int GetCaretIndex( ) const;** ♦

Remarks An application calls the **GetCaretIndex** member function to determine the index of the item that has the focus rectangle in a multiple-selection list box. The item may or may not be selected.

Return Value The zero-based index of the item that has the focus rectangle in a list box. If the list box is a single-selection list box, the return value is the index of the item that is selected, if any.

See Also **CListBox::SetCaretIndex**, **LB_GETCARETINDEX**

CListBox::GetCount

int GetCount( ) const;

Remarks Retrieves the number of items in a list box. The returned count is one greater than the index value of the last item (the index is zero-based).

Return Value The number of items in the list box, or **LB_ERR** if an error occurs.

See Also **LB_GETCOUNT**

CListBox::GetCurSel

int GetCurSel( ) const;

Remarks Retrieves the zero-based index of the currently selected item, if any, in a single-selection list box. **GetCurSel** should not be called for a multiple-selection list box.

Return Value The zero-based index of the currently selected item. It is **LB_ERR** if no item is currently selected or if the list box is a multiple-selection list box.

See Also **LB_GETCURSEL**, **CListBox::SetCurSel**

CListBox::GetHorizontalExtent

int GetHorizontalExtent( ) const;

Remarks Retrieves from a list box the width in pixels by which the list box can be scrolled horizontally if the list box has horizontal scroll bars. To respond to **GetHorizontalExtent**, the list box must have been defined with the **WS_HSCROLL** style.

Return Value The scrollable width of the list box, in pixels.

See Also **CListBox::SetHorizontalExtent**, **LB_GETHORIZONTALEXTENT**

CListBox::GetItemData

DWORD GetItemData(int *nIndex* **) const;**

nIndex Specifies the zero-based index of the item in the list box.

Remarks Retrieves the application-supplied doubleword value associated with the specified list-box item. The doubleword value was the *dwItemData* parameter of a **SetItemData** call.

Return Value The 32-bit value associated with the item, or **LB_ERR** if an error occurs.

See Also **CListBox::AddString**, **CListBox::GetItemDataPtr**, **CListBox::SetItemDataPtr**, **CListBox::InsertString**, **CListBox::SetItemData**, **LB_GETITEMDATA**

CListBox::GetItemDataPtr

void* GetItemDataPtr(int *nIndex* **) const;**

nIndex Specifies the zero-based index of the item in the list box.

Remarks

Retrieves the application-supplied 32-bit value associated with the specified list-box item as a pointer (**void***).

Return Value

Retrieves a pointer, or –1 if an error occurs.

See Also

CListBox::AddString, **CListBox::GetItemData**, **CListBox::InsertString**, **CListBox::SetItemData**, **LB_GETITEMDATA**

CListBox::GetItemHeight

Windows 3.1 Only

int GetItemHeight(int *nIndex* **) const;** ♦

nIndex Specifies the zero-based index of the item in the list box. This parameter is used only if the list box has the **LBS_OWNERDRAWVARIABLE** style; otherwise, it should be set to 0.

Remarks

An application calls the **GetItemHeight** member function to determine the height of items in a list box.

Return Value

The height, in pixels, of the items in the list box. If the list box has the **LBS_OWNERDRAWVARIABLE** style, the return value is the height of the item specified by *nIndex*. If an error occurs, the return value is **LB_ERR**.

See Also

LB_GETITEMHEIGHT, **CListBox::SetItemHeight**

CListBox::GetItemRect

int GetItemRect(int *nIndex*, **LPRECT** *lpRect* **) const;**

nIndex Specifies the zero-based index of the item.

lpRect Specifies a long pointer to a **RECT** data structure that receives the list-box client coordinates of the item.

Remarks Retrieves the dimensions of the rectangle that bounds a list-box item as it is currently displayed in the list-box window.

Return Value **LB_ERR** if an error occurs.

See Also **LB_GETITEMRECT**

CListBox::GetSel

int GetSel(int *nIndex* **) const;**

nIndex Specifies the zero-based index of the item.

Remarks Retrieves the selection state of an item. This member function works with both single- and multiple-selection list boxes.

Return Value A positive number if the specified item is selected; otherwise, it is 0. The return value is **LB_ERR** if an error occurs.

See Also **LB_GETSEL**, **CListBox::SetSel**

CListBox::GetSelCount

int GetSelCount( ) const;

Remarks Retrieves the total number of selected items in a multiple-selection list box.

Return Value The count of selected items in a list box. If the list box is a single-selection list box, the return value is **LB_ERR**.

See Also **CListBox::SetSel**, **LB_GETSELCOUNT**

CListBox::GetSelItems

int GetSelItems(int *nMaxItems*, **LPINT** *rgIndex* **) const;**

nMaxItems Specifies the maximum number of selected items whose item numbers are to be placed in the buffer.

rgIndex Specifies a long pointer to a buffer large enough for the number of integers specified by *nMaxItems*.

Remarks Fills a buffer with an array of integers that specifies the item numbers of selected items in a multiple-selection list box.

Return Value The actual number of items placed in the buffer. If the list box is a single-selection list box, the return value is **LB_ERR**.

See Also **LB_GETSELITEMS**

CListBox::GetText

int GetText(int *nIndex*, **LPSTR** *lpszBuffer* **) const;**

void GetText(int *nIndex*, **CString&** *rString* **) const;**

nIndex Specifies the zero-based index of the string to be retrieved.

lpszBuffer Points to the buffer that receives the string. The buffer must have sufficient space for the string and a terminating null character. The size of the string can be determined ahead of time by calling the **GetTextLen** member function.

rString A reference to a **CString** object.

Remarks Gets a string from a list box. The second form of this member function fills a **CString** object with the string text.

Return Value The length (in bytes) of the string, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **LB_ERR**.

See Also **CListBox::GetTextLen**, **LB_GETTEXT**

CListBox::GetTextLen

int GetTextLen(int *nIndex* **) const;**

nIndex Specifies the zero-based index of the string.

Remarks Gets the length of a string in a list-box item.

Return Value The length of the string in bytes, excluding the terminating null character. If *nIndex* does not specify a valid index, the return value is **LB_ERR**.

See Also **CListBox::GetText**, **LB_GETTEXTLEN**

CListBox::GetTopIndex

int GetTopIndex( ) const;

Remarks Retrieves the zero-based index of the first visible item in a list box. Initially, item 0 is at the top of the list box, but if the list box is scrolled, another item may be at the top.

Return Value The zero-based index of the first visible item in a list box.

See Also **CListBox::SetTopIndex**, **LB_GETTOPINDEX**

CListBox::InsertString

int InsertString(int *nIndex***, LPCSTR** *lpszItem* **);**

nIndex Specifies the zero-based index of the position to insert the string. If this parameter is –1, the string is added to the end of the list.

lpszItem Points to the null-terminated string that is to be inserted.

Remarks Inserts a string into the list box. Unlike the **AddString** member function, **InsertString** does not cause a list with the **LBS_SORT** style to be sorted.

Return Value The zero-based index of the position at which the string was inserted. The return value is **LB_ERR** if an error occurs; the return value is **LB_ERRSPACE** if insufficient space is available to store the new string.

See Also **CListBox::AddString**, **LB_INSERTSTRING**

CListBox::MeasureItem

virtual void MeasureItem(LPMEASUREITEMSTRUCT *lpMeasureItemStruct* **);**

lpMeasureItemStruct A long pointer to a **MEASUREITEMSTRUCT** structure.

Remarks

Called by the framework when a list box with an owner-draw style is created.

By default, this member function does nothing. Override this member function and fill in the **MEASUREITEMSTRUCT** structure to inform Windows of the list-box dimensions. If the list box is created with the **LBS_OWNERDRAWVARIABLE** style, the framework calls this member function for each item in the list box. Otherwise, this member is called only once.

For further information about using the **OWNERDRAWFIXED** style in an owner-draw list box created with the **SubclassDlgItem** member function of **CWnd**, see the discussion in Technical Note 14 in MSVC\HELP\MFCNOTES.HLP.

See **CWnd::OnMeasureItem** on page 980 for a description of the **MEASUREITEMSTRUCT** structure.

See Also

CListBox::CompareItem, **CWnd::OnMeasureItem**, **CListBox::DrawItem**, **::MeasureItem**, **CListBox::DeleteItem**

CListBox::ResetContent

void ResetContent( );

Remarks

Removes all items from a list box.

See Also

LB_RESETCONTENT

CListBox::SelectString

int SelectString(int *nStartAfter*, **LPCSTR** *lpszItem* **);**

nStartAfter Contains the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by *nStartAfter*. If *nStartAfter* is –1, the entire list box is searched from the beginning.

lpszItem Points to the null-terminated string that contains the prefix to search for. The search is case independent, so this string may contain any combination of uppercase and lowercase letters.

Remarks

Searches for a list-box item that matches the specified string, and if a matching item is found, it selects the item. The list box is scrolled, if necessary, to bring the selected item into view. This member function cannot be used with a list box that has the **LBS_MULTIPLESEL** style. An item is selected only if its initial characters (from the starting point) match the characters in the string specified by *lpszItem*. Use the **FindString** member function to find a string without selecting the item.

Return Value

The index of the selected item if the search was successful. If the search was unsuccessful, the return value is **LB_ERR** and the current selection is not changed.

See Also

CListBox::FindString, **LB_SELECTSTRING**

CListBox::SelItemRange

int SelItemRange(BOOL *bSelect*, **int** *nFirstItem*, **int** *nLastItem* **);**

bSelect Specifies how to set the selection. If *bSelect* is **TRUE**, the string is selected and highlighted; if **FALSE**, the highlight is removed and the string is no longer selected.

nFirstItem Specifies the zero-based index of the first item to set.

nLastItem Specifies the zero-based index of the last item to set.

Remarks

Selects one or more consecutive items in a multiple-selection list box. Use this member function only with multiple-selection list boxes.

Return Value

LB_ERR if an error occurs.

See Also

LB_SELITEMRANGE, **CListBox::GetSelItems**

CListBox::SetCaretIndex

Windows 3.1 Only

int SetCaretIndex(int *nIndex*, **BOOL** *bScroll* = **TRUE);** ♦

nIndex Specifies the zero-based index of the item to receive the focus rectangle in the list box.

bScroll If this value is 0, the item is scrolled until it is fully visible. If this value is not 0, the item is scrolled until it is at least partially visible.

Remarks

An application calls the **SetCaretIndex** member function to set the focus rectangle to the item at the specified index in a multiple-selection list box. If the item is not visible, it is scrolled into view.

Return Value

LB_ERR if an error occurs.

See Also

CListBox::GetCaretIndex, LB_SETCARETINDEX

CListBox::SetColumnWidth

void SetColumnWidth(int *cxWidth* **);**

cxWidth Specifies the width in pixels of all columns.

Remarks

Sets the width in pixels of all columns in a multicolumn list box (created with the **LBS_MULTICOLUMN** style).

See Also

LB_SETCOLUMNWIDTH

CListBox::SetCurSel

int SetCurSel(int *nSelect* **);**

nSelect Specifies the zero-based index of the string to be selected. If *nSelect* is –1, the list box is set to have no selection.

Remarks

Selects a string and scrolls it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string. Use this member function only with single-selection list boxes. It cannot be used to set or remove a selection in a multiple-selection list box.

Return Value **LB_ERR** if an error occurs.

See Also **LB_SETCURSEL**, **CListBox::GetCurSel**

CListBox::SetHorizontalExtent

void SetHorizontalExtent(int *cxExtent* **);**

cxExtent Specifies the number of pixels by which the list box can be scrolled horizontally.

Remarks Sets the width, in pixels, by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will horizontally scroll items in the list box. If the list box is as large or larger than this value, the horizontal scroll bar is hidden. To respond to a call to **SetHorizontalExtent**, the list box must have been defined with the **WS_HSCROLL** style. This member function is not useful for multicolumn listboxes. For multicolumn list boxes, call the **SetColumnWidth** member function.

See Also **CListBox::GetHorizontalExtent**, **LB_SETHORIZONTALEXTENT**

CListBox::SetItemData

int SetItemData(int *nIndex*, **DWORD** *dwItemData* **);**

nIndex Specifies the zero-based index of the item.

dwItemData Specifies the value to be associated with the item.

Remarks Sets a 32-bit value associated with the specified item in a list box.

Return Value **LB_ERR** if an error occurs.

See Also **CListBox::SetItemDataPtr**, **CListBox::GetItemData**, **LB_SETITEMDATA**

CListBox::SetItemDataPtr

int SetItemDataPtr(int *nIndex*, **void*** *pData* **);**

nIndex Specifies the zero-based index of the item.

pData Specifies the pointer to be associated with the item.

Remarks Sets the 32-bit value associated with the specified item in a combo box to be the specified pointer (**void***).

Return Value **LB_ERR** if an error occurs.

See Also **CListBox::SetItemData**, **CListBox::GetItemData**, **CListBox::GetItemDataPtr**, **LB_SETITEMDATA**

CListBox::SetItemHeight

Windows 3.1 Only **int SetItemHeight(int** *nIndex*, **UINT** *cyItemHeight* **);** ♦

nIndex Specifies the zero-based index of the item in the list box. This parameter is used only if the list box has the **LBS_OWNERDRAWVARIABLE** style; otherwise, it should be set to 0.

cyItemHeight Specifies the height, in pixels, of the item.

Remarks An application calls the **SetItemHeight** member function to set the height of items in a list box. If the list box has the **LBS_OWNERDRAWVARIABLE** style, this function sets the height of the item specified by *nIndex*. Otherwise, this function sets the height of all items in the list box.

Return Value **LB_ERR** if the index or height is invalid.

See Also **CListBox::GetItemHeight**, **LB_SETITEMHEIGHT**

CListBox::SetSel

int SetSel(int *nIndex*, **BOOL** *bSelect* **= TRUE);**

nIndex Contains the zero-based index of the string to be set. If –1, the selection is added to or removed from all strings, depending on the value of *bSelect*.

bSelect Specifies how to set the selection. If *bSelect* is **TRUE**, the string is selected and highlighted; if **FALSE**, the highlight is removed and the string is no longer selected. The specified string is selected and highlighted by default.

Remarks Selects a string in a multiple-selection list box. Use this message only with multiple-selection list boxes.

Return Value **LB_ERR** if an error occurs.

See Also **CListBox::GetSel**, **LB_SETSEL**

CListBox::SetTabStops

void SetTabStops( );

BOOL SetTabStops(const int& *cxEachStop* **);**

BOOL SetTabStops(int *nTabStops*, **LPINT** *rgTabStops* **);**

cxEachStop Tab stops are set at every *cxEachStop* dialog units. See *rgTabStops* for a description of a dialog unit.

nTabStops Specifies the number of tab stops to have in the list box.

rgTabStops Points to the first member of an array of integers containing the tab-stop positions in dialog units. A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to one-fourth of the current dialog base width unit, and 1 vertical dialog unit is equal to one-eighth of the current dialog base height unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** Windows function returns the current dialog base units in pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.

Remarks Sets the tab-stop positions in a list box.

To set tab stops to the default size of 2 dialog units, call the parameterless version of this member function. To set tab stops to a size other than 2, call the version with the *cxEachStop* argument.

To set tab stops to an array of sizes, use the version with the *rgTabStops* and *nTabStops* arguments. A tab stop will be set for each value in *rgTabStops*, up to the number specified by *nTabStops*. To respond to a call to the **SetTabStops** member function, the list box must have been created with the **LBS_USETABSTOPS** style.

Return Value Nonzero if all the tabs were set; otherwise 0.

See Also **LB_SETTABSTOPS**, **::GetDialogBaseUnits**

CListBox::SetTopIndex

int SetTopIndex(int *nIndex* **);**

nIndex Specifies the zero-based index of the list-box item.

Remarks Ensures that a particular list-box item is visible. The system scrolls the list box until either the list-box item appears at the top of the list box or the maximum scroll range has been reached.

Return Value **LB_ERR** if an error occurs.

See Also **CListBox::GetTopIndex**, **LB_SETTOPINDEX**

class CMapPtrToPtr : public CObject

CObject
└ CMapPtrToPtr

The **CMapPtrToPtr** class supports maps of void pointers keyed by void pointers. The member functions of **CMapPtrToPtr** are similar to the member functions of class **CMapStringToOb**. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute a pointer to **void**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapPtrToPtr::Lookup( void* <key>, void*& <rValue> ) const;
```

CMapPtrToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements (pointer values), you must set the depth of the dump context to 1 or greater. Pointer-to-pointer maps may not be serialized. When a **CMapPtrToPtr** object is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

#include <afxcoll.h>

See Also **CMapStringToOb**

Construction/Destruction—Public Members

CMapPtrToPtr	Constructs a collection that maps void pointers to void pointers.

Operations—Public Members

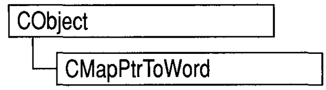
Lookup	Looks up a void pointer based on the void pointer key. The pointer value, not the entity it points to, is used for the key comparison.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapPtrToWord : public CObject



The **CMapPtrToWord** class supports maps of 16-bit words keyed by void pointers. The member functions of **CMapPtrToWord** are similar to the member functions of class **CMapStringToOb**. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute **WORD**. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute a pointer to **void**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapPtrToWord::Lookup( const void* <key>, WORD& <rValue> ) const;
```

CMapWordToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements, you must set the depth of the dump context to 1 or greater. Pointer-to-word maps may not be serialized. When a **CMapPtrToWord** object is deleted, or when its elements are removed, the pointers and the words are removed. The entities referenced by the key pointers are not removed.

#include <afxcoll.h>

See Also **CMapStringToOb**

Construction/Destruction—Public Members

CMapPtrToWord	Constructs a collection that maps void pointers to 16-bit words.

Operations—Public Members

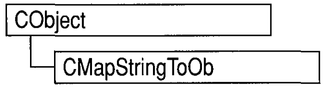
Lookup	Returns a **WORD** using a void pointer as a key. The pointer value, not the entity it points to, is used for the key comparison.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapStringToOb : public CObject



CMapStringToOb is a dictionary collection class that maps unique **CString** objects to **CObject** pointers. Once you have inserted a **CString-CObject*** pair (element) into the map, you can efficiently retrieve or delete the pair using a string or a **CString** value as a key. You can also iterate over all the elements in the map.

A variable of type **POSITION** is used for alternate entry access in all map variations. You can use a **POSITION** to "remember" an entry and to iterate through the map. You might think that this iteration is sequential by key value; it is not. The sequence of retrieved elements is indeterminate.

CMapStringToOb incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. Each element is serialized in turn if a map is stored to an archive, either with the overloaded insertion (<<) operator or with the **Serialize** member function. If you need a diagnostic dump of the individual elements in the map (the **CString** value and the **CObject** contents), you must set the depth of the dump context to 1 or greater.

When a **CMapStringToOb** object is deleted, or when its elements are removed, the **CString** objects and the **CObject** pointers are removed. The objects referenced by the **CObject** pointers are not destroyed.

Map class derivation is similar to list derivation. See the Chapter 13 of the *Class Library User's Guide* for a description of the derivation of a special-purpose list class.

#include <afxcoll.h>

See Also **CMapPtrToPtr**, **CMapPtrToWord**, **CMapStringToPtr**, **CMapStringToString**, **CMapWordToOb**, **CMapWordToPtr**

Construction/Destruction—Public Members

CMapStringToOb	Constructs a collection that maps **CString** values to **CObject** pointers.

Operations—Public Members

Lookup	Returns a **CObject** pointer based on a **CString** value.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

Member Functions

CMapStringToOb::CMapStringToOb

CMapStringToOb(int *nBlockSize* **= 10);**

nBlockSize Specifies the memory-allocation granularity for extending the map.

Remarks

Constructs an empty **CString**-to-**CObject*** map. As the map grows, memory is allocated in units of *nBlockSize* entries.

Example

See **CObList::CObList** for a listing of the `CAge` class used in all collection examples.

```
CMapStringToOb map(20);  // Map on the stack with blocksize of 20

CMapStringToOb* pm = new CMapStringToOb;  // Map on the heap
                                           // with default blocksize
```

CMapStringToOb::GetCount

int GetCount( ) const;

Return Value

The number of elements in this map.

See Also

CMapStringToOb::IsEmpty

Example

```
CMapStringToOb map;

map.SetAt( "Bart", new CAge( 13 ) );
map.SetAt( "Homer", new CAge( 36 ) );
ASSERT( map.GetCount() == 2 );
```

CMapStringToOb::GetNextAssoc

void GetNextAssoc(POSITION& *rNextPosition*, **CString&** *rKey*, **CObject*&** *rValue* **) const;**

rNextPosition Specifies a reference to a **POSITION** value returned by a previous **GetNextAssoc** or **GetStartPosition** call.

rKey Specifies the returned key of the retrieved element (a string).

rValue Specifies the returned value of the retrieved element (a **CObject** pointer).

Remarks

Retrieves the map element at *rNextPosition*, then updates *rNextPosition* to refer to the next element in the map. This function is most useful for iterating through all the elements in the map. Note that the position sequence is not necessarily the same as the key value sequence. If the retrieved element is the last in the map, then the new value of *rNextPosition* is set to **NULL**.

See Also

CMapStringToOb::GetStartPosition

Example

```
    CMapStringToOb map;
    POSITION pos;
    CString key;
    CAge* pa;

    map.SetAt( "Bart", new CAge( 13 ) );
    map.SetAt( "Lisa", new CAge( 11 ) );
    map.SetAt( "Homer", new CAge( 36 ) );
    map.SetAt( "Marge", new CAge( 35 ) );
    // Iterate through the entire map, dumping both name and age.
    for( pos = map.GetStartPosition(); pos != NULL; )
    {
    map.GetNextAssoc( pos, key, pa );
#ifdef _DEBUG
        afxDump << key << " : " << pa << "\n";
#endif
    }
```

The results from this program are as follows:

```
Lisa : a CAge at $4724 11
Marge : a CAge at $47A8 35
Homer : a CAge at $4766 36
Bart : a CAge at $45D4 13
```

CMapStringToOb::GetStartPosition

POSITION GetStartPosition( ) const;

Remarks Starts a map iteration by returning a **POSITION** value that can be passed to a **GetNextAssoc** call. The iteration sequence is not predictable; therefore, the "first element in the map" has no special significance.

Example See the example for the member function **GetNextAssoc**.

CMapStringToOb::IsEmpty

BOOL IsEmpty( ) const;

Return Value **TRUE** if this map contains no elements; otherwise **FALSE**.

See Also **CMapStringToOb::GetCount**

Example See the example for **RemoveAll**.

CMapStringToOb::Lookup

BOOL Lookup(const char* *key*, **CObject*&** *rValue* **) const;**

key Specifies the string key that identifies the element to be looked up.

rValue Specifies the returned value from the looked-up element.

Remarks **Lookup** uses a hashing algorithm to quickly find the map element with a key that matches exactly (**CString** value).

Return Value **TRUE** if the element was found; otherwise **FALSE**.

See Also **CMapStringToOb::operator [ ]**

Example

```
CMapStringToOb map;
CAge* pa;

map.SetAt( "Bart", new CAge( 13 ) );
map.SetAt( "Lisa", new CAge( 11 ) );
map.SetAt( "Homer", new CAge( 36 ) );
map.SetAt( "Marge", new CAge( 35 ) );
ASSERT( map.Lookup( "Lisa", pa ) ); // Is "Lisa" in the map?
ASSERT( *pa ==  CAge( 11 ) ); // Is she 11?
```

CMapStringToOb::RemoveAll

void RemoveAll( );

Remarks Removes all the elements from this map and destroys the **CString** key objects. The **CObject** objects referenced by each key are not destroyed. The **RemoveAll** function can cause memory leaks if you do not ensure that the referenced **CObject** objects are destroyed. The function works correctly if the map is already empty.

See Also **CMapStringToOb::RemoveKey**

Example

```
{
    CMapStringToOb map;

    CAge age1( 13 ); // Two objects on the stack
    CAge age2( 36 );
    map.SetAt( "Bart", &age1 );
    map.SetAt( "Homer", &age2 );
    ASSERT( map.GetCount() == 2 );
    map.RemoveAll(); // CObject pointers removed; objects not removed.
    ASSERT( map.GetCount() == 0 );
    ASSERT( map.IsEmpty() );
} // The two CAge objects are deleted when they go out of scope.
```

CMapStringToOb::RemoveKey

BOOL RemoveKey(const char* *key* **);**

key Specifies the string used for map lookup.

Remarks

Looks up the map entry corresponding to the supplied key; then, if the key is found, removes the entry. This can cause memory leaks if the **CObject** object is not deleted elsewhere.

Return Value

TRUE if the entry was found and successfully removed; otherwise **FALSE**.

See Also

CMapStringToOb::RemoveAll

Example

```
    CMapStringToOb map;

    map.SetAt( "Bart", new CAge( 13 ) );
    map.SetAt( "Lisa", new CAge( 11 ) );
    map.SetAt( "Homer", new CAge( 36 ) );
    map.SetAt( "Marge", new CAge( 35 ) );
    map.RemoveKey( "Lisa" ); // Memory leak: CAge object not
                             // deleted.
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "RemoveKey example: " << &map << "\n";
#endif
```

The results from this program are as follows:

```
RemoveKey example: A CMapStringToOb with 3 elements
    [Marge] = a CAge at $49A0 35
    [Homer] = a CAge at $495E 36
    [Bart] = a CAge at $4634 13
```

CMapStringToOb::SetAt

void SetAt(const char* *key*, **CObject*** *newValue* **)**
throw(CMemoryException);

key Specifies the string that is the key of the new element.

newValue Specifies the **CObject** pointer that is the value of the new element.

Remarks The primary means to insert an element in a map. First, the key is looked up. If the key is found, then the corresponding value is changed; otherwise a new key-value element is created.

See Also **CMapStringToOb::Lookup**, **CMapStringToOb::operator [ ]**

Example

```
    CMapStringToOb map;
    CAge* pa;

    map.SetAt( "Bart", new CAge( 13 ) );
    map.SetAt( "Lisa", new CAge( 11 ) ); // Map contains 2
                                         // elements.
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "before Lisa's birthday: " << &map << "\n";
#endif
    if( map.Lookup( "Lisa", pa ) )
    { // CAge 12 pointer replaces CAge 11 pointer.
        map.SetAt( "Lisa", new CAge( 12 ) );
        delete pa;  // Must delete CAge 11 to avoid memory leak.
    }
#ifdef _DEBUG
    afxDump << "after Lisa's birthday: " << &map << "\n";
#endif
```

The results from this program are as follows:

```
before Lisa's birthday: A CMapStringToOb with 2 elements
    [Lisa] = a CAge at $493C 11
    [Bart] = a CAge at $4654 13
after Lisa's birthday: A CMapStringToOb with 2 elements
    [Lisa] = a CAge at $49C0 12
    [Bart] = a CAge at $4654 13
```

Operators

CMapStringToOb::operator [ ]

CObject*& operator [ ](const char* *key* **);**

Remarks

This operator is a convenient substitute for the **SetAt** member function. Thus it can be used only on the left side of an assignment statement (an l-value). If there is no map element with the specified key, then a new element is created. There is no "right side" (r-value) equivalent to this operator because there is a possibility that a key may not be found in the map. Use the **Lookup** member function for element retrieval.

See Also

CMapStringToOb::SetAt, **CMapStringToOb::Lookup**

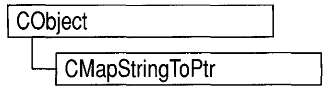
Example

```
    CMapStringToOb map;

    map["Bart"] = new CAge( 13 );
    map["Lisa"] = new CAge( 11 );
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "Operator [] example: " << &map << "\n";
#endif
```

The results from this program are as follows:

```
Operator [] example: A CMapStringToOb with 2 elements
    [Lisa] = a CAge at $4A02 11
    [Bart] = a CAge at $497E 13
```

class CMapStringToPtr : public CObject



The **CMapStringToPtr** class supports maps of void pointers keyed by **CString** objects. The member functions of **CMapStringToPtr** are similar to the member functions of class **CMapStringToOb**. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapStringToPtr::Lookup( const char* <key>, void*& <rValue> )
                              const;
```

CMapStringToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements, you must set the depth of the dump context to 1 or greater. String-to-pointer maps may not be serialized. When a **CMapStringToPtr** object is deleted, or when its elements are removed, the **CString** key objects and the words are removed.

#include <afxcoll.h>

See Also **CMapStringToOb**

Construction/Destruction—Public Members

CMapStringToPtr	Constructs a collection that maps **CString** objects to void pointers.

Operations—Public Members

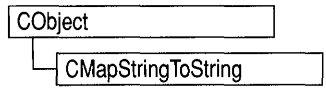
Lookup	Returns a void pointer based on a **CString** value.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapStringToString : public CObject



The **CMapStringToString** class supports maps of **CString** objects keyed by **CString** objects. The member functions of **CMapStringToString** are similar to the member functions of class **CMapStringToOb**. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value or "output" function parameter, substitute a pointer to **char**. Wherever you see a **CObject** pointer as an "input" function parameter, substitute a pointer to **char**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapStringToString::Lookup( const char* <key>,
                                 CString& <rValue> ) const;
```

CMapStringToString incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. Each element is serialized in turn if a map is stored to an archive, either with the overloaded insertion (<<) operator or with the **Serialize** member function. If you need a dump of individual **CString-CString** elements, you must set the depth of the dump context to 1 or greater. When a **CMapStringToString** object is deleted, or when its elements are removed, the **CString** objects are removed as appropriate.

#include <afxcoll.h>

See Also **CMapStringToOb**

Construction/Destruction—Public Members

CMapStringToString	Constructs a collection that maps **CString** objects to **CString** objects.

Operations—Public Members

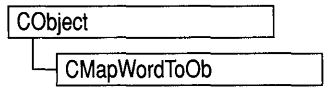
Lookup	Returns a **CString** using a **CString** value as a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapWordToOb : public CObject



The **CMapWordToOb** class supports maps of **CObject** pointers keyed by 16-bit words. The member functions of **CMapWordToOb** are similar to the member functions of class **CMapStringToOb**. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute **WORD**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapWordToOb::Lookup( WORD <key>, CObject*& <rValue> ) const;
```

CMapWordToOb incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. Each element is serialized in turn if a map is stored to an archive, either with the overloaded insertion (<<) operator or with the **Serialize** member function. If you need a dump of individual **WORD-CObject** elements, you must set the depth of the dump context to 1 or greater. When a **CMapWordToOb** object is deleted, or when its elements are removed, the **CObject** objects are deleted as appropriate.

#include <afxcoll.h>

See Also **CMapStringToOb**

Construction/Destruction—Public Members

CMapWordToOb	Constructs a collection that maps words to **CObject** pointers.

Operations—Public Members

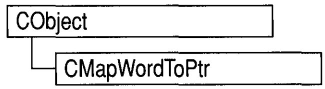
Lookup	Returns a **CObject** pointer using a word value as a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapWordToPtr : public CObject



The **CMapWordToPtr** class supports maps of void pointers keyed by 16-bit words. The member functions of **CMapWordToPtr** are similar to the member functions of class **CMapStringToOb**. Because of this similarity, you can use the **CMapStringToOb** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**. Wherever you see a **CString** or a **const** pointer to **char** as a function parameter or return value, substitute **WORD**.

```
BOOL CMapStringToOb::Lookup( const char* <key>,
                             CObject*& <rValue> ) const;
```

for example, translates to

```
BOOL CMapWordToPtr::Lookup( WORD <key>, void*& <rValue> ) const;
```

CMapWordToPtr incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual map elements, you must set the depth of the dump context to 1 or greater. Word-to-pointer maps may not be serialized. When a **CMapWordToPtr** object is deleted, or when its elements are removed, the words and the pointers are removed. The entities referenced by the pointers are not removed.

#include <afxcoll.h>

See Also **CMapStringToOb**

Construction/Destruction — Public Members

CMapWordToPtr	Constructs a collection that maps words to void pointers.

Operations — Public Members

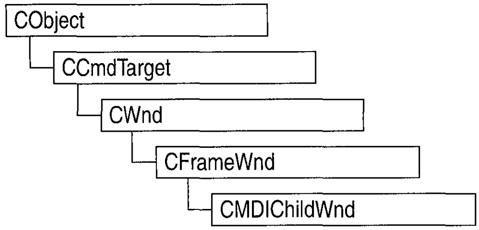
Lookup	Returns a void pointer using a word value as a key.
SetAt	Inserts an element into the map; replaces an existing element if a matching key is found.
operator [ ]	Inserts an element into the map—operator substitution for **SetAt**.

RemoveKey	Removes an element specified by a key.
RemoveAll	Removes all the elements from this map.
GetStartPosition	Returns the position of the first element.
GetNextAssoc	Gets the next element for iterating.

Status—Public Members

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMDIChildWnd : public CFrameWnd



The **CMDIChildWnd** class provides the functionality of a Windows multiple document interface (MDI) child window, along with members for managing the window. An MDI child window looks much like a typical frame window, except that the MDI child window appears inside an MDI frame window rather than on the desktop. An MDI child window does not have a menu bar of its own, but instead shares the menu of the MDI frame window. The framework automatically changes the MDI frame menu to represent the currently active MDI child window.

To create a useful MDI child window for your application, derive a class from **CMDIChildWnd**. Add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window. There are three ways to construct an MDI child window:

- Directly construct it using **Create**.
- Directly construct it using **LoadFrame**.
- Indirectly construct it through a document template.

Before you call **Create** or **LoadFrame**, you must construct the frame-window object on the heap using the C++ **new** operator. Before calling **Create** you may also register a window class with the **AfxRegisterWndClass** global function to set the icon and class styles for the frame. Use the **Create** member function to pass the frame's creation parameters as immediate arguments.

LoadFrame requires fewer arguments than **Create**, and instead retrieves most of its default values from resources, including the frame's caption, icon, accelerator table, and menu. To be accessible by **LoadFrame**, all these resources must have the same resource ID (for example, **IDR_MAINFRAME**).

When a **CMDIChildWnd** object contains views and documents, they are created indirectly by the framework instead of directly by the programmer. The **CDocTemplate** object orchestrates the creation of the frame, the creation of the containing views, and the connection of the views to the appropriate document. The parameters of the **CDocTemplate** constructor specify the **CRuntimeClass** of the three classes involved (document, frame, and view). A **CRuntimeClass** object is used by the framework to dynamically create new frames when specified by the user (for example, by using the File New command or the MDI Window New command).

A frame-window class derived from **CMDIChildWnd** must be declared with **DECLARE_DYNCREATE** in order for the above **RUNTIME_CLASS** mechanism to work correctly.

The **CMDIChildWnd** class inherits much of its default implementation from **CFrameWnd**. For a detailed list of these features, please refer to the **CFrameWnd** class description. The **CMDIChildWnd** class has the following additional features:

- In conjunction with the **CMultiDocTemplate** class, multiple **CMDIChildWnd** objects from the same document template share the same menu, saving Windows system resources.
- The currently active MDI child window menu entirely replaces the MDI frame window's menu, and the caption of the currently active MDI child window is added to the MDI frame window's caption. For further examples of MDI child window functions that are implemented in conjunction with an MDI frame window, see the **CMDIFrameWnd** class description.

Do not use the C++ **delete** operator to destroy a frame window. Use **CWnd::DestroyWindow** instead. The **CFrameWnd** implementation of **PostNcDestroy** will delete the C++ object when the window is destroyed. When the user closes the frame window, the default **OnClose** handler will call **DestroyWindow**.

#include <afxwin.h>

See Also **CWnd**, **CFrameWnd**, **CMDIFrameWnd**

Construction/Destruction — Public Members

CMDIChildWnd	Constructs a **CMDIChildWnd** object.

Initialization — Public Members

Create	Creates the Windows MDI child window associated with the **CMDIChildWnd** object.

Operations — Public Members

MDIDestroy	Destroys this MDI child window.
MDIActivate	Activates this MDI child window.
MDIMaximize	Maximizes this MDI child window.
MDIRestore	Restores this MDI child window from maximized or minimized size.
GetMDIFrame	Returns the parent MDI frame of the MDI client window.

Member Functions

CMDIChildWnd::CMDIChildWnd

CMDIChildWnd( );

Remarks Call to construct a **CMDIChildWnd** object. Call **Create** to create the visible window.

See Also **CMDIChildWnd::Create**

CMDIChildWnd::Create

BOOL Create(LPCSTR *lpszClassName*, **LPCSTR** *lpszWindowName*, **DWORD** *dwStyle* = **WS_CHILD | WS_VISIBLE | WS_OVERLAPPEDWINDOW, const RECT&** *rect* = **rectDefault, CMDIFrameWnd*** *pParentWnd* = **NULL, CCreateContext*** *pContext* = **NULL);**

lpszClassName Points to a null-terminated character string that names the Windows class (a **WNDCLASS** structure). The class name can be any name registered with the **AfxRegisterWndClass** global function. Should be **NULL** for a standard **CMDIChildWnd**.

lpszWindowName Points to a null-terminated character string that represents the window name. Used as text for the title bar.

dwStyle Specifies the window style attributes. The **WS_CHILD** style is required.

See the **Create** member function in the **CWnd** class for a full list of window styles.

rect Contains the size and position of the window. The **rectDefault** value allows Windows to specify the size and position of the new **CMDIChildWnd**.

pParentWnd Specifies the window's parent. If **NULL**, the main application window is used.

pContext Specifies a **CCreateContext** structure. This parameter can be **NULL**.

Remarks

Call this member function to create a Windows MDI child window and attach it to the **CMDIChildWnd** object. The currently active MDI child frame window can determine the caption of the parent frame window. This feature is disabled by turning off the **FWS_ADDTOTITLE** style bit of the child frame window.

The framework calls this member function in response to a user command to create a child window, and the framework uses the *pContext* parameter to properly connect the child window to the application. When you call **Create**, *pContext* may be **NULL**.

Return Value

Nonzero if successful; otherwise 0.

See Also

CMDIChildWnd::CMDIChildWnd, **CWnd::PreCreateWindow**

CMDIChildWnd::GetMDIFrame

CMDIFrameWnd* GetMDIFrame( );

Remarks

Call this function to return the MDI parent frame. The frame returned is two parents removed from the **CMDIChildWnd** and is the parent of the window of type **MDICLIENT** that manages the **CMDIChildWnd** object. Call the **GetParent** member function to return the **CMDIChildWnd** object's immediate **MDICLIENT** parent as a temporary **CWnd** pointer.

See Also

CWnd::GetParent

CMDIChildWnd::MDIActivate

void MDIActivate( );

Remarks

Call this member function to activate an MDI child window independently of the MDI frame window. When the frame becomes active, the child window that was last activated will be activated as well.

See Also

CMDIFrameWnd::MDIGetActive, **CWnd::OnNcActivate**, **CMDIFrameWnd::MDINext**, **WM_MDIACTIVATE**

CMDIChildWnd::MDIDestroy

void MDIDestroy( );

Remarks

Call this member function to destroy an MDI child window. The member function removes the title of the child window from the frame window and deactivates the child window.

See Also

WM_MDIDESTROY, **CMDIChildWnd::Create**

CMDIChildWnd::MDIMaximize

void MDIMaximize( );

Remarks

Call this member function to maximize an MDI child window. When a child window is maximized, Windows resizes it to make its client area fill the client area of the frame window. Windows places the child window's Control menu in the frame's menu bar so that the user can restore or close the child window and adds the title of the child window to the frame-window title.

See Also

WM_MDIMAXIMIZE, **CMDIChildWnd::MDIRestore**

CMDIChildWnd::MDIRestore

void MDIRestore( );

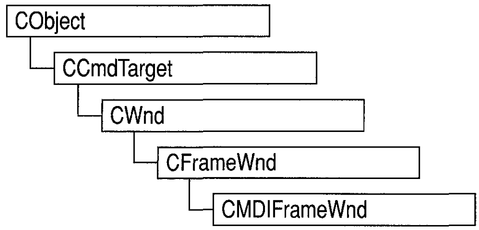
Remarks

Call this member function to restore an MDI child window from maximized or minimized size.

See Also

CMDIChildWnd::MDIMaximize, **WM_MDIRESTORE**

class CMDIFrameWnd : public CFrameWnd



The **CMDIFrameWnd** class provides the functionality of a Windows multiple document interface (MDI) frame window, along with members for managing the window. To create a useful MDI frame window for your application, derive a class from **CMDIFrameWnd**. Add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

You can construct an MDI frame window by calling the **Create** or **LoadFrame** member functions of **CFrameWnd**.

Before you call **Create** or **LoadFrame**, you must construct the frame window object on the heap using the C++ **new** operator. Before calling **Create** you may also register a window class with the **AfxRegisterWndClass** global function to set the icon and class styles for the frame.

Use the **Create** member function to pass the frame's creation parameters as immediate arguments.

LoadFrame requires fewer arguments than **Create**, and instead retrieves most of its default values from resources, including the frame's caption, icon, accelerator table, and menu. To be accessed by **LoadFrame**, all these resources must have the same resource ID (for example, **IDR_MAINFRAME**).

Though **MDIFrameWnd** is derived from **CFrameWnd**, a frame window class derived from **CMDIFrameWnd** need not be declared with **DECLARE_DYNCREATE**.

The **CMDIFrameWnd** class inherits much of its default implementation from **CFrameWnd**. For a detailed list of these features, refer to the **CFrameWnd** class description. The **CMDIFrameWnd** class has the following additional features:

- An MDI frame window manages the **MDICLIENT** window, repositioning it in conjunction with control bars. The MDI client window is the direct parent of MDI child frame windows. The **WS_HSCROLL** and **WS_VSCROLL** window styles specified on a **CMDIFrameWnd** apply to the MDI client window rather than the main frame window so the user can scroll the MDI client area (as in the Windows Program Manager, for example).

- An MDI frame window owns a default menu that is used as the menu bar when there is no active MDI child window. When there is an active MDI child, the MDI frame window's menu bar is automatically replaced by the MDI child window menu.
- An MDI frame window works in conjunction with the current MDI child window, if there is one. For instance, command messages are delegated to the currently active MDI child before the MDI frame window.
- An MDI frame window has default handlers for the following standard Window menu commands:

  ID_WINDOW_TILE_VERT

  ID_WINDOW_TILE_HORZ

  ID_WINDOW_CASCADE

  ID_WINDOW_ARRANGE

  An MDI frame window also has an implementation of **ID_WINDOW_NEW**, which creates a new frame and view on the current document. An application can override these default command implementations to customize MDI window handling.

Do not use the C++ **delete** operator to destroy a frame window. Use **CWnd::DestroyWindow** instead. The **CFrameWnd** implementation of **PostNcDestroy** will delete the C++ object when the window is destroyed. When the user closes the frame window, the default **OnClose** handler will call **DestroyWindow**.

#include <afxwin.h>

See Also **CWnd**, **CFrameWnd**, **CMDIChildWnd**

Construction/Destruction — Public Members

CMDIFrameWnd	Constructs a **CMDIFrameWnd**.

Operations — Public Members

MDIActivate	Activates a different MDI child window.
MDIGetActive	Retrieves the currently active MDI child window, along with a flag indicating whether or not the child is maximized.
MDIIconArrange	Arranges all minimized document child windows.
MDIMaximize	Maximizes an MDI child window.

MDINext	Activates the child window immediately behind the currently active child window and places the currently active child window behind all other child windows.
MDIRestore	Restores an MDI child window from maximized or minimized size.
MDISetMenu	Replaces the menu of an MDI frame window, the Window pop-up menu, or both.
MDITile	Arranges all child windows in a tiled format.
MDICascade	Arranges all child windows in a cascaded format.

Overridables—Public Members

CreateClient	Creates a Windows **MDICLIENT** window for this **CMDIFrameWnd**. Called by the **OnCreate** member function of **CWnd**.
GetWindowMenuPopup	Returns the Window pop-up menu.

Member Functions

CMDIFrameWnd::CMDIFrameWnd

CMDIFrameWnd( );

Remarks Call this member function to construct a **CMDIFrameWnd** object. Call the **Create** or **LoadFrame** member functions to create the visible MDI frame window.

See Also **CFrameWnd::Create**, **CFrameWnd::LoadFrame**

CMDIFrameWnd::CreateClient

virtual BOOL CreateClient(LPCREATESTRUCT *lpCreateStruct*, **CMenu*** *pWindowMenu* **);**

lpCreateStruct A long pointer to a **CREATESTRUCT** structure.

pWindowMenu A pointer to the Window pop-up menu.

Remarks Creates the MDI client window that manages the **CMDIChildWnd** objects.

This member function should be called if you override the **OnCreate** member function directly.

Return Value Nonzero if successful; otherwise 0.

See Also **CMDIFrameWnd::CMDIFrameWnd**

CMDIFrameWnd::GetWindowMenuPopup

virtual HMENU GetWindowMenuPopup(HMENU *hMenuBar* **);**

hMenuBar The current menu bar.

Remarks Call this member function to obtain a handle to the current pop-up menu named "Window" (the pop-up menu with menu items for MDI window management).

The default implementation looks for a pop-up menu containing standard Window menu commands such as **ID_WINDOW_NEW** and **ID_WINDOW_TILE_HORZ**.

Override this member function if you have a Window menu that doesn't use the standard menu command IDs.

Return Value The Window pop-up menu if one exists; otherwise **NULL**.

See Also **CMDIFrameWnd::MDIGetActive**

CMDIFrameWnd::MDIActivate

void MDIActivate(CWnd* *pWndActivate* **);**

pWndActivate Points to the MDI child window to be activated.

Remarks Call this member function to activate a different MDI child window. This member function sends the **WM_MDIACTIVATE** message to both the child window being activated and the child window being deactivated. This is the same message that is sent if the user changes the focus to an MDI child window by using the mouse or keyboard.

Note An MDI child window is activated independently of the MDI frame window. When the frame becomes active, the child window that was last activated is sent a **WM_NCACTIVATE** message to draw an active window frame and caption bar, but it does not receive another **WM_MDIACTIVATE** message.

See Also **CMDIFrameWnd::MDIGetActive**, **CMDIFrameWnd::MDINext**, **WM_ACTIVATE**, **WM_NCACTIVATE**

CMDIFrameWnd::MDICascade

void MDICascade( );

Windows 3.1 Only **void MDICascade(int** *nType* **);** ♦

nType Specifies a cascade flag. Only the following flag may be specified: **MDITILE_SKIPDISABLED**, which prevents disabled MDI child windows from being cascaded.

Remarks Call this member function to arrange all the MDI child windows in a cascade format.

The first version of **MDICascade**, with no parameters, cascades all MDI child windows, including disabled ones. The second version optionally does not cascade disabled MDI child windows if you specify **MDITILE_SKIPDISABLED** for the *nType* parameter.

See Also **CMDIFrameWnd::MDIIconArrange**, **CMDIFrameWnd::MDITile**, **WM_MDICASCADE**

CMDIFrameWnd::MDIGetActive

CMDIChildWnd* MDIGetActive(BOOL* *pbMaximized* = **NULL) const;**

pbMaximized A pointer to a **BOOL** return value. Set to **TRUE** on return if the window is maximized; otherwise **FALSE**.

Remarks Retrieves the current active MDI child window, along with a flag indicating whether the child window is maximized.

Return Value A pointer to the active MDI child window.

See Also **CMDIFrameWnd::MDIActivate, WM_MDIGETACTIVE**

CMDIFrameWnd::MDIIconArrange

void MDIIconArrange( );

Remarks Arranges all minimized document child windows. It does not affect child windows that are not minimized.

See Also **CMDIFrameWnd::MDICascade**, **CMDIFrameWnd::MDITile**, **WM_MDIICONARRANGE**

CMDIFrameWnd::MDIMaximize

void MDIMaximize(CWnd* *pWnd* **);**

pWnd Points to the window to maximize.

Remarks Call this member function to maximize the specified MDI child window. When a child window is maximized, Windows resizes it to make its client area fill the client window. Windows places the child window's Control menu in the frame's menu bar so the user can restore or close the child window. It also adds the title of the child window to the frame-window title. If another MDI child window is activated when the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child window.

See Also **WM_MDIMAXIMIZE**, **CMDIFrameWnd::MDIRestore**

CMDIFrameWnd::MDINext

void MDINext( );

Remarks Activates the child window immediately behind the currently active child window and places the currently active child window behind all other child windows. If the currently active MDI child window is maximized, the member function restores the currently active child and maximizes the newly activated child.

See Also **CMDIFrameWnd::MDIActivate**, **CMDIFrameWnd::MDIGetActive**, **WM_MDINEXT**

CMDIFrameWnd::MDIRestore

void MDIRestore(CWnd* *pWnd* **);**

pWnd Points to the window to restore.

Remarks Restores an MDI child window from maximized or minimized size.

See Also **CMDIFrameWnd::MDIMaximize**, **WM_MDIRESTORE**

CMDIFrameWnd::MDISetMenu

CMenu* MDISetMenu(CMenu* *pFrameMenu*, **CMenu*** *pWindowMenu* **);**

pFrameMenu Specifies the menu of the new frame-window menu. If **NULL**, the menu is not changed.

pWindowMenu Specifies the menu of the new Window pop-up menu. If **NULL**, the menu is not changed.

Remarks Call this member function to replace the menu of an MDI frame window, the Window pop-up menu, or both. After calling **MDISetMenu**, an application must call the **DrawMenuBar** member function of **CWnd** to update the menu bar. If this call replaces the Window pop-up menu, MDI child-window menu items are removed from the previous Window menu and added to the new Window pop-up menu. If an MDI child window is maximized and this call replaces the MDI frame-window menu, the Control menu and restore controls are removed from the previous frame-window menu and added to the new menu.

Do not call this member function if you use the framework to manage your MDI child windows.

Return Value A pointer to the frame-window menu replaced by this message. The pointer may be temporary and should not be stored for later use.

See Also **CWnd::DrawMenuBar**, **WM_MDISETMENU**

CMDIFrameWnd::MDITile

void MDITile( );

Windows 3.1 Only **void MDITile(int** *nType* **);** ♦

nType Specifies a tiling flag. This parameter can be one of the following flags, with the indicated meaning:

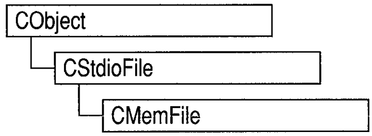
- **MDITILE_HORIZONTAL** Tiles MDI child windows so that one window appears above another.
- **MDITILE_SKIPDISABLED** Prevents disabled MDI child windows from being tiled.
- **MDITILE_VERTICAL** Tiles MDI child windows so that one window appears beside another.

Remarks Call this member function to arrange all child windows in a tiled format.

The first version of **MDITile**, without parameters, tiles the windows vertically under Windows version 3.1 and arbitrarily under Windows version 3.0. The second version tiles windows vertically or horizontally, depending on the value of the *nType* parameter.

See Also **CMDIFrameWnd::MDICascade**, **CMDIFrameWnd::MDIIconArrange**, **WM_MDITILE**

class CMemFile : public CFile



CMemFile is the **CFile**-derived class that supports in-memory files. These in-memory files behave like binary disk files except that bytes are stored in RAM. An in-memory file is a useful means of transferring raw bytes or serialized objects between independent processes. Contiguous memory is automatically allocated in specified increments, and it is deleted when the object is destroyed. You can access this memory through a pointer supplied by a member function.

The **Duplicate**, **LockRange**, and **UnlockRange** functions are not implemented for **CMemFile**. If you call these functions on a **CMemFile** object, you will get a **CNotSupportedException**. The data member **CFile::m_hFile** is not used and has no meaning.

If you derive a class from **CMemFile**, you must use the protected memory-allocation functions listed above, overriding them as necessary. If you need global memory access from the medium model in the Windows operating system, for example, derive a class with the four protected functions overridden. Your replacement functions should call the Windows **GlobalAlloc** family of functions.

#include <afx.h>

Construction/Destruction—Public Members

CMemFile	Constructs a memory file using internally allocated memory.
~CMemFile	Closes the memory file, freeing allocated memory.

Member Functions

CMemFile::CMemFile

CMemFile(UINT *nGrowBytes* **= 1024)**
throw (CFileException, CMemoryException);

nGrowBytes The memory-allocation increment in bytes.

Remarks Allocates memory and opens an empty memory file.

Example

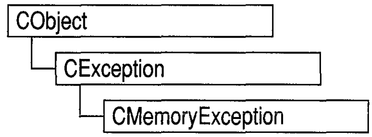
```
CMemFile f; // Ready to use - no Open necessary.
```

CMemFile::~CMemFile

virtual ~CMemFile( );

Remarks Frees all allocated memory associated with this memory file, effectively closing it.

class CMemoryException : public CException



A **CMemoryException** object represents an out-of-memory exception condition. No further qualification is necessary or possible. Memory exceptions are thrown automatically by **new**. If you write your own memory functions, using **malloc**, for example, then you are responsible for throwing memory exceptions.

#include <afx.h>

Construction/Destruction — Public Members

CMemoryException	Constructs a **CMemoryException** object.

Member Functions

CMemoryException::CMemoryException

CMemoryException( );

Remarks

Constructs a **CMemoryException** object. Do not use this constructor directly, but rather call the global function **AfxThrowMemoryException**. This global function can succeed in an out-of-memory situation because it constructs the exception object in previously allocated memory. For more information about exception processing, see Chapter 16, "Exceptions," in the *Class Library User's Guide*.

See Also

AfxThrowMemoryException

struct CMemoryState

CMemoryState provides a convenient way to detect memory leaks in your program. A "memory leak" occurs when memory for an object is allocated on the heap but not deallocated when it is no longer required. Such memory leaks can eventually lead to out-of-memory errors. To allocate and deallocate memory:

- Use the **malloc/free** family of functions from the run-time library
- Use the Windows API memory management functions, **LocalAlloc/LocalFree** and **GlobalAlloc/GlobalFree**
- Use the C++ **new** and **delete** operators

The **CMemoryState** diagnostics only help detect memory leaks caused when memory allocated using the **new** operator is not deallocated using **delete**. The other two groups of memory-management functions are for non-C++ programs, and mixing them with **new** and **delete** in the same program is not recommended. An additional macro, **DEBUG_NEW**, is provided to replace the **new** operator when you need file and line-number tracking of memory allocations. **DEBUG_NEW** is used whenever you would normally use the **new** operator.

As with other diagnostics, the **CMemoryState** diagnostics are only available in debug versions of your program. A debug version must have the **_DEBUG** constant defined.

If you suspect your program has a memory leak, you can use the **Checkpoint**, **Difference**, and **DumpStatistics** functions to find the difference between the memory state (objects allocated) at two different points in program execution. This can help you determine if a function is cleaning up all the objects it allocates.

If simply knowing where the imbalance in allocation and deallocation occurs does not provide enough information, you can use the **DumpAllObjectsSince** function to dump all objects allocated since the previous call to **Checkpoint**. This dump shows the order of allocation, the source file and line where the object was allocated (if you are using **DEBUG_NEW** for allocation), and the derivation of the object, its address, and its size. **DumpAllObjectsSince** also invokes each object's **Dump** function to provide information about its current state.

For more information about how to use **CMemoryState** and other diagnostics, see the *Class Library User's Guide*.

Note Declarations of objects of type **CMemoryState** and calls to member functions should be bracketed by **#if defined(_DEBUG)/#endif** directives so that memory diagnostics will be included only in debugging builds of your program.

Construction/Destruction—Public Members

CMemoryState	Constructs a class-like structure that controls memory checkpoints.
Checkpoint	Obtains a snapshot or "checkpoint" of the current memory state.

Operations—Public Members

Difference	Computes the difference between two objects of type **CMemoryState**.
DumpAllObjectsSince	Dumps a summary of all currently allocated objects since a previous checkpoint.
DumpStatistics	Prints memory allocation statistics for a **CMemoryState** object.

Member Functions

CMemoryState::Checkpoint

void Checkpoint( );

Remarks

Takes a snapshot summary of memory and stores it in this **CMemoryState** object. The **CMemoryState** member functions **Difference** and **DumpAllObjectsSince** use this snapshot data.

Example

See the example for the **CMemoryState** constructor.

CMemoryState::CMemoryState

CMemoryState( );

Remarks

Constructs an empty **CMemoryState** object that must be filled in by the **Checkpoint** or **Difference** member functions.

Example

```
// Includes all CMemoryState functions
CMemoryState msOld, msNew, msDif;
msOld.Checkpoint();
CAge* page1 = new CAge( 21 );
CAge* page2 = new CAge( 22 );
msOld.DumpAllObjectsSince();
msNew.Checkpoint();
msDif.Difference( msOld, msNew );
msDif.DumpStatistics();
```

The results from this program are as follows:

```
Dumping objects ->
{2} a CObject at $190A
{1} a CObject at $18EA
Object dump complete.
0 bytes in 0 Free Blocks
8 bytes in 2 Object Blocks
0 bytes in 0 Non-Object Blocks
Largest number used: 8 bytes
Total allocations: 8 bytes
```

CMemoryState::Difference

BOOL Difference(const CMemoryState& *oldState***,**
const CMemoryState& *newState* **);**

oldState The initial memory state as defined by a **CMemoryState** checkpoint.

newState The new memory state as defined by a **CMemoryState** checkpoint.

Remarks

Compares two **CMemoryState** objects, then stores the difference into this **CMemoryState** object. **Checkpoint** must have been called for each of the two memory-state parameters.

Example

See the example for the **CMemoryState** constructor.

CMemoryState::DumpAllObjectsSince

void DumpAllObjectsSince( ) const;

Remarks

Calls the **Dump** function for all objects of a type derived from class **CObject** that were allocated (and are still allocated) since the last **Checkpoint** call for this **CMemoryState** object.

Calling **DumpAllObjectsSince** with an uninitialized **CMemoryState** object will dump out all objects currently in memory.

Example

See the example for the **CMemoryState** constructor.

CMemoryState::DumpStatistics

void DumpStatistics( ) const;

Remarks

Prints a concise memory statistics report from a **CMemoryState** object that is filled by the **Difference** member function. The report, which is printed on the **afxDump** device, shows the following:

- Number of "object" blocks (blocks of memory allocated using **CObject::operator new**) still allocated on the heap.
- Number of non-object blocks still allocated on the heap.
- The maximum memory used by the program at any one time (in bytes).
- The total memory currently used by the program (in bytes).

A sample report looks as follows:

```
0 bytes in 0 Free Blocks
8 bytes in 2 Object Blocks
0 bytes in 0 Non-Object Blocks
Largest number used: 8 bytes
Total allocations: 8 bytes
```

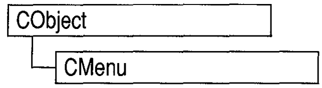
- The first line describes the number of blocks whose deallocation was delayed if **afxMemDF** was set to **delayFreeMemDF**. For a description of **afxMemDF**, see "Macros and Globals."
- The second line describes how many object blocks still remain allocated on the heap.

- The third line describes how many nonobject blocks (arrays or structures allocated with new) were allocated on the heap and not deallocated.
- The fourth line gives the maximum memory used by your program at any one time.
- The last line lists the total amount of memory used by your program.

Example

See the example for the **CMemoryState** constructor.

class CMenu : public CObject



The **CMenu** class is an encapsulation of the Windows **HMENU**. It provides member functions for creating, tracking, updating, and destroying a menu.

Create a **CMenu** object on the stack frame as a local, then call **CMenu**'s member functions to manipulate the new menu as needed. Next, call **CWnd::SetMenu** to set the menu to a window, followed immediately by a call to the **Detach** member function. The **CWnd::SetMenu** member function sets the window's menu to the new menu, causes the window to be redrawn to reflect the menu change, and also passes ownership of the menu to the window. The call to **Detach** detaches the **HMENU** from the **CMenu** object, so that when the local **CMenu** variable passes out of scope, the **CMenu** object destructor does not attempt to destroy a menu it no longer owns. The menu itself is automatically destroyed when the window is destroyed.

You can use the **LoadMenuIndirect** member function to create a menu from a template in memory, but a menu created from a resource by a call to **LoadMenu** is more easily maintained, and the menu resource itself can be created and modified by App Studio.

#include <afxwin.h>

See Also **CObject**

Data Members—Public Members

m_hMenu	Specifies the handle to the Windows menu attached to the **CMenu** object.

Construction/Destruction—Public Members

CMenu	Constructs a **CMenu** object.

Initialization—Public Members

Attach	Attaches a Windows menu handle to a **CMenu** object.
Detach	Detaches a Windows menu handle from a **CMenu** object and returns the handle.
FromHandle	Returns a pointer to a **CMenu** object given a Windows menu handle.
GetSafeHmenu	Returns the **m_hMenu** wrapped by this **CMenu** object.
DeleteTempMap	Deletes any temporary **CMenu** objects created by the **FromHandle** member function.

CreateMenu	Creates an empty menu and attaches it to a **CMenu** object.
CreatePopupMenu	Creates an empty pop-up menu and attaches it to a **CMenu** object.
LoadMenu	Loads a menu resource from the executable file and attaches it to a **CMenu** object.
LoadMenuIndirect	Loads a menu from a menu template in memory and attaches it to a **CMenu** object.
DestroyMenu	Destroys the menu attached to a **CMenu** object and frees any memory that the menu occupied.

Menu Operations—Public Members

DeleteMenu	Deletes a specified item from the menu. If the menu item has an associated pop-up menu, destroys the handle to the pop-up menu and frees the memory used by it.
TrackPopupMenu	Displays a floating pop-up menu at the specified location and tracks the selection of items on the pop-up menu.

Menu Item Operations—Public Members

AppendMenu	Appends a new item to the end of this menu.
CheckMenuItem	Places check marks next to or removes check marks from menu items in the pop-up menu.
EnableMenuItem	Enables, disables, or dims (grays) a menu item.
GetMenuItemCount	Determines the number of items in a pop-up or top-level menu.
GetMenuItemID	Obtains the menu-item identifier for a menu item located at the specified position.
GetMenuState	Returns the status of the specified menu item or the number of items in a pop-up menu.
GetMenuString	Retrieves the label of the specified menu item.
GetSubMenu	Retrieves a pointer to a pop-up menu.
InsertMenu	Inserts a new menu item at the specified position, moving other items down the menu.
ModifyMenu	Changes an existing menu item at the specified position.

RemoveMenu	Deletes a menu item with an associated pop-up menu from the specified menu.
SetMenuItemBitmaps	Associates the specified check-mark bitmaps with a menu item.

Overridables—Public Members

DrawItem	Called by the framework when a visual aspect of an owner-drawn menu changes.
MeasureItem	Called by the framework to determine menu dimensions when an owner-drawn menu is created.

Member Functions

CMenu::AppendMenu

BOOL AppendMenu(UINT *nFlags*, **UINT** *nIDNewItem* = **0,**
LPCSTR *lpszNewItem* = **NULL);**

BOOL AppendMenu(UINT *nFlags*, **UINT** *nIDNewItem*,
const CBitmap* *pBmp* **);**

nFlags Specifies information about the state of the new menu item when it is added to the menu. It consists of one or more of the values listed in the "Remarks" section.

nIDNewItem Specifies either the command ID of the new menu item or, if *nFlags* is set to **MF_POPUP**, the menu handle (**HMENU**) of a pop-up menu. The *nIDNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpszNewItem Specifies the content of the new menu item. The *nFlags* parameter is used to interpret *lpszNewItem* in the following way:

nFlags	Interpretation of lpszNewItem
MF_OWNERDRAW	Contains an application-supplied 32-bit value that the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application when it processes **WM_MEASUREITEM** and **WM_DRAWITEM** messages. The value is stored in the **itemData** member of the structure supplied with those messages.
MF_STRING	Contains a pointer to a null-terminated string. This is the default interpretation.
MF_SEPARATOR	The *lpszNewItem* parameter is ignored (not needed).

pBmp Points to a **CBitmap** object that will be used as the menu item.

Remarks

Appends a new item to the end of a menu. The application can specify the state of the menu item by setting values in *nFlags*. When *nIDNewItem* specifies a pop-up menu, it becomes part of the menu to which it is appended. If that menu is destroyed, the appended menu will also be destroyed. An appended menu should be detached from a **CMenu** object to avoid conflict. Note that **MF_STRING** and **MF_OWNERDRAW** are not valid for the bitmap version of **AppendMenu**.

The following list describes the flags that may be set in *nFlags*:

- **MF_CHECKED** Acts as a toggle with **MF_UNCHECKED** to place the default check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark on" bitmap is displayed.
- **MF_UNCHECKED** Acts as a toggle with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark off" bitmap is displayed.
- **MF_DISABLED** Disables the menu item so that it cannot be selected but does not dim it.
- **MF_ENABLED** Enables the menu item so that it can be selected and restores it from its dimmed state.
- **MF_GRAYED** Disables the menu item so that it cannot be selected and dims it.
- **MF_MENUBARBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. The new pop-up menu column will be separated from the old column by a vertical dividing line.

- **MF_MENUBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. No dividing line is placed between the columns.
- **MF_OWNERDRAW** Specifies that the item is an owner-draw item. When the menu is displayed for the first time, the window that owns the menu receives a **WM_MEASUREITEM** message, which retrieves the height and width of the menu item. The **WM_DRAWITEM** message is the one sent whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.
- **MF_POPUP** Specifies that the menu item has a pop-up menu associated with it. The ID parameter specifies a handle to a pop-up menu that is to be associated with the item. This is used for adding either a top-level pop-up menu or a hierarchical pop-up menu to a pop-up menu item.
- **MF_SEPARATOR** Draws a horizontal dividing line. Can only be used in a pop-up menu. This line cannot be dimmed, disabled, or highlighted. Other parameters are ignored.
- **MF_STRING** Specifies that the menu item is a character string.

Each of the following groups lists flags that are mutually exclusive and cannot be used together:

- **MF_DISABLED**, **MF_ENABLED**, and **MF_GRAYED**
- **MF_STRING**, **MF_OWNERDRAW**, **MF_SEPARATOR**, and the bitmap version
- **MF_MENUBARBREAK** and **MF_MENUBREAK**
- **MF_CHECKED** and **MF_UNCHECKED**

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application should call **CWnd::DrawMenuBar**.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CWnd::DrawMenuBar**, **CMenu::InsertMenu**, **CMenu::RemoveMenu**, **CMenu::SetMenuItemBitmaps**, **CMenu::Detach**, **::AppendMenu**

CMenu::Attach

BOOL Attach(HMENU *hMenu* **);**

hMenu Specifies a handle to a Windows menu.

Remarks

Attaches an existing Windows menu to a **CMenu** object. This function should not be called if a menu is already attached to the **CMenu** object. The menu handle is stored in the **m_hMenu** data member.

Return Value

Nonzero if the operation was successful; otherwise 0.

See Also

CMenu::Detach, **CMenu::CMenu**

CMenu::CheckMenuItem

UINT CheckMenuItem(UINT *nIDCheckItem*, **UINT** *nCheck* **);**

nIDCheckItem Specifies the menu item to be checked, as determined by *nCheck*.

nCheck Specifies how to check the menu item and how to determine the item's position in the menu. The *nCheck* parameter can be a combination of **MF_CHECKED** or **MF_UNCHECKED** with **MF_BYPOSITION** or **MF_BYCOMMAND** flags. These flags can be combined by using the bitwise-OR operator. They have the following meanings:

- **MF_BYCOMMAND** Specifies that the parameter gives the command ID of the existing menu item. This is the default.
- **MF_BYPOSITION** Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.
- **MF_CHECKED** Acts as a toggle with **MF_UNCHECKED** to place the default check mark next to the item.
- **MF_UNCHECKED** Acts as a toggle with **MF_CHECKED** to remove a check mark next to the item.

Remarks

Adds check marks to or removes check marks from menu items in the pop-up menu. The *nIDCheckItem* parameter specifies the item to be modified. The *nIDCheckItem* parameter may identify a pop-up menu item as well as a menu item. No special steps are required to check a pop-up menu item. Top-level menu items cannot be checked. A pop-up menu item must be checked by position since it does not have a menu-item identifier associated with it.

Return Value

The previous state of the item: **MF_CHECKED** or **MF_UNCHECKED**, or –1 if the menu item did not exist.

See Also

CMenu::GetMenuState, **::CheckMenuItem**

CMenu::CMenu

CMenu( );

Remarks

The menu is not created until you call one of the create or load member functions of **CMenu**, as listed in "See Also."

See Also

CMenu::CreateMenu, **CMenu::CreatePopupMenu**, **CMenu::LoadMenu**, **CMenu::LoadMenuIndirect**, **CMenu::Attach**

CMenu::CreateMenu

BOOL CreateMenu( );

Remarks

Creates a menu and attaches it to the **CMenu** object. The menu is initially empty. Menu items can be added by using the **AppendMenu** or **InsertMenu** member function. If the menu is assigned to a window, it is automatically destroyed when the window is destroyed.

Before exiting, an application must free system resources associated with a menu if the menu is not assigned to a window. An application frees a menu by calling the **DestroyMenu** member function.

Return Value

Nonzero if the menu was created successfully; otherwise 0.

See Also

CMenu::CMenu, **CMenu::DestroyMenu**, **CMenu::InsertMenu**, **CWnd::SetMenu**, **::CreateMenu**, **CMenu::AppendMenu**

CMenu::CreatePopupMenu

BOOL CreatePopupMenu( );

Remarks

Creates a pop-up menu and attaches it to the **CMenu** object. The menu is initially empty. Menu items can be added by using the **AppendMenu** or **InsertMenu** member function. The application can add the pop-up menu to an existing menu or pop-up menu. The **TrackPopupMenu** member function may be used to display this menu as a floating pop-up menu and to track selections on the pop-up menu. If the menu is assigned to a window, it is automatically destroyed when the window is destroyed. If the menu is added to an existing menu, it is automatically destroyed when that menu is destroyed.

Before exiting, an application must free system resources associated with a pop-up menu if the menu is not assigned to a window. An application frees a menu by calling the **DestroyMenu** member function.

Return Value Nonzero if the pop-up menu was successfully created; otherwise 0.

See Also **CMenu::CreateMenu**, **CMenu::InsertMenu**, **CWnd::SetMenu**, **CMenu::TrackPopupMenu**, **::CreatePopupMenu**, **CMenu::AppendMenu**

CMenu::DeleteMenu

BOOL DeleteMenu(UINT *nPosition*, **UINT** *nFlags* **);**

nPosition Specifies the menu item that is to be deleted, as determined by *nFlags*.

nFlags Is used to interpret *nPosition* in the following way:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Remarks Deletes an item from the menu. If the menu item has an associated pop-up menu, **DeleteMenu** destroys the handle to the pop-up menu and frees the memory used by the pop-up menu. Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application must call **CWnd::DrawMenuBar**.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CWnd::DrawMenuBar**, **::DeleteMenu**

CMenu::DeleteTempMap

static void PASCAL DeleteTempMap( );

Remarks Called automatically by the **CWinApp** idle-time handler, **DeleteTempMap** deletes any temporary **CMenu** objects created by the **FromHandle** member function. **DeleteTempMap** detaches the Windows menu object attached to a temporary **CMenu** object before deleting the **CMenu** object.

CMenu::DestroyMenu

BOOL DestroyMenu( );

Remarks

Destroys the menu and any Windows operating system resources that were used. The menu is detached from the **CMenu** object before it is destroyed. The Windows **DestroyMenu** function is automatically called in the **CMenu** destructor.

Return Value

Nonzero if the menu is destroyed; otherwise 0.

See Also

::DestroyMenu

CMenu::Detach

HMENU Detach( );

Remarks

Detaches a Windows menu from a **CMenu** object and returns the handle. The **m_hMenu** data member is set to **NULL**.

Return Value

The handle, of type **HMENU**, to a Windows menu, if successful; otherwise **NULL**.

See Also

CMenu::Attach

CMenu::DrawItem

virtual void DrawItem(LPDRAWITEMSTRUCT *lpDrawItemStruct* **);**

lpDrawItemStruct A pointer to a **DRAWITEMSTRUCT** structure that contains information about the type of drawing required.

Remarks

Called by the framework when a visual aspect of an owner-drawn menu changes. The *itemAction* member of the **DRAWITEMSTRUCT** structure defines the drawing action that is to be performed. Override this member function to implement drawing for an owner-draw **CMenu** object. The application should restore all graphics device interface (GDI) objects selected for the display context supplied in *lpDrawItemStruct* before the termination of this member function.

See **CWnd::OnDrawItem** on page 964 for a description of the **DRAWITEMSTRUCT** structure.

CMenu::EnableMenuItem

UINT EnableMenuItem(UINT *nIDEnableItem*, **UINT** *nEnable* **);**

nIDEnableItem Specifies the menu item to be enabled, as determined by *nEnable*. This parameter can specify pop-up menu items as well as standard menu items.

nEnable Specifies the action to take. It can be a combination of **MF_DISABLED**, **MF_ENABLED**, or **MF_GRAYED**, with **MF_BYCOMMAND** or **MF_BYPOSITION**. These values can be combined by using the bitwise-OR operator. These values have the following meanings:

- **MF_BYCOMMAND** Specifies that the parameter gives the command ID of the existing menu item. This is the default.
- **MF_BYPOSITION** Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.
- **MF_DISABLED** Disables the menu item so that it cannot be selected but does not dim it.
- **MF_ENABLED** Enables the menu item so that it can be selected and restores it from its dimmed state.
- **MF_GRAYED** Disables the menu item so that it cannot be selected and dims it.

Remarks

Enables, disables, or dims a menu item. The **CreateMenu**, **InsertMenu**, **ModifyMenu**, and **LoadMenuIndirect** member functions can also set the state (enabled, disabled, or dimmed) of a menu item.

Using the **MF_BYPOSITION** value requires an application to use the correct **CMenu**. If the **CMenu** of the menu bar is used, a top-level menu item (an item in the menu bar) is affected. To set the state of an item in a pop-up or nested pop-up menu by position, an application must specify the **CMenu** of the pop-up menu. When an application specifies the **MF_BYCOMMAND** flag, Windows checks all pop-up menu items that are subordinate to the **CMenu**; therefore, unless duplicate menu items are present, using the **CMenu** of the menu bar is sufficient.

Return Value

Previous state (**MF_DISABLED**, **MF_ENABLED**, or **MF_GRAYED**) or –1 if not valid.

See Also

CMenu::GetMenuState, **::EnableMenuItem**

CMenu::FromHandle

static CMenu* PASCAL FromHandle(HMENU *hMenu* **);**

hMenu A Windows handle to a menu.

Remarks Returns a pointer to a **CMenu** object given a Windows handle to a menu. If a **CMenu** object is not already attached to the Windows menu object, a temporary **CMenu** object is created and attached. This temporary **CMenu** object is only valid until the next time the application has idle time in its event loop, at which time all temporary objects are deleted.

Return Value A pointer to a **CMenu** that may be temporary or permanent.

CMenu::GetMenuItemCount

UINT GetMenuItemCount( ) const;

Remarks Determines the number of items in a pop-up or top-level menu.

Return Value The number of items in the menu if the function is successful; otherwise –1.

See Also **CWnd::GetMenu**, **CMenu::GetMenuItemID**, **CMenu::GetSubMenu**, **::GetMenuItemCount**

CMenu::GetMenuItemID

UINT GetMenuItemID(int *nPos* **) const;**

nPos Specifies the position (zero-based) of the menu item whose ID is being retrieved.

Remarks Obtains the menu-item identifier for a menu item located at the position defined by *nPos*.

Return Value The item ID for the specified item in a pop-up menu if the function is successful. If the specified item is a pop-up menu (as opposed to an item within the pop-up menu), the return value is –1. If *nPos* corresponds to a **SEPARATOR** menu item, the return value is 0.

See Also **CWnd::GetMenu**, **CMenu::GetMenuItemCount**, **CMenu::GetSubMenu**

CMenu::GetMenuState

UINT GetMenuState(UINT *nID*, **UINT** *nFlags* **) const;**

nID Specifies the menu item ID, as determined by *nFlags*.

nFlags Specifies the nature of *nID*. It can be one of the following values:

- **MF_BYCOMMAND** Specifies that the parameter gives the command ID of the existing menu item. This is the default.
- **MF_BYPOSITION** Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Remarks

Returns the status of the specified menu item or the number of items in a pop-up menu.

Return Value

The value –1 if the specified item does not exist. If *nID* identifies a pop-up menu, the high-order byte contains the number of items in the pop-up menu and the low-order byte contains the menu flags associated with the pop-up menu. Otherwise the return value is a mask (Boolean OR) of the values from the following list (this mask describes the status of the menu item that *nID* identifies):

- **MF_CHECKED** Acts as a toggle with **MF_UNCHECKED** to place the default check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark on" bitmap is displayed.
- **MF_DISABLED** Disables the menu item so that it cannot be selected but does not dim it.
- **MF_ENABLED** Enables the menu item so that it can be selected and restores it from its dimmed state. Note that the value of this constant is 0; an application should not test against 0 for failure when using this value.
- **MF_GRAYED** Disables the menu item so that it cannot be selected and dims it.
- **MF_MENUBARBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. The new pop-up menu column will be separated from the old column by a vertical dividing line.
- **MF_MENUBREAK** Places the item on a new line in static menus or in a new column in pop-up menus. No dividing line is placed between the columns.
- **MF_SEPARATOR** Draws a horizontal dividing line. Can only be used in a pop-up menu. This line cannot be dimmed, disabled, or highlighted. Other parameters are ignored.

- **MF_UNCHECKED** Acts as a toggle with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see the **SetMenuItemBitmaps** member function), the "check mark off" bitmap is displayed. Note that the value of this constant is 0; an application should not test against 0 for failure when using this value.

See Also **::GetMenuState**, **CMenu::CheckMenuItem**, **CMenu::EnableMenuItem**

CMenu::GetMenuString

int GetMenuString(UINT *nIDItem*, **LPSTR** *lpString*, **int** *nMaxCount*, **UINT** *nFlags* **) const;**

nIDItem Specifies the integer identifier of the menu item or the offset of the menu item in the menu, depending on the value of *nFlags*.

lpString Points to the buffer that is to receive the label. You can pass a **CString** object for this parameter.

nMaxCount Specifies the maximum length (in bytes) of the label to be copied. If the label is longer than the maximum specified in *nMaxCount*, the extra characters are truncated.

nFlags Specifies the interpretation of the *nIDItem* parameter. It can be one of the following values:

nFlags	Interpretation of nIDItem
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

Remarks Copies the label of the specified menu item to the specified buffer. The *nMaxCount* parameter should be one larger than the number of characters in the label to accommodate the null character that terminates a string.

Return Value Specifies the actual number of bytes copied to the buffer, not including the null terminator.

See Also **CWnd::GetMenu**, **CMenu::GetMenuItemID**, **::GetMenuString**

CMenu::GetSafeHmenu

HMENU GetSafeHmenu() const;

Remarks

Returns the **HMENU** wrapped by this **CMenu** object, or a **NULL CMenu** pointer.

CMenu::GetSubMenu

CMenu* GetSubMenu(int *nPos* **) const;**

nPos Specifies the position of the pop-up menu contained in the menu. Position values start at 0 for the first menu item. The pop-up menu's identifier cannot be used in this function.

Remarks

Retrieves the **CMenu** object of a pop-up menu.

Return Value

A pointer to a **CMenu** object whose **m_hMenu** member contains a handle to the pop-up menu if a pop-up menu exists at the given position; otherwise **NULL**. If a **CMenu** object does not exist, then a temporary one is created. The **CMenu** pointer returned should not be stored.

See Also

::GetSubMenu

CMenu::InsertMenu

BOOL InsertMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem* = **0, LPCSTR** *lpszNewItem* = **NULL);**

BOOL InsertMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem*, **const CBitmap*** *pBmp* **);**

nPosition Specifies the menu item before which the new menu item is to be inserted. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0. If *nPosition* is –1, the new menu item is appended to the end of the menu.

nFlags Specifies how *nPosition* is interpreted and specifies information about the state of the new menu item when it is added to the menu. For a list of the flags that may be set, see the **AppendMenu** member function. To specify more than one value, use the bitwise-OR operator to combine them with the **MF_BYCOMMAND** or **MF_BYPOSITION** flag.

nIDNewItem Specifies either the command ID of the new menu item or, if *nFlags* is set to **MF_POPUP**, the menu handle (**HMENU**) of the pop-up menu. The *nIDNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpszNewItem Specifies the content of the new menu item. The *nFlags* parameter can be used to interpret *lpszNewItem* in the following ways:

nFlags	Interpretation of lpszNewItem
MF_OWNERDRAW	Contains an application-supplied 32-bit value that the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** member of the structure supplied by the **WM_MEASUREITEM** and **WM_DRAWITEM** messages. These messages are sent when the menu item is initially displayed or is changed.
MF_STRING	Contains a long pointer to a null-terminated string. This is the default interpretation.
MF_SEPARATOR	The *lpszNewItem* parameter is ignored (not needed).

pBmp Points to a **CBitmap** object that will be used as the menu item.

Remarks

Inserts a new menu item at the position specified by *nPosition* and moves other items down the menu. The application can specify the state of the menu item by setting values in *nFlags*. Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application should call **CWnd::DrawMenuBar**. When *nIDNewItem* specifies a pop-up menu, it becomes part of the menu in which it is inserted. If that menu is destroyed, the inserted menu will also be destroyed. An inserted menu should be detached from a **CMenu** object to avoid conflict.

If the active multiple document interface (MDI) child window is maximized and an application inserts a pop-up menu into the MDI application's menu by calling this function and specifying the **MF_BYPOSITION** flag, the menu is inserted one position farther left than expected. This happens because the Control menu of the active MDI child window is inserted into the first position of the MDI frame window's menu bar. To position the menu properly, the application must add 1 to the position value that would otherwise be used. An application can use the **WM_MDIGETACTIVE** message to determine whether the currently active child window is maximized.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CMenu::AppendMenu**, **CWnd::DrawMenuBar**, **CMenu::SetMenuItemBitmaps**, **CMenu::Detach**, **::InsertMenu**

CMenu::LoadMenu

BOOL LoadMenu(LPCSTR *lpszResourceName* **);**

BOOL LoadMenu(UINT *nIDResource* **);**

lpszResourceName Points to a null-terminated string that contains the name of the menu resource to load.

nIDResource Specifies the menu ID of the menu resource to load.

Remarks Loads a menu resource from the application's executable file and attaches it to the **CMenu** object. Before exiting, an application must free system resources associated with a menu if the menu is not assigned to a window. An application frees a menu by calling the **DestroyMenu** member function.

Return Value Nonzero if the menu resource was loaded successfully; otherwise 0.

See Also **CMenu::AppendMenu**, **CMenu::DestroyMenu**, **CMenu::LoadMenuIndirect**, **::LoadMenu**.

CMenu::LoadMenuIndirect

BOOL LoadMenuIndirect(const void FAR* *lpMenuTemplate* **);**

lpMenuTemplate Points to a menu template (which is a single **MENUITEMTEMPLATEHEADER** structure and a collection of one or more **MENUITEMTEMPLATE** structures).

The **MENUITEMTEMPLATEHEADER** structure has the following generic form:

```
typedef struct {
   UINT   versionNumber;
   UINT   offset;
} MENUITEMTEMPLATEHEADER;
```

The **MENUITEMTEMPLATE** structure has the following generic form:

```
typedef struct {
   UINT mtOption;
   UINT mtID;
   char mtString[1];
} MENUITEMTEMPLATE;
```

For more information on the above two structures, see the *Windows Software Development Kit* (SDK).

Remarks Loads a resource from a menu template in memory and attaches it to the **CMenu** object. A menu template is a header followed by a collection of one or more **MENUITEMTEMPLATE** structures, each of which may contain one or more menu items and pop-up menus. The version number should be 0. The **mtOption** flags should include **MF_END** for the last item in a pop-up list and for the last item in the main list. See the **AppendMenu** member function for other flags. The **mtId** member must be omitted from the **MENUITEMTEMPLATE** structure when **MF_POPUP** is specified in **mtOption**. The space allocated for the **MENUITEMTEMPLATE** structure must be large enough for **mtString** to contain the name of the menu item as a null-terminated string.

Before exiting, an application must free system resources associated with a menu if the menu is not assigned to a window. An application frees a menu by calling the **DestroyMenu** member function.

Return Value Nonzero if the menu resource was loaded successfully; otherwise 0.

See Also **CMenu::DestroyMenu**, **CMenu::LoadMenu**, **::LoadMenuIndirect**, **CMenu::AppendMenu**

CMenu::MeasureItem

virtual void MeasureItem(LPMEASUREITEMSTRUCT *lpMeasureItemStruct* **);**

lpMeasureItemStruct A pointer to a **MEASUREITEMSTRUCT** structure.

Remarks

Called by the framework when a menu with the owner-draw style is created. By default, this member function does nothing. Override this member function and fill in the **MEASUREITEM** structure to inform the Windows operating system of the menu's dimensions.

See **CWnd::OnMeasureItem** on page 980 for a description of the **MEASUREITEM** structure.

CMenu::ModifyMenu

BOOL ModifyMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem* **= 0, LPCSTR** *lpszNewItem* **= NULL);**

BOOL ModifyMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem*, **const CBitmap*** *pBmp* **);**

nPosition Specifies the menu item to be changed. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

nFlags Specifies how *nPosition* is interpreted and gives information about the changes to be made to the menu item. For a list of flags that may be set, see the **AppendMenu** member function.

nIDNewItem Specifies either the command ID of the modified menu item or, if *nFlags* is set to **MF_POPUP**, the menu handle (**HMENU**) of a pop-up menu. The *nIDNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpszNewItem Specifies the content of the new menu item. The *nFlags* parameter can be used to interpret *lpszNewItem* in the following ways:

nFlags	Interpretation of lpszNewItem
MF_OWNERDRAW	Contains an application-supplied 32-bit value that the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application when it processes **MF_MEASUREITEM** and **MF_DRAWITEM**.
MF_STRING	Contains a long pointer to a null-terminated string or to a **CString**.
MF_SEPARATOR	The *lpszNewItem* parameter is ignored (not needed).

pBmp Points to a **CBitmap** object that will be used as the menu item.

Remarks

Changes an existing menu item at the position specified by *nPosition*. The application specifies the new state of the menu item by setting values in *nFlags*. If this function replaces a pop-up menu associated with the menu item, it destroys the old pop-up menu and frees the memory used by the pop-up menu. When *nIDNewItem* specifies a pop-up menu, it becomes part of the menu in which it is inserted. If that menu is destroyed, the inserted menu will also be destroyed. An inserted menu should be detached from a **CMenu** object to avoid conflict.

Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application should call **CWnd::DrawMenuBar**. To change the attributes of existing menu items, it is much faster to use the **CheckMenuItem** and **EnableMenuItem** member functions.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CMenu::AppendMenu, **CMenu::InsertMenu**, **CMenu::CheckMenuItem**, **CWnd::DrawMenuBar**, **CMenu::EnableMenuItem**, **CMenu::SetMenuItemBitmaps**, **CMenu::Detach**, **::ModifyMenu**

CMenu::RemoveMenu

BOOL RemoveMenu(UINT *nPosition***, UINT** *nFlags* **);**

nPosition Specifies the menu item to be removed. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

nFlags Specifies how *nPosition* is interpreted.

Remarks Deletes a menu item with an associated pop-up menu from the menu. It does not destroy the handle for a pop-up menu, so the menu can be reused. Before calling this function, the application may call the **GetSubMenu** member function to retrieve the pop-up **CMenu** object for reuse. Whenever a menu that resides in a window is changed (whether or not the window is displayed), the application must call **CWnd::DrawMenuBar**.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CWnd::DrawMenuBar**, **CMenu::GetSubMenu**, **::RemoveMenu**

CMenu::SetMenuItemBitmaps

BOOL SetMenuItemBitmaps(UINT *nPosition*, **UINT** *nFlags*,
const CBitmap* *pBmpUnchecked*, **const CBitmap*** *pBmpChecked* **);**

nPosition Specifies the menu item to be changed. The *nFlags* parameter can be used to interpret *nPosition* in the following ways:

nFlags	Interpretation of nPosition
MF_BYCOMMAND	Specifies that the parameter gives the command ID of the existing menu item. This is the default if neither **MF_BYCOMMAND** nor **MF_BYPOSITION** is set.
MF_BYPOSITION	Specifies that the parameter gives the position of the existing menu item. The first item is at position 0.

nFlags Specifies how *nPosition* is interpreted.

pBmpUnchecked Specifies the bitmap to use for menu items that are not checked.

pBmpChecked Specifies the bitmap to use for menu items that are checked.

Remarks Associates the specified bitmaps with a menu item. Whether the menu item is checked or unchecked, the Windows operating system displays the appropriate bitmap next to the menu item. If either *pBmpUnchecked* or *pBmpChecked* is

NULL, then the Windows operating system displays nothing next to the menu item for the corresponding attribute. If both parameters are **NULL**, the Windows operating system uses the default check mark when the item is checked and removes the check mark when the item is unchecked. When the menu is destroyed, these bitmaps are not destroyed; the application must destroy them.

The Windows **GetMenuCheckMarkDimensions** function retrieves the dimensions of the default check mark used for menu items. The application uses these values to determine the appropriate size for the bitmaps supplied with this function. Get the size, create your bitmaps, then set them.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **::GetMenuCheckMarkDimensions**, **::SetMenuItemBitmaps**

CMenu::TrackPopupMenu

BOOL TrackPopupMenu(UINT *nFlags*, **int** *x*, **int** *y*, **CWnd*** *pWnd*, **LPCRECT** *lpRect* = **0);**

nFlags Specifies a screen-position flag and a mouse-button flag. The screen-position flag can be one of the following:

- **TPM_CENTERALIGN** Centers the pop-up menu horizontally relative to the coordinate specified by *x*.
- **TPM_LEFTALIGN** Positions the pop-up menu so that its left side is aligned with the coordinate specified by *x*.
- **TPM_RIGHTALIGN** Positions the pop-up menu so that its right side is aligned with the coordinate specified by *x*.

The mouse-button flag can be one of the following:

- **TPM_LEFTBUTTON** Causes the pop-up menu to track the left mouse button.
- **TPM_RIGHTBUTTON** Causes the pop-up menu to track the right mouse button.

x Specifies the horizontal position in screen coordinates of the pop-up menu. Depending on the value of the *nFlags* parameter, the menu can be left-aligned, right-aligned, or centered relative to this position.

y Specifies the vertical position in screen coordinates of the top of the menu on the screen.

pWnd Identifies the window that owns the pop-up menu. This window receives all **WM_COMMAND** messages from the menu. In Windows 3.1, the window does not receive **WM_COMMAND** messages until **TrackPopupMenu** returns. In Windows 3.0, the window receives **WM_COMMAND** messages before **TrackPopupMenu** returns.

lpRect Points to a **RECT** structure or **CRect** object that contains the screen coordinates of a rectangle within which the user can click without dismissing the pop-up menu. If this parameter is **NULL**, the pop-up menu is dismissed if the user clicks outside the pop-up menu. This must be **NULL** for Windows 3.0.

Windows 3.1 Only The use of the following constants for *lpRect* is new in Windows 3.1:

- **TPM_CENTERALIGN**
- **TPM_LEFTALIGN**
- **TPM_RIGHTALIGN**
- **TPM_RIGHTBUTTON** ♦

Remarks Displays a floating pop-up menu at the specified location and tracks the selection of items on the pop-up menu. A floating pop-up menu can appear anywhere on the screen.

Return Value Nonzero if the function is successful; otherwise 0.

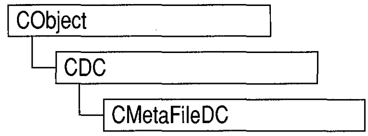
See Also **CMenu::CreatePopupMenu**, **CMenu::GetSubMenu**, **::TrackPopupMenu**

Data Members

CMenu::m_hMenu

Remarks Specifies the **HMENU** handle of the Windows menu attached to the **CMenu** object.

class CMetaFileDC : public CDC



A Windows metafile contains a sequence of graphics device interface (GDI) commands that you can replay to create a desired image or text.

To implement a Windows metafile, first create a **CMetaFileDC** object. Invoke the **CMetaFileDC** constructor, then call the **Create** member function, which creates a Windows metafile device context and attaches it to the **CMetaFileDC** object.

Next send the **CMetaFileDC** object the sequence of **CDC** GDI commands that you intend for it to replay. Only those GDI commands that create output, such as **MoveTo** and **LineTo**, may be used.

After you have sent the desired commands to the metafile, call the **Close** member function, which closes the metafile device contexts and returns a metafile handle. Then dispose of the **CMetaFileDC** object.

CDC::PlayMetaFile can then use the metafile handle to play the metafile repeatedly. The metafile can also be manipulated by Windows functions such as **CopyMetaFile**, which copies a metafile to disk.

When the metafile is no longer needed, delete it from memory with the **DeleteMetaFile** Windows function.

You may also implement the **CMetaFileDC** object so that it can handle both output calls and attribute GDI calls such as **GetTextExtent**. Such a metafile is more flexible and can more easily reuse general GDI code, which often consists of a mix of output and attribute calls. The **CMetaFileDC** class inherits two device contexts, **m_hDC** and **m_hAttribDC**, from **CDC**. The **m_hDC** device context handles all **CDC** GDI output calls and the **m_hAttribDC** device context handles all **CDC** GDI attribute calls. Normally, these two device contexts refer to the same device. In the case of **CMetaFileDC**, the attribute DC is set to **NULL** by default. Create a second device context that points to the screen, a printer, or device other than a metafile, then call the **SetAttribDC** member function to associate the new device context with **m_hAttribDC**. GDI calls for information will now be directed to the new **m_hAttribDC**. Output GDI calls will go to **m_hDC**, which represents the metafile.

#include <afxext.h>

See Also **CDC**

Construction/Destruction—Public Members

CMetaFileDC	Constructs a **CMetaFileDC** object.

Initialization—Public Members

Create	Creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.

Operations—Public Members

Close	Closes the device context and creates a metafile handle.

Member Functions

CMetaFileDC::Close

HMETAFILE Close( );

Remarks

Closes the metafile device context and creates a Windows metafile handle that can be used to play the metafile by using the **CDC::PlayMetaFile** member function. The Windows metafile handle can also be used to manipulate the metafile with Windows functions such as **CopyMetaFile**.

Delete the metafile after use by calling the Windows **DeleteMetaFile** function.

Return Value

A valid **HMETAFILE** if the function is successful; otherwise **NULL**.

See Also

CDC::PlayMetaFile, ::CloseMetaFile, ::GetMetaFileBits, ::CopyMetaFile, ::DeleteMetaFile

CMetaFileDC::CMetaFileDC

CMetaFileDC( );

Remarks

Construct a **CMetaFileDC** object in two steps. First, call **CMetaFileDC**, then call **Create**, which creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.

See Also

CMetaFileDC::Create

CMetaFileDC::Create

BOOL Create(LPCSTR *lpszFilename* = **NULL);**

lpszFilename Points to a null-terminated character string. Specifies the filename of the metafile to create. If *lpszFilename* is **NULL**, a new in-memory metafile is created.

Remarks

Construct a **CMetaFileDC** object in two steps. First, call the constructor **CMetaFileDC**, then call **Create**, which creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.

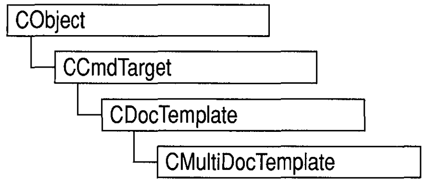
Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CMetaFileDC::CMetaFileDC, **CDC::SetAttribDC**, **::CreateMetaFile**

class CMultiDocTemplate : public CDocTemplate



The **CMultiDocTemplate** class defines a document template that implements the multiple document interface (MDI). An MDI application uses the main frame window as a workspace in which the user can open zero or more document frame windows, each of which displays a document. For a more detailed description of the MDI, see *The Windows Interface: An Application Design Guide.*

A document template defines the relationship between three types of classes:

- A document class, which you derive from **CDocument**.
- A view class, which displays data from the document class listed above. You can derive this class from **CView**, **CScrollView**, **CFormView**, or **CEditView**. (You can also use **CEditView** directly.)
- A frame window class, which contains the view. For an MDI document template, you can derive this class from **CMDIChildWnd**, or, if you don't need to customize the behavior of the document frame windows, you can use **CMDIChildWnd** directly without deriving your own class.

An MDI application can support more than one type of document, and documents of different types can be open at the same time. Your application has one document template for each document type that it supports. For example, if your MDI application supports both spreadsheets and text documents, the application has two **CMultiDocTemplate** objects.

The application uses the document template(s) when the user creates a new document. If the application supports more than one type of document, then the framework gets the names of the supported document types from the document templates and displays them in a list in the File New dialog box. Once the user has selected a document type, the application creates a document object, a frame window object, and a view object and attaches them to each other.

You don't need to call any member functions of **CMultiDocTemplate** except the constructor. The framework handles **CMultiDocTemplate** objects internally.

See Also **CDocTemplate**, **CDocument**, **CMDIChildWnd**, **CSingleDocTemplate**, **CView**, **CWinApp**

Construction/Destruction—Public Members

CMultiDocTemplate	Constructs a **CMultiDocTemplate** object.

Member Functions

CMultiDocTemplate::CMultiDocTemplate

CMultiDocTemplate(UINT *nIDResource*, **CRuntimeClass*** *pDocClass*, **CRuntimeClass*** *pFrameClass*, **CRuntimeClass*** *pViewClass* **);**

nIDResource Specifies the ID of the resources used with the document type. This may include menu, icon, accelerator table, and string resources.

The string resource consists of up to seven substrings separated by the '\n' character (the '\n' character is needed as a place holder if a substring is not included; however, trailing '\n' characters are not necessary); these substrings describe the document type. For information about the substrings, see **CDocTemplate::GetDocString**. This string resource is found in the application's resource file. For example:

```
// MYCALC.RC
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_SHEETTYPE "\nSheet\nWorksheet\nWorksheets (*.myc)\n.myc\n
MyCalcSheet\nMyCalc Worksheet"
END
```

Note that the string begins with a '\n' character; this is because the first substring is not used for MDI applications and so is not included. You can edit this string using the String Editor in App Studio; the entire string appears as a single entry in the String Editor, not as seven separate entries.

For more information about these resource types, see the *App Studio User's Guide*.

pDocClass Points to the **CRuntimeClass** object of the document class. This class is a **CDocument**-derived class you define to represent your documents.

pFrameClass Points to the **CRuntimeClass** object of the frame-window class. This class can be a **CMDIChildWnd**-derived class, or it can be **CMDIChildWnd** itself if you want default behavior for your document frame windows.

pViewClass Points to the **CRuntimeClass** object of the view class. This class is a **CView**-derived class you define to display your documents.

Remarks

Constructs a **CMultiDocTemplate** object. Dynamically allocate one **CMultiDocTemplate** object for each document type that your application supports and pass each one to **CWinApp::AddDocTemplate** from the `InitInstance` member function of your application class.

See Also

CDocTemplate::GetDocString, **CWinApp::AddDocTemplate**, **CWinApp::InitInstance**, **CRuntimeClass**, **RUNTIME_CLASS**

Example

```
BOOL CMyApp::InitInstance()
{
        // ...
        // Establish all of the document types
        // supported by the application

        AddDocTemplate( new CMultiDocTemplate( IDR_SHEETTYPE,
                            RUNTIME_CLASS( CSheetDoc ),
                            RUNTIME_CLASS( CMDIChildWnd ),
                            RUNTIME_CLASS( CSheetView ) ) );

        AddDocTemplate( new CMultiDocTemplate( IDR_NOTETYPE,
                            RUNTIME_CLASS( CNoteDoc ),
                            RUNTIME_CLASS( CMDIChildWnd ),
                            RUNTIME_CLASS( CNoteView ) ) );
        // ...
}
```

class CNotSupportedException : public CException

A **CNotSupportedException** object represents an exception that is the result of a request for an unsupported feature. No further qualification is necessary or possible.

#include <afx.h>

Construction/Destruction—Public Member

CNotSupportedException	Constructs a **CNotSupportedException** object.

Member Functions

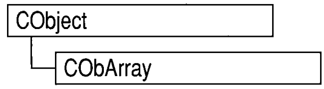
CNotSupportedException::CNotSupportedException

CNotSupportedException( );

Remarks Constructs a **CNotSupportedException** object. Do not use this constructor directly, but rather call the global function **AfxThrowNotSupportedException**. For more information about exception processing, see Chapter 16, "Exceptions," in the *Class Library User's Guide*.

See Also **AfxThrowNotSupportedException**

class CObArray : public CObject



The **CObArray** class supports arrays of **CObject** pointers. These object arrays are similar to C arrays, but they can dynamically shrink and grow as necessary. Array indexes always start at position 0. You can decide whether to fix the upper bound or allow the array to expand when you add elements past the current bound. Memory is allocated contiguously to the upper bound, even if some elements are null.

The elements of a **CObArray** object must fit in one 64K segment together with approximately 100 allocation overhead bytes. If **CObject** pointers are 16-bit near pointers (as they are in the small and medium memory models), then an array size limit is about 32,000 elements, but because there is only one data segment, the objects themselves will probably exhaust memory before the array does. If **CObject** pointers are 32-bit far pointers (as they are in the compact and large memory models), then an array size limit is about 16,000 elements.

As with a C array, the access time for a **CObArray** indexed element is constant and is independent of the array size. **CObArray** incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of **CObject** pointers is stored to an archive, either with the overloaded insertion operator or with the **Serialize** member function, each **CObject** element is, in turn, serialized along with its array index. If you need a dump of individual **CObject** elements in an array, you must set the depth of the **CDumpContext** object to 1 or greater. When a **CObArray** object is deleted, or when its elements are removed, only the **CObject** pointers are removed, not the objects they reference.

Array class derivation is similar to list derivation. For details on the derivation of a special-purpose list class, see Chapter 13, "Collections," in the *Class Library User's Guide.*

Note You must use the **IMPLEMENT_SERIAL** macro in the implementation of your derived class if you intend to serialize the array.

#include <afxcoll.h>

See Also **CStringArray**, **CPtrArray**, **CByteArray**, **CWordArray**, **CDWordArray**

Construction/Destruction—Public Members

CObArray	Constructs an empty array for **CObject** pointers.
~CObArray	Destroys a **CObArray** object.

Bounds—Public Members

GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations—Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; array not allowed to grow.
ElementAt	Returns a temporary reference to the element pointer within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

Member Functions

CObArray::Add

int Add(CObject* *newElement* **)**
throw(CMemoryException);

newElement The **CObject** pointer to be added to this array.

Remarks Adds a new element to the end of an array, growing the array by 1. If **SetSize** has been used with an *nGrowBy* value greater than 1, then extra memory may be allocated. However, the upper bound will increase by only 1.

Return Value The index of the added element.

See Also **CObArray::SetAt, CObArray::SetAtGrow, CObArray::InsertAt, CObArray::operator [ ]**

Example

```
    CObArray array;

    array.Add( new CAge( 21 ) ); // Element 0
    array.Add( new CAge( 40 ) ); // Element 1
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "Add example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
Add example: A CObArray with 2 elements
    [0] = a CAge at $442A 21
    [1] = a CAge at $4468 40
```

CObArray::CObArray

CObArray();

Remarks Constructs an empty **CObject** pointer array. The array grows one element at a time.

See Also **CObList::CObList**

Example See the **CObList** constructor for a listing of the `CAge` class used in all collection examples.

CObArray::~CObArray

~CObArray();

Remarks Destroys a **CObArray** object but does not destroy the **CObject** objects that are referenced in the array.

CObArray::ElementAt

CObject*& ElementAt(int *nIndex* **);**

nIndex An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

Remarks Returns a temporary reference to the element pointer within the array. It is used to implement the left-side assignment operator for arrays. Note that this is an advanced function that should be used only to implement special array operators.

Return Value A reference to a **CObject** pointer.

See Also **CObArray::operator [ ]**

CObArray::FreeExtra

void FreeExtra();

Remarks

Frees any extra memory that was allocated while the array was grown. This function has no effect on the size or upper bound of the array.

CObArray::GetAt

CObject* GetAt(int *nIndex* **) const;**

nIndex An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

Remarks

Returns the array element at the specified index.

Return Value

The **CObject** pointer element currently at this index; **NULL** if no element is stored at the index.

See Also

CObArray::SetAt, CObArray::operator [ ]

Example

```
CObArray array;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
ASSERT( *(CAge*) array.GetAt( 0 ) == CAge( 21 ) );
```

CObArray::GetSize

int GetSize() const;

Remarks

Returns the size of the array. Since indexes are zero-based, the size is 1 greater than the largest index.

See Also

CObArray::GetUpperBound, CObArray::SetSize

CObArray::GetUpperBound

int GetUpperBound() const;

Remarks

Returns the current upper bound of this array. Because array indexes are zero-based, this function returns a value 1 less than **GetSize**. The condition **GetUpperBound( )** = –1 indicates that the array contains no elements.

See Also

CObArray::GetSize, CObArray::SetSize

Example

```
CObArray array;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
ASSERT( array.GetUpperBound() == 1 ); // Largest index
```

CObArray::InsertAt

void InsertAt(int *nIndex*, **CObject*** *newElement*, **int** *nCount* **= 1)**
throw(CMemoryException);

void InsertAt(int *nStartIndex*, **CObArray*** *pNewArray* **)**
throw(CMemoryException);

nIndex An integer index that may be greater than the value returned by **GetUpperBound**.

newElement The **CObject** pointer to be placed in this array. A *newElement* of value **NULL** is allowed.

nCount The number of times this element should be inserted (defaults to 1).

nStartIndex An integer index that may be greater than the value returned by **GetUpperBound**.

pNewArray Another array that contains elements to be added to this array.

Remarks

The first version of **InsertAt** inserts one element (or multiple copies of an element) at a specified index in an array. In the process, it shifts up (by incrementing the index) the existing element at this index, and it shifts up all the elements above it. The second version inserts all the elements from another **CObArray** collection, starting at the *nStartIndex* position. The **SetAt** function, in contrast, replaces one specified array element and does not shift any elements.

See Also **CObArray::SetAt, CObArray::RemoveAt**

Example

```
    CObArray array;

    array.Add( new CAge( 21 ) ); // Element 0
    array.Add( new CAge( 40 ) ); // Element 1 (will become 2).
    array.InsertAt( 1, new CAge( 30 ) );  // New element 1
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "InsertAt example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
InsertAt example: A CObArray with 3 elements
    [0] = a CAge at $45C8 21
    [1] = a CAge at $4646 30
    [2] = a CAge at $4606 40
```

CObArray::RemoveAll

void RemoveAll();

Remarks Removes all the pointers from this array but does not actually delete the **CObject** objects. If the array is already empty, the function still works. The **RemoveAll** function frees all memory used for pointer storage.

Example

```
CObArray array;
CAge* pa1;
CAge* pa2;

array.Add( pa1 = new CAge( 21 ) ); // Element 0
array.Add( pa2 = new CAge( 40 ) ); // Element 1
ASSERT( array.GetSize() == 2 );
array.RemoveAll(); // Pointers removed but objects not deleted.
ASSERT( array.GetSize() == 0 );
delete pa1;
delete pa2;  // Cleans up memory.
```

CObArray::RemoveAt

void RemoveAt(**int** *nIndex*, **int** *nCount* **= 1);**

nIndex An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

nCount The number of elements to remove.

Remarks

Removes one or more elements starting at a specified index in an array. In the process, it shifts down all the elements above the removed element(s). It decrements the upper bound of the array but does not free memory. If you try to remove more elements than are contained in the array above the removal point, then the Debug version of the library asserts. The **RemoveAt** function removes the **CObject** pointer from the array, but it does not delete the object itself.

See Also

CObArray::SetAt, **CObArray::SetAtGrow**, **CObArray::InsertAt**

Example

```
   CObArray array;
   CObject* pa;

   array.Add( new CAge( 21 ) ); // Element 0
   array.Add( new CAge( 40 ) ); // Element 1
   if( ( pa = array.GetAt( 0 ) ) != NULL )
   {
       array.RemoveAt( 0 );  // Element 1 moves to 0.
       delete pa; // Delete the original element at 0.
   }
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "RemoveAt example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
RemoveAt example: A CObArray with 1 elements
    [0] = a CAge at $4606 40
```

CObArray::SetAt

void SetAt(int *nIndex*, **CObject*** *newElement* **);**

nIndex An integer index that is greater than or equal to 0 and less than or equal to the value returned by **GetUpperBound**.

newElement The object pointer to be inserted in this array. A **NULL** value is allowed.

Remarks

Sets the array element at the specified index. **SetAt** will not cause the array to grow. Use **SetAtGrow** if you want the array to grow automatically.

You must ensure that your index value represents a valid position in the array. If it is out of bounds, then the Debug version of the library asserts.

See Also

CObArray::GetAt, **CObArray::SetAtGrow**, **CObArray::ElementAt**, **CObArray::operator [ ]**

Example

```
   CObArray array;
   CObject* pa;

   array.Add( new CAge( 21 ) ); // Element 0
   array.Add( new CAge( 40 ) ); // Element 1
   if( ( pa = array.GetAt( 0 ) ) != NULL )
   {
       array.SetAt( 0, new CAge( 30 ) );  // Replace element 0.
       delete pa; // Delete the original element at 0.
   }
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "SetAt example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
SetAt example: A CObArray with 2 elements
    [0] = a CAge at $47E0 30
    [1] = a CAge at $47A0 40
```

CObArray::SetAtGrow

void SetAtGrow(int *nIndex*, **CObject*** *newElement* **)**
throw(CMemoryException);

nIndex An integer index that is greater than or equal to 0.

newElement The object pointer to be added to this array. A **NULL** value is allowed.

Remarks

Sets the array element at the specified index. The array grows automatically if necessary (that is, the upper bound is adjusted to accommodate the new element).

See Also

CObArray::GetAt, CObArray::SetAt, CObArray::ElementAt, CObArray::operator [ ]

Example

```
   CObArray array;

   array.Add( new CAge( 21 ) ); // Element 0
   array.Add( new CAge( 40 ) ); // Element 1
   array.SetAtGrow( 3, new CAge( 65 ) ); // Element 2 deliberately
                                         // skipped.
#ifdef _DEBUG
   afxDump.SetDepth( 1 );
   afxDump << "SetAtGrow example: " << &array << "\n";
#endif
```

The results from this program are as follows:

```
SetAtGrow example: A CObArray with 4 elements
    [0] = a CAge at $47C0 21
    [1] = a CAge at $4800 40
    [2] = NULL
    [3] = a CAge at $4840 65
```

CObArray::SetSize

void SetSize(int *nNewSize*, **int** *nGrowBy* = **–1)**
throw(CMemoryException);

nNewSize The new array size (number of elements). Must be greater than or equal to 0.

nGrowBy The minimum number of element slots to allocate if a size increase is necessary.

Remarks

Establishes the size of an empty or existing array; allocates memory if necessary. If the new size is smaller than the old size, then the array is truncated and all unused memory is released. The *nGrowBy* parameter affects internal memory allocation while the array is growing. Its use never affects the array size as reported by **GetSize** and **GetUpperBound**.

Operators

CObArray::operator [ ]

CObject*& operator [ ](int *nIndex* **);**

CObject* operator [ ](int *nIndex* **) const;**

Remarks

These subscript operators are a convenient substitute for the **SetAt** and **GetAt** functions. The first operator, invoked for arrays that are not **const**, may be used on either the right (r-value) or the left (l-value) of an assignment statement. The second, invoked for **const** arrays, may be used only on the right. The Debug version of the library asserts if the subscript (either on the left or right side of an assignment statement) is out of bounds.

See Also

CObArray::GetAt, **CObArray::SetAt**

Example

```
CObArray array;
CAge* pa;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
pa = (CAge*)array[0]; // Get element 0
ASSERT( *pa == CAge( 21 ) ); // Get element 0
array[0] = new CAge( 30 ); // Replace element 0
delete pa;
ASSERT( *(CAge*) array[0] == CAge( 30 ) ); // Get new element 0
```

class CObject

CObject is the principal base class for the Microsoft Foundation Class Library. It serves as the root not only for library classes such as **CFile** and **CObList**, but also for the classes that you write. **CObject** provides basic services, including:

- Serialization support
- Run-time class information
- Object diagnostic output
- Compatibility with collection classes

For a detailed description of these features, see Chapters 12 through 15 of the *Class Library User's Guide.*

Note that **CObject** does not support multiple inheritance. Your derived classes can have only one **CObject** base class, and that **CObject** must be leftmost in the hierarchy. It is permissible, though, to have structures and non-**CObject**-derived classes in right-hand multiple-inheritance branches.

You will realize major benefits from **CObject** derivation if you use some of the optional macros in your class implementation and declarations. The **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros permit run-time access to the class name and its position in the hierarchy. This, in turn, allows meaningful diagnostic dumping. The **DECLARE-DYNCREATE** and **IMPLEMENT-DYNCREATE** macros permit you to create an object of a specific class at run time. The **DECLARE_SERIAL** and **IMPLEMENT_SERIAL** macros include all the functionality of the previously discussed macros, and they enable an object to be "serialized" to and from an "archive."

For important information about deriving Microsoft Foundation classes and Visual C++ classes in general, see "Deriving a Class from CObject" in Chapter 12 of the *Class Library User's Guide.*

#include <afx.h>

Construction/Destruction—Public Members

~CObject	Virtual destructor.
operator new	Special **new** operator.
operator delete	Special **delete** operator.

Diagnostics—Public Members

AssertValid	Validates this object's integrity.
Dump	Produces a diagnostic dump of this object.

Serialization — Public Members

IsSerializable	Tests to see if this object can be serialized.
Serialize	Loads or stores an object from/to an archive.

Miscellaneous — Public Members

GetRuntimeClass	Returns the **CRuntimeClass** structure corresponding to this object's class.
IsKindOf	Tests this object's relationship to a given class.

Construction/Destruction — Protected Members

CObject	Default constructor.

Private Members

CObject	Copy constructor.
operator =	Assignment operator.

Member Functions

CObject::AssertValid

virtual void AssertValid( ) const;

Remarks

AssertValid performs a validity check on this object by checking its internal state. In the Debug version of the library, **AssertValid** may assert and thus terminate the program with a message that lists the line number and filename where the assertion failed. When you write your own class, you should override the **AssertValid** function to provide diagnostic services for yourself and other users of your class. The overridden **AssertValid** usually calls the **AssertValid** function of its base class before checking data members unique to the derived class.

Because **AssertValid** is a **const** function, you are not permitted to change the object state during the test. Your own derived class **AssertValid** functions should not throw exceptions but rather should assert if they detect invalid object data. The definition of "validity" depends on the object's class. As a rule, the function should perform a "shallow check." That is, if an object contains pointers to other objects, it should check to see if the pointers are not null but should not perform validity testing on the objects referred to by the pointers.

Example

See **CObList::CObList** for a listing of the `CAge` class used in all **CObject** examples.

```
void CAge::AssertValid() const
{
    CObject::AssertValid();
    ASSERT( m_years > 0 );
    ASSERT( m_years < 105 );
}
```

CObject::CObject

Protected

CObject( ); ♦

Private

CObject(constCObject& *objectSrc* **);** ♦

objectSrc A reference to another **CObject**.

Remarks

These functions are the standard **CObject** constructors. The default version is automatically called by the constructor of your derived class. If your class is serializable (it incorporates the **IMPLEMENT_SERIAL** macro), then you must have a default constructor (a constructor with no arguments) in your class declaration. If you don't need a default constructor, declare a private or protected "empty" constructor. For more information, see "Deriving a Class from CObject" in Chapter 12 of the *Class Library User's Guide*. The standard Visual C++ default class copy constructor does a member-by-member copy. The presence of the private **CObject** copy constructor guarantees a compiler error message if the copy constructor of your class is needed but not available. You must, therefore, provide a copy constructor if your class requires this capability.

CObject::~CObject

virtual ~CObject( );

Remarks

This function is the standard **CObject** destructor. If your derived class must free allocated memory or do other cleanup work, you must provide your own destructor. Because **~CObject** is a virtual destructor, Visual C++ ensures that **CObject::~CObject** is automatically called as part of the destructor of your class.

Note Your destructor should not throw exceptions or allocate objects.

CObject::Dump

virtual void Dump(CDumpContext& *dc* **) const;**

dc The diagnostic dump context for dumping, usually **afxDump**.

Remarks

Dumps the contents of your object to a **CDumpContext** object. When you write your own class, you should override the **Dump** function to provide diagnostic services for yourself and other users of your class. The overridden **Dump** usually calls the **Dump** function of its base class before printing data members unique to the derived class. **CObject::Dump** prints the class name if your class uses the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macro.

Note Your **Dump** function should not print a newline character at the end of its output.

Dump calls make sense only in the Debug version of the Microsoft Foundation Class Library. Bracket calls, function declarations, and function implementations with **#ifdef _DEBUG/#endif** statements for conditional compilation. Since **Dump** is a **const** function, you are not permitted to change the object state during the dump. The **CDumpContext** insertion (<<) operator calls **Dump** when a **CObject** pointer is inserted. **Dump** permits only "acyclic" dumping of objects. You can dump a list of objects, for example, but if one of the objects is the list itself, you will eventually overflow the stack.

Example

```
void CAge::Dump( CDumpContext &dc ) const
 {
 CObject::Dump( dc );
 dc << "Age = " << m_years;
 }
```

CObject::GetRuntimeClass

virtual CRuntimeClass* GetRuntimeClass( ) const;

Remarks

There is one **CRuntimeClass** structure for each **CObject**-derived class. The structure members are as follows:

- **const char* m_pszClassName** A null-terminated string containing the ASCII class name.
- **int m_nObjectSize** The actual size of the object. If the object has data members that point to allocated memory, the size of that memory is not included.

- **WORD m_wSchema** The schema number (–1 for nonserializable classes). See the **IMPLEMENT_SERIAL** macro for a description of schema number.
- **void (*m_pfnConstruct)(void* p)** A pointer to the default constructor of your class (valid only if the class is serializable).
- **CRuntimeClass* m_pBaseClass** A pointer to the **CRuntimeClass** structure that corresponds to the base class.

This function requires use of the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macros in the class implementation. You will get incorrect results otherwise.

Return Value

A pointer to the **CRuntimeClass** structure corresponding to this object's class; never **NULL**.

See Also

CObject::IsKindOf, RUNTIME_CLASS Macro

Example

```
CAge a(21);
CRuntimeClass* prt = a.GetRuntimeClass();
ASSERT( strcmp( prt->m_pszClassName, "CAge" )  == 0 );
```

CObject::IsKindOf

BOOL IsKindOf(const CRuntimeClass* *pClass* **) const;**

pClass A pointer to a **CRuntimeClass** structure associated with your **CObject**-derived class.

Remarks

Tests *pClass* to see if (1) it is an object of the specified class or (2) it is an object of a class derived from the specified class. This function only works for classes declared with the **DECLARE_DYNAMIC** or **DECLARE_SERIAL** macros. Do not use this function extensively because it defeats the Visual C++ polymorphism feature. Use virtual functions instead.

Return Value

TRUE if the object corresponds to the class; otherwise **FALSE**.

See Also

CObject::GetRuntimeClass, RUNTIME_CLASS Macro

Example

```
CAge a(21); // Must use IMPLEMENT_DYNAMIC or IMPLEMENT_SERIAL
ASSERT( a.IsKindOf( RUNTIME_CLASS( CAge ) ) );
ASSERT( a.IsKindOf( RUNTIME_CLASS( CObject ) ) );
```

CObject::IsSerializable

BOOL IsSerializable( ) const;

Remarks

Tests whether this object is eligible for serialization. For a class to be serializable, its declaration must contain the **DECLARE_SERIAL** macro, and the implementation must contain the **IMPLEMENT_SERIAL** macro.

Note Do not override this function.

Return Value

TRUE if this object can be serialized; otherwise **FALSE**.

See Also

CObject::Serialize

Example

```
CAge a(21);
ASSERT( a.IsSerializable() );
```

CObject::Serialize

virtual void Serialize(CArchive& *ar* **)**
throw(CMemoryException, CArchiveException, CFileException);

ar A **CArchive** object to serialize to or from.

Remarks

Reads or writes this object from or to an archive. You must override **Serialize** for each class that you intend to serialize. The overridden **Serialize** must first call the **Serialize** function of its base class. You must also use the **DECLARE_SERIAL** macro in your class declaration, and you must use the **IMPLEMENT_SERIAL** macro in the implementation.

Use **CArchive::IsLoading** or **CArchive::IsStoring** to determine whether the archive is loading or storing. **Serialize** is called by **CArchive::ReadObject** and **CArchive::WriteObject**. These functions are associated with the **CArchive** insertion operator (<<) and extraction operator (>>). For serialization examples, refer to Chapters 3 and 14 in the *Class Library User's Guide*.

Example

```
void CAge::Serialize( CArchive& ar )
 {
 CObject::Serialize( ar );
     if( ar.IsStoring() )
     ar << m_years;
     else
     ar >> m_years;
 }
```

Operators

CObject::operator =

Private **void operator =(const CObject&** *src* **);** ♦

Remarks The standard Visual C++ default class assignment behavior is a member-by-member copy. The presence of this private assignment operator guarantees a compiler error message if you assign without the overridden operator. You must, therefore, provide an assignment operator in your derived class if you intend to assign objects of your derived class.

CObject::operator delete

void operator delete(void* *p* **);**

Remarks For the Release version of the library, operator **delete** simply frees the memory allocated by operator **new**. In the Debug version, operator **delete** participates in an allocation-monitoring scheme designed to detect memory leaks. If you override operators **new** and **delete**, you forfeit the diagnostic capability.

See Also **CObject::operator new**

CObject::operator new

void* operator new(size_t *nSize* **)**
throw(CMemoryException);

void* operator new(size_t *nSize*, **const char FAR*** *lpszFileName*, **int** *nLine* **)**
throw(CMemoryException);

Remarks For the Release version of the library, operator **new** performs an optimal memory allocation in a manner similar to **malloc**. In the Debug version, operator **new** participates in an allocation-monitoring scheme designed to detect memory leaks.

If you use the code line

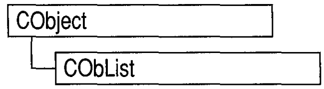
```
#define new DEBUG_NEW
```

before any of your implementations in a .CPP file, then the second version of **new** will be used, storing the filename and line number in the allocated block for later reporting. You do not have to worry about supplying the extra parameters; a macro takes care of that for you. Even if you don't use **DEBUG_NEW** in Debug mode, you still get leak detection but without the source-file line-number reporting described above.

Note If you override this operator, you must also override **delete**. Do not use the standard library **_new_handler** function.

See Also **CObject::operator delete**

class CObList : public CObject



The **CObList** class supports ordered lists of nonunique **CObject** pointers accessible sequentially or by pointer value. **CObList** lists behave like doubly-linked lists. A variable of type **POSITION** is a key for the list. You can use a **POSITION** variable as an iterator to sequentially traverse a list and as a bookmark to hold a place. A position is not the same as an index, however. Element insertion is very fast at the list head, at the tail, and at a known **POSITION**. A sequential search is necessary to look up an element by value or index. This search can be slow if the list is long.

CObList incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If a list of **CObject** pointers is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each **CObject** element is, in turn, serialized.

If you need a dump of individual **CObject** elements in the list, you must set the depth of the dump context to 1 or greater. When a **CObList** object is deleted, or when its elements are removed, only the **CObject** pointers are removed, not the objects they reference.

You can derive your own classes from **CObList**. Your new list class, designed to hold pointers to objects derived from **CObject**, adds new data members and new member functions. Note that the resulting list is not strictly type safe because it allows insertion of any **CObject** pointer.

Note You must use the **IMPLEMENT_SERIAL** macro in the implementation of your derived class if you intend to serialize the list.

#include <afxcoll.h>

See Also **CStringList**, **CPtrList**

Construction/Destruction—Public Members

CObList	Constructs an empty list for **CObject** pointers.

Head/Tail Access—Public Members

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations—Public Members

RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.
AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.

Iteration—Public Members

GetHeadPosition	Returns the position of the head element of the list.
GetTailPosition	Returns the position of the tail element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.

Retrieval/Modification—Public Members

GetAt	Gets the element at a given position.
SetAt	Sets the element at a given position.
RemoveAt	Removes an element from this list, specified by position.

Insertion—Public Members

InsertBefore	Inserts a new element before a given position.
InsertAfter	Inserts a new element after a given position.

Searching—Public Members

Find	Gets the position of an element specified by pointer value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status—Public Members

GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

Member Functions

CObList::AddHead

POSITION AddHead(CObject* *newElement* **)**
throw(CMemoryException);

void AddHead(CObList* *pNewList* **)**
throw(CMemoryException);

newElement The **CObject** pointer to be added to this list.

pNewList A pointer to another **CObList** list. The elements in *pNewList* will be added to this list.

Remarks

Adds a new element or list of elements to the head of this list. The list may be empty before the operation.

Return Value

The first version returns the **POSITION** value of the newly inserted element.

See Also

CObList::GetHead, **CObList::RemoveHead**

Example

```
    CObList list;
    list.AddHead( new CAge( 21 ) ); // 21 is now at head.
    list.AddHead( new CAge( 40 ) ); // 40 replaces 21 at head.
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "AddHead example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
AddHead example: A CObList with 2 elements
    a CAge at $44A8 40
    a CAge at $442A 21
```

CObList::AddTail

POSITION AddTail(CObject* *newElement* **)**
throw(CMemoryException);

void AddTail(CObList* *pNewList* **)**
throw(CMemoryException);

newElement The **CObject** pointer to be added to this list.

pNewList A pointer to another **CObList** list. The elements in *pNewList* will be added to this list.

Remarks Adds a new element or list of elements to the tail of this list. The list may be empty before the operation.

Return Value The first version returns the **POSITION** value of the newly inserted element.

See Also **CObList::GetTail**, **CObList::RemoveTail**

Example

```
    CObList list;
    list.AddTail( new CAge( 21 ) );
    list.AddTail( new CAge( 40 ) ); // List now contains (21, 40).
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "AddTail example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
AddTail example: A CObList with 2 elements
     a CAge at $444A 21
     a CAge at $4526 40
```

CObList::CObList

CObList(int *nBlockSize* **= 10);**

nBlockSize The memory-allocation granularity for extending the list.

Remarks Constructs an empty **CObject** pointer list. As the list grows, memory is allocated in units of *nBlockSize* entries. If a memory allocation fails, a **CMemoryException** is thrown.

Example

Below is a listing of the **CObject**-derived class `CAge` used in all the collection examples:

```
// Simple CObject-derived class for CObList examples
class CAge : public CObject
{
    DECLARE_SERIAL( CAge )
private:
    int m_years;
public:
    CAge() { m_years = 0; }
    CAge( int age ) { m_years = age; }
    CAge( const CAge& a ) { m_years = a.m_years; } // Copy constructor
    void Serialize( CArchive& ar);
    void AssertValid() const;
    const CAge& operator=( const CAge& a )
    {
        m_years = a.m_years; return *this;
    }
    BOOL operator==(CAge a)
    {
        return m_years == a.m_years;
    }
 #ifdef _DEBUG
    void Dump( CDumpContext& dc ) const
    {
        CObject::Dump( dc );
        dc << m_years;
    }
 #endif
};
```

Below is an example of **CObList** constructor usage:

```
CObList list( 20 );  // List on the stack with blocksize = 20.

CObList* plist = new CObList; // List on the heap with default
                              // blocksize.
```

CObList::Find

POSITION Find(CObject* *searchValue*, **POSITION** *startAfter* = **NULL)**
const;

searchValue The object pointer to be found in this list.

startAfter The start position for the search.

Remarks

Searches the list sequentially to find the first **CObject** pointer matching the specified **CObject** pointer. Note that the pointer values are compared, not the contents of the objects.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the object is not found.

See Also

CObList::GetNext, **CObList::GetPrev**

Example

```
CObList list;
CAge* pa1;
CAge* pa2;
POSITION pos;
list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) );    // List now contains (40, 21).
if( ( pos = list.Find( pa1 ) ) != NULL ) // Hunt for pa1
{                                        // starting at head by default.
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 21 ) );
}
```

CObList::FindIndex

POSITION FindIndex(int *nIndex* **) const;**

nIndex The zero-based index of the list element to be found.

Remarks

Uses the value of *nIndex* as an index into the list. It starts a sequential scan from the head of the list, stopping on the nth element.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if *nIndex* is negative or too large.

See Also

CObList::Find, **CObList::GetNext**, **CObList::GetPrev**

Example

```
CObList list;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
if( ( pos = list.FindIndex( 0 )) != NULL )
{
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 40 ) );
}
```

CObList::GetAt

CObject*& GetAt(POSITION *position* **);**

CObject* GetAt(POSITION *position* **) const;**

position A **POSITION** value returned by a previous **GetHeadPosition** or **Find** member function call.

Remarks

A variable of type **POSITION** is a key for the list. It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **GetAt** retrieves the **CObject** pointer associated with a given position. You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

Return Value

See the return value description for **GetHead**.

See Also

CObList::Find, **CObList::SetAt**, **CObList::GetNext**, **CObList::GetPrev**, **CObList::GetHead**

Example

See the example for **FindIndex**.

CObList::GetCount

int GetCount() const;

Remarks

Gets the number of elements in this list.

Return Value

An integer value containing the element count.

See Also

CObList::IsEmpty

Example

```
CObList list;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( list.GetCount() == 2 );
```

CObList::GetHead

CObject*& GetHead( );

CObject* GetHead( ) const;

Remarks

Gets the **CObject** pointer that represents the head element of this list. You must ensure that the list is not empty before calling **GetHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use **IsEmpty** to verify that the list contains elements.

Return Value

If the list is accessed through a pointer to a **const CObList**, then **GetHead** returns a **CObject** pointer. This allows the function to be used only on the right side of an assignment statement and thus protects the list from modification. If the list is accessed directly or through a pointer to a **CObList**, then **GetHead** returns a reference to a **CObject** pointer. This allows the function to be used on either side of an assignment statement and thus allows the list entries to be modified.

See Also

CObList::GetTail, **CObList::GetTailPosition**, **CObList::AddHead**, **CObList::RemoveHead**

Example

The following example illustrates the use of **GetHead** on the left side of an assignment statement.

```
   const CObList* cplist;

   CObList* plist = new CObList;
   CAge* page1 = new CAge( 21 );
   CAge* page2 = new CAge( 30 );
   CAge* page3 = new CAge( 40 );
   plist->AddHead( page1 );
   plist->AddHead( page2 );  // List now contains (30, 21).
   // The following statement REPLACES the head element.
   plist->GetHead() = page3; // List now contains (40, 21).
   ASSERT( *(CAge*) plist->GetHead() == CAge( 40 ) );
   cplist = plist;  // cplist is a pointer to a const list.
// cplist->GetHead() = page3; // Does not compile!
   ASSERT( *(CAge*) plist->GetHead() == CAge( 40 ) ); // OK

   delete page1;
   delete page2;
   delete page3;
   delete plist; // Cleans up memory.
```

CObList::GetHeadPosition

POSITION GetHeadPosition( ) const;

Remarks

Gets the position of the head element of this list.

Return Value

A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

See Also

CObList::GetTailPosition

Example

```
CObList list;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
if( ( pos = list.GetHeadPosition() ) != NULL )
{
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 40 ) );
}
```

CObList::GetNext

CObject*& GetNext(POSITION& *rPosition* **);**

CObject* GetNext(POSITION& *rPosition* **) const;**

rPosition A reference to a **POSITION** value returned by a previous **GetNext**, **GetHeadPosition**, or other member function call.

Remarks

Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the next entry in the list. You can use **GetNext** in a forward iteration loop if you establish the initial position with a call to **GetHeadPosition** or **Find**.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

If the retrieved element is the last in the list, then the new value of *rPosition* is set to **NULL**. It is possible to remove an element during an iteration. See the example for **RemoveAt**.

Return Value

See the return value description for **GetHead**.

See Also **CObList::Find**, **CObList::GetHeadPosition**, **CObList::GetTailPosition**, **CObList::GetPrev**, **CObList::GetHead**

Example

```
    CObList list;
    POSITION pos;
    list.AddHead( new CAge( 21 ) );
    list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
    // Iterate through the list in head-to-tail order.
#ifdef _DEBUG
    for( pos = list.GetHeadPosition(); pos != NULL; )
    {
    afxDump << list.GetNext( pos ) << "\n";
    }
#endif
```

The results from this program are as follows:

```
a CAge at $479C 40
a CAge at $46C0 21
```

CObList::GetPrev

CObject*& GetPrev(POSITION& *rPosition* **);**

CObject* GetPrev(POSITION& *rPosition* **) const;**

rPosition A reference to a **POSITION** value returned by a previous **GetPrev** or other member function call.

Remarks Gets the list element identified by *rPosition*, then sets *rPosition* to the **POSITION** value of the previous entry in the list. You can use **GetPrev** in a reverse iteration loop if you establish the initial position with a call to **GetTailPosition** or **Find**.

You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts. If the retrieved element is the first in the list, then the new value of *rPosition* is set to **NULL**.

Return Value See the return value description for **GetHead**.

See Also **CObList::Find**, **CObList::GetTailPosition**, **CObList::GetHeadPosition**, **CObList::GetNext**, **CObList::GetHead**

Example

```
    CObList list;
    POSITION pos;

    list.AddHead( new CAge(21) );
    list.AddHead( new CAge(40) ); // List now contains (40, 21).
    // Iterate through the list in tail-to-head order.
    for( pos = list.GetTailPosition(); pos != NULL; )
    {
#ifdef _DEBUG
    afxDump << list.GetPrev( pos ) << "\n";
#endif
    }
```

The results from this program are as follows:

```
a CAge at $421C 21
a CAge at $421C 40
```

CObList::GetTail

CObject*& GetTail( );

CObject* GetTail( ) const;

Remarks Gets the **CObject** pointer that represents the tail element of this list. You must ensure that the list is not empty before calling **GetTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use **IsEmpty** to verify that the list contains elements.

Return Value See the return value description for **GetHead**.

See Also **CObList::AddTail**, **CObList::AddHead**, **CObList::RemoveHead**, **CObList::GetHead**

Example

```
CObList list;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( *(CAge*) list.GetTail() == CAge( 21 ) );
```

CObList::GetTailPosition

POSITION GetTailPosition( ) const;

Remarks Gets the position of the tail element of this list; **NULL** if the list is empty.

Return Value A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

See Also **CObList::GetHeadPosition**, **CObList::GetTail**

Example

```
CObList list;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
if( ( pos = list.GetTailPosition() ) != NULL )
{
    ASSERT( *(CAge*) list.GetAt( pos ) == CAge( 21 ) );
}
```

CObList::InsertAfter

POSITION InsertAfter(POSITION *position*, **CObject*** *newElement* **)**
throw (CMemoryException);

position A **POSITION** value returned by a previous **GetNext**, **GetPrev**, or **Find** member function call.

newElement The object pointer to be added to this list.

Remarks Adds an element to this list after the element at the specified position.

See Also **CObList::Find**, **CObList::InsertBefore**

Example

```
    CObList list;
    POSITION pos1, pos2;
    list.AddHead( new CAge( 21 ) );
    list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
    if( ( pos1 = list.GetHeadPosition() ) != NULL )
    {
        pos2 = list.InsertAfter( pos1, new CAge( 65 ) );
    }
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "InsertAfter example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
InsertAfter example: A CObList with 3 elements
     a CAge at $4A44 40
     a CAge at $4A64 65
     a CAge at $4968 21
```

CObList::InsertBefore

POSITION InsertBefore(POSITION *position*, **CObject*** *newElement*)
throw (CMemoryException);

position A **POSITION** value returned by a previous **GetNext**, **GetPrev**, or **Find** member function call.

newElement The object pointer to be added to this list.

Remarks Adds an element to this list before the element at the specified position.

Return Value A **POSITION** value that can be used for iteration or object pointer retrieval; **NULL** if the list is empty.

See Also **CObList::Find**, **CObList::InsertAfter**

Example

```
    CObList list;
    POSITION pos1, pos2;
    list.AddHead( new CAge( 21 ) );
    list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
    if( ( pos1 = list.GetTailPosition() ) != NULL )
    {
        pos2 = list.InsertBefore( pos1, new CAge( 65 ) );
    }
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "InsertBefore example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
InsertBefore example: A CObList with 3 elements
    a CAge at $4AE2 40
    a CAge at $4B02 65
    a CAge at $49E6 21
```

CObList::IsEmpty

BOOL IsEmpty( ) const;

Remarks Indicates if this list contains no elements.

Return Value **TRUE** if this list is empty; otherwise **FALSE**.

See Also **CObList::GetCount**

Example See the example for **RemoveAll**.

CObList::RemoveAll

void RemoveAll( );

Remarks Removes all the elements from this list and frees the associated **CObList** memory. No error is generated if the list is already empty. When you remove elements from a **CObList**, you remove the object pointers from the list. It is your responsibility to delete the objects themselves.

Example

```
CObList list;
CAge* pa1;
CAge* pa2;
ASSERT( list.IsEmpty()); // Yes it is.
list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( !list.IsEmpty()); // No it isn't.
list.RemoveAll(); // CAge's aren't destroyed.
ASSERT( list.IsEmpty()); // Yes it is.
delete pa1;     // Now delete the CAge objects.
delete pa2;
```

CObList::RemoveAt

void RemoveAt(POSITION *position* **);**

position The position of the element to be removed from the list.

Remarks

Removes the specified element from this list. When you remove an element from a **CObList**, you remove the object pointer from the list. It is your responsibility to delete the objects themselves. You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

Example

Be careful when removing an element during a list iteration. The following example shows a removal technique that guarantees a valid **POSITION** value for **GetNext**:

```
CObList list;
POSITION pos1, pos2;
CObject* pa;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) );
list.AddHead( new CAge( 65 ) ); // List now contains (65 40, 21).
for( pos1 = list.GetHeadPosition(); ( pos2 = pos1 ) != NULL; )
{
    if( *(CAge*) list.GetNext( pos1 ) == CAge( 40 ) )
    {
        pa = list.GetAt( pos2 ); // Save the old pointer for
                                 //deletion.
        list.RemoveAt( pos2 );
        delete pa; // Deletion avoids memory leak.
    }
}
```

```
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "RemoveAt example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
RemoveAt example: A CObList with 2 elements
    a CAge at $4C1E 65
    a CAge at $4B22 21
```

CObList::RemoveHead

CObject* RemoveHead( );

Remarks

Removes the element from the head of the list and returns a pointer to it. You must ensure that the list is not empty before calling **RemoveHead**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use **IsEmpty** to verify that the list contains elements.

Return Value

The **CObject** pointer previously at the head of the list.

See Also

CObList::GetHead, **CObList::AddHead**

Example

```
CObList list;
CAge* pa1;
CAge* pa2;

list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( *(CAge*) list.RemoveHead() == CAge( 40 ) );  // Old head
ASSERT( *(CAge*) list.GetHead() == CAge( 21 ) );  // New head
delete pa1;
delete pa2;
```

CObList::RemoveTail

CObject* RemoveTail( );

Remarks

Removes the element from the tail of the list and returns a pointer to it. You must ensure that the list is not empty before calling **RemoveTail**. If the list is empty, then the Debug version of the Microsoft Foundation Class Library asserts. Use **IsEmpty** to verify that the list contains elements.

Return Value A pointer to the object that was at the tail of the list.

See Also **CObList::GetTail, CObList::AddTail**

Example

```
CObList list;
CAge* pa1;
CAge* pa2;

list.AddHead( pa1 = new CAge( 21 ) );
list.AddHead( pa2 = new CAge( 40 ) ); // List now contains (40, 21).
ASSERT( *(CAge*) list.RemoveTail() == CAge( 21 ) );  // Old tail
ASSERT( *(CAge*) list.GetTail() == CAge( 40 ) );  // New tail
delete pa1;
delete pa2; // Clean up memory.
```

CObList::SetAt

void SetAt(POSITION *pos*, **CObject*** *newElement* **);**

pos The **POSITION** of the element to be set.

newElement The **CObject** pointer to be written to the list.

Remarks A variable of type **POSITION** is a key for the list. It is not the same as an index, and you cannot operate on a **POSITION** value yourself. **SetAt** writes the **CObject** pointer to the specified position in the list. You must ensure that your **POSITION** value represents a valid position in the list. If it is invalid, then the Debug version of the Microsoft Foundation Class Library asserts.

See Also **CObList::Find**, **CObList::GetAt**, **CObList::GetNext**, **CObList::GetPrev**

Example

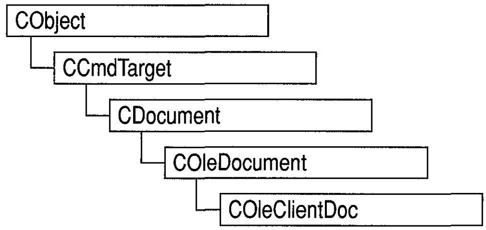
```
CObList list;
CObject* pa;
POSITION pos;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21).
```

```
    if( ( pos = list.GetTailPosition()) != NULL )
    {
        pa = list.GetAt( pos ); // Save the old pointer for
                                //deletion.
        list.SetAt( pos, new CAge( 65 ) );  // Replace the tail
                                            //element.
        delete pa;  // Deletion avoids memory leak.
    }
#ifdef _DEBUG
    afxDump.SetDepth( 1 );
    afxDump << "SetAt example: " << &list << "\n";
#endif
```

The results from this program are as follows:

```
SetAt example: A CObList with 2 elements
    a CAge at $4D98 40
    a CAge at $4DB8 65
```

class COleClientDoc : public COleDocument



COleClientDoc is the base class for Object Linking and Embedding (OLE) client documents. A client document can contain **COleClientItem** objects as well as any data created by the client application itself. The **COleClientItem** objects represent embedded items, which contain data created by other applications (servers), or linked items, which contain links to files created by servers.

To use **COleClientDoc**, derive a class from it and design a data structure for storing the application's native data as well as embedded or linked items. If you use **CDocItem**-derived classes to store the application's native data, you can use the interface defined by **COleDocument** to manipulate a document as a collection of items. This allows you to treat the application's native data in the same way you treat embedded or linked items.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **COleDocument**, **COleClientItem**

Construction/Destruction—Public Members

COleClientDoc	Constructs a **COleClientDoc** object.

Registration/Revocation—Public Members

RegisterClientDoc	Registers a client document with the OLE system dynamic-link library (DLL).
Revoke	Revokes the client document registration.

Operations — Public Members

GetPrimarySelectedItem	Returns primary selected item in the document.
NotifyRename	Notifies the OLE system DLL that the client document has been renamed.
NotifyRevert	Notifies the OLE system DLL that the client document has reverted to its previous state.
NotifySaved	Notifies the OLE system DLL that the client document has been saved.

Member Functions

COleClientDoc::COleClientDoc

COleClientDoc( );

Remarks

Creates a **COleClientDoc** object. It does not register the document with the OLE system DLL. You must call the **RegisterClientDoc** member function before you can create embedded or linked items.

See Also

COleClientDoc::RegisterClientDoc

COleClientDoc::GetPrimarySelectedItem

virtual COleClientItem* GetPrimarySelectedItem(CView* *pView* **);**

pView A pointer to the active view object displaying the document.

Remarks

Call this function to get the currently selected OLE item in the specified view. If one and only one **COleClientItem** object is selected, the function returns a pointer to it; otherwise the function returns **NULL**. You must implement the **IsSelected** member function in your view class for this function to work.

Return Value

A pointer to the single, selected OLE item; **NULL** if there are no OLE items selected or if there are more than one selected.

See Also

CView::IsSelected

COleClientDoc::NotifyRename

void NotifyRename(LPCSTR *lpszNewName* **);**

lpszNewName Pointer to the new name of the document. Must be a valid filename.

Remarks

Call this function after the user renames the client document. In the case where the user chooses the Save As command from the File menu, **NotifyRename** is called for you by **COleClientDoc**'s implementation of the **OnSaveDocument** member function. This function notifies the OLE system DLL.

See Also

COleClientDoc::NotifyRevert, **COleClientDoc::NotifySaved**, **CDocument::OnSaveDocument**, **::OleRenameClientDoc**

COleClientDoc::NotifyRevert

void NotifyRevert( );

Remarks

Call this function after the user reverts the client document, that is, reloads it without saving changes. This function notifies the OLE system DLL.

See Also

COleClientDoc::NotifyRename, **COleClientDoc::NotifySaved**, **::OleRevertClientDoc**

COleClientDoc::NotifySaved

void NotifySaved( );

Remarks

Call this function after the user saves the client document. In the case where the user chooses the Save command from the File menu, **NotifySaved** is called for you by **COleClientDoc**'s implementation of **OnSaveDocument.** This function notifies the OLE system DLL.

See Also

COleClientDoc::NotifyRename, **COleClientDoc::NotifyRevert**, **CDocument::OnSaveDocument**, **::OleSavedClientDoc**

COleClientDoc::RegisterClientDoc

BOOL RegisterClientDoc(LPCSTR *lpszTypeName*, **LPCSTR** *lpszDoc* **);**

lpszTypeName Pointer to the name of the client document's type, usually the client application name.

lpszDoc Pointer to the fully qualified name of the client document.

Remarks

Call this function to register your client document with the OLE system DLL; this allows the client to interact with server applications. When the user chooses the File New or File Open commands, **RegisterClientDoc** is called for you by **COleClientDoc**'s implementation of **OnNewDocument** or **OnOpenDocument**, respectively.

When a document being copied onto the Clipboard exists only because the client application is copying Native data that contains objects, the name specified in the *lpszDoc* parameter must be "Clipboard."

Return Value

Nonzero if the document was successfully registered with the OLE system DLL; otherwise 0.

See Also

COleClientDoc::COleClientDoc, **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**, **::OleRegisterClientDoc**

COleClientDoc::Revoke

void Revoke( );

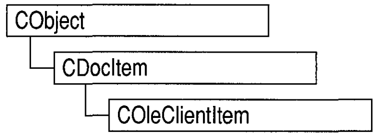
Remarks

Call this function to revoke a client document, that is, inform the OLE system DLL that the document is closed. This function is called by the **COleClientDoc** destructor, so you rarely need to call it explicitly. **Revoke** may be called for an already revoked document with no ill effects. Before you call **Revoke**, you must delete or call **COleClientItem::Release** or **COleClientItem::Delete** for each item in the document.

See Also

COleClientItem::Release, **COleClientItem::Delete**, **::OleRevokeClientDoc**

class COleClientItem : public CDocItem



The **COleClientItem** class defines the client interface to Object Linking and Embedding (OLE) items. An OLE item represents data incorporated into a client application's document but created by a server application; a document containing OLE items is called a "compound document."

An item can be either embedded or linked. If it is embedded, its data is stored in the compound document. If it is linked, its data is stored as part of a separate file created by the server application and only a link to that file is stored in the compound document. All items contain information specifying the server application that should be invoked to edit them.

COleClientItem defines several overridable functions that are called indirectly by the OLE system dynamic-link library (DLL), usually in response to notifications from the server application. This allows the server application to inform the client of changes that the user makes when editing the item.

To use **COleClientItem**, derive a class from it and implement the **OnChange** member function. This function defines how the client responds to changes made to the item.

Each item must be given a name that is unique within the document. An item's name must be preserved when the document is saved and cannot contain the "/" or "\" characters.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **CDocItem**, **COleClientDoc**, **COleServerItem**

Construction/Destruction—Public Members

COleClientItem	Constructs a **COleClientItem** object.

Creation — Public Members

CreateFromClipboard	Creates an embedded item from the Clipboard.
CreateInvisibleObject	Creates an invisible embedded item.
CreateStaticFromClipboard	Creates an embedded picture of an item from the Clipboard.
CreateLinkFromClipboard	Creates a linked item from the Clipboard.
CreateNewObject	Creates a new embedded item by launching the server application.
CreateCloneFrom	Creates a duplicate of an existing item.

Status — Public Members

GetLastStatus	Returns the status of the last OLE operation.
GetType	Returns the type (embedded, linked, or static) of the item.
GetName	Returns the name of the item.
GetSize	Returns the size of the item.
GetBounds	Returns the bounds of the item's rectangle.
IsOpen	Indicates whether the item is currently attached to the OLE system DLL.

Data Access — Public Members

EnumFormats	Enumerates the Clipboard formats supported by an item.
GetData	Gets data from an item in a specified format.
SetData	Stores data to an item in a specified format.
RequestData	Initiates a data request from a server.
IsEqual	Compares two items.
GetDocument	Returns the **COleClientDoc** object that contains this item.

Global State — Public Members

InWaitForRelease	Indicates whether any item is still waiting for a server to respond.

Clipboard Helpers—Public Members

CanPaste	Indicates whether the Clipboard contains an embeddable or static OLE item.
CanPasteLink	Indicates whether the Clipboard contains a linkable OLE item.

Linked Object Status—Public Members

GetLinkUpdateOptions	Returns the update mode for a linked item (advanced feature).
SetLinkUpdateOptions	Sets the update mode for a linked item (advanced feature).

General Operations—Public Members

Release	Releases the connection to an OLE linked item and closes it if it was open. Does not destroy the server item.
Delete	Deletes the item or closes it if it was a linked item.
Draw	Draws the item.
DoVerb	Executes the specified verb.
Activate	Opens the item for an operation, then executes the specified verb.

Advanced Operations—Public Members

Rename	Renames the item.
CopyToClipboard	Copies the item to the Clipboard.
SetTargetDevice	Sets the target device used by the server to draw the item.

Embedded Object Operations—Public Members

SetHostNames	Sets the names the server displays when editing the item.
SetBounds	Sets the bounding rectangle of the item.
SetColorScheme	Sets the item's color scheme.

Linked Object Operations — Public Members

UpdateLink	Updates a link to a server.
CloseLink	Closes a link to a server but does not destroy the item.
ReconnectLink	Reconnects a linked item to a server.

Overridables — Protected Members

OnChange	Called when the server changes the item. Implementation required.
OnRenamed	Called when the server renames a document containing the item.

Member Functions

COleClientItem::Activate

void Activate(UINT *nVerb*, **BOOL** *bShow* = **TRUE,**
BOOL *bTakeFocus* = **TRUE, CWnd*** *pWndContainer* = **NULL,**
LPCRECT *lpBounds* = **NULL);**

nVerb Index of the verb to execute; 0 is the primary verb, 1 is the secondary verb, and so forth.

bShow **TRUE** if the server window is to be shown; **FALSE** if the server should remain active without being visible.

bTakeFocus **TRUE** if the server should receive the input focus. Relevant only if *bShow* is **TRUE**.

pWndContainer Pointer to the client window object that contains the item.

lpBounds Pointer to a **RECT** structure or **CRect** object that contains the bounding rectangle in which the destination document displays the item. Units are determined by the device-context mapping mode. Can be **NULL**.

Remarks

Call this function to execute the specified verb if you want full control of how the server will be displayed. For default server behavior, call the **DoVerb** member function to execute a verb. Both functions cause the **OnDoVerb** member function of **COleServerItem** to be executed. If the verb specified is Edit, the server application is launched in a separate window and editing occurs asynchronously. You typically specify the primary verb when the user of the client application double-clicks the item. The action taken in response to each verb depends on the server. If the server supports only one action, it takes that action no matter which value is specified in the *nVerb* parameter.

See Also **COleClientItem::DoVerb**, **COleServerItem::OnDoVerb**, **::OleActivate**

COleClientItem::CanPaste

static BOOL PASCAL CanPaste(OLEOPT_RENDER *renderopt* **= olerender_draw, OLECLIPFORMAT** *cfFormat* **= 0);**

renderopt Flag specifying how the server will render the item. For possible values, see **COleClientItem::CreateNewObject**.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

Remarks Call this function to see if an embedded item can be pasted from the Clipboard. This function is called for you by the framework when enabling or disabling the Paste command on the Edit menu.

Return Value Nonzero if the Clipboard currently contains an embeddable or static (metafile-based) OLE item; otherwise 0.

See Also **COleClientItem::CanPasteLink**, **COleClientItem::CreateFromClipboard**, **COleClientItem::CreateStaticFromClipboard**, **::OleQueryCreateFromClip**

COleClientItem::CanPasteLink

static BOOL PASCAL CanPasteLink(OLEOPT_RENDER *renderopt* **= olerender_draw, OLECLIPFORMAT** *cfFormat* **= 0);**

renderopt Flag specifying how the server will render the item. For possible values, see **COleClientItem::CreateNewObject**.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

Remarks Call this function to see if a linked item can be pasted from the Clipboard. This function is called for you by the framework when enabling or disabling the Paste Link command on the Edit menu.

Return Value Nonzero if the Clipboard currently contains a linkable OLE item; otherwise 0.

See Also **COleClientItem::CanPaste**, **COleClientItem::CreateLinkFromClipboard**, **::OleQueryCreateFromClip**

COleClientItem::CloseLink

void CloseLink( );

Remarks Call this function to close the link between an open linked item and the server application. This function does not destroy the linked item; the item can be reconnected later.

See Also **COleClientItem::ReconnectLink**, **COleClientItem::UpdateLink**, **::OleClose**

COleClientItem::COleClientItem

COleClientItem(COleClientDoc* *pContainerDoc* **);**

pContainerDoc Pointer to the registered client document that will contain this item.

Remarks Constructs a **COleClientItem** object and adds it to the container document's collection of document items. You must call one of the following creation member functions before you use the item: **CreateFromClipboard**, **CreateInvisibleObject**, **CreateStaticFromClipboard**, **CreateLinkFromClipboard**, **CreateNewObject**, or **CreateCloneFrom**.

See Also **COleClientDoc**, **COleDocument::AddItem**

COleClientItem::CopyToClipboard

void CopyToClipboard( );

Remarks Call this function to copy the item to the Clipboard. Typically, you call this function when writing message handlers for the Copy or Cut commands from the Edit menu. You must implement selection in your client application to implement the Copy or

Cut commands. To use this function, you should open and empty the Clipboard, call **CopyToClipboard** for the selected item, and then close the Clipboard.

See Also **::OleCopyToClipboard**

COleClientItem::CreateCloneFrom

BOOL CreateCloneFrom(COleClientItem* *pSrcItem*, **LPCSTR** *lpszItemName* **);**

pSrcItem Pointer to the OLE item to be duplicated.

lpszItemName Pointer to the client name of the new item.

Remarks Call this function to create a copy of the specified item. The copy is identical to the source item but is not connected to the server. You can use this function to support "undo" or "revert" operations.

Return Value Nonzero if successful; otherwise 0.

See Also **::OleClone**

COleClientItem::CreateFromClipboard

BOOL CreateFromClipboard(LPCSTR *lpszItemName*, **OLEOPT_RENDER** *renderopt* = **olerender_draw, OLECLIPFORMAT** *cfFormat* = **0);**

lpszItemName Pointer to the client name of the new item.

renderopt Flag specifying how the server will render the item. For the possible values, see **COleClientItem::CreateNewObject**.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

Remarks Call this function to create an embedded item from the contents of the Clipboard. You typically call this function from the message handler for the Paste command on the Edit menu. (The Paste command is enabled by the framework if the **CanPaste** member function returns nonzero.) If the function is unsuccessful, try calling **CreateStaticFromClipboard** to paste a static (metafile-based) item.

Return Value Nonzero if successful; otherwise 0.

See Also **COleClientItem::CreateStaticFromClipboard**, **COleClientItem::CanPaste**, **::OleCreateFromClip**

COleClientItem::CreateInvisibleObject

BOOL CreateInvisibleObject(LPCSTR *lpszTypeName*, **LPCSTR** *lpszItemName*, **OLEOPT_RENDER** *renderopt* = **olerender_draw**, **OLECLIPFORMAT** *cfFormat* = **0**, **BOOL** *bActivate* = **FALSE);**

lpszTypeName Pointer to the type name of the new item to create. This string is usually obtained from the global function **AfxOleInsertDialog**.

lpszItemName Pointer to the client name of the new item.

renderopt Flag specifying how the server will render the item. For the possible values, see **COleClientItem::CreateNewObject**.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

bActivate Specifies whether to activate the item or not.

Remarks Call this function to create an item without displaying the server application to the user. This is an advanced operation; typically you call **CreateNewObject**.

Return Value Nonzero if successful; otherwise 0.

See Also **COleClientItem::CreateNewObject**, **::OleCreateInvisible**

COleClientItem::CreateLinkFromClipboard

BOOL CreateLinkFromClipboard(LPCSTR *lpszItemName*, **OLEOPT_RENDER** *renderopt* = **olerender_draw**, **OLECLIPFORMAT** *cfFormat* = **0);**

lpszItemName Pointer to the client name of the new item.

renderopt Flag specifying how the server will render the item. For the possible values, see **COleClientItem::CreateNewObject**.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

Remarks

Call this function to create a linked item from the contents of the Clipboard. You typically call this function from the message handler for the Paste Link command on the Edit menu. (The Paste Link command is enabled by the framework if the **CanPasteLink** member function returns nonzero.)

Return Value

Nonzero if successful; otherwise 0.

See Also

COleClientItem::CanPasteLink, **::OleCreateLinkFromClip**

COleClientItem::CreateNewObject

BOOL CreateNewObject(LPCSTR *lpszTypeName*, **LPCSTR** *lpszItemName*,
OLEOPT_RENDER *renderopt* = **olerender_draw,**
OLECLIPFORMAT *cfFormat* = **0);**

lpszTypeName Pointer to the type name of the new item to create. This string is usually obtained from the global function **AfxOleInsertDialog**.

lpszItemName Pointer to the name of the new item.

renderopt Flag specifying how the server will render the item. This parameter may have one of the following values:

- **olerender_draw** The item is drawn using **COleClientItem::Draw**. In this case the OLE system DLL obtains and manages the presentation data and stores the Native data for archiving purposes only.
- **olerender_none** The OLE system DLL does not obtain the presentation data and does not draw the object. The client calls **COleClientItem::GetData** to retrieve the server data in Native format, and it is assumed that the client knows how to interpret this format.
- **olerender_format** The client calls **COleClientItem::GetData** to retrieve data in the format specified by *cfFormat*. The client then uses the retrieved data to render the item.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

Remarks

Call this function to create an embedded item; this function launches the server application to allow the user to create the item. You typically call this function from the message handler for the Insert New Object command on the Edit menu. To create a linked item, use the **CreateLinkFromClipboard** function.

Return Value Nonzero if successful; otherwise 0.

See Also **AfxOleInsertDialog**, **COleClientItem::CreateLinkFromClipboard**, **::OleCreate**

COleClientItem::CreateStaticFromClipboard

BOOL CreateStaticFromClipboard(LPCSTR *lpszItemName*,
OLEOPT_RENDER *renderopt* = **olerender_draw**,
OLECLIPFORMAT *cfFormat* = **0);**

lpszItemName Pointer to the client name of the new item.

renderopt Flag specifying how the server will render the item. For possible values, see **COleClientItem::CreateNewObject**.

cfFormat Specifies the Clipboard data format if *renderopt* is **olerender_format**.

Remarks Call this function to create a static (metafile-based) embedded item from the contents of the Clipboard. You typically call this function from the message handler for the Paste command on the Edit menu, following an unsuccessful call to **CreateFromClipboard**. (The Paste command is enabled by the framework if the **CanPaste** member function returns nonzero.)

Return Value Nonzero if successful; otherwise 0.

See Also **COleClientItem::CreateFromClipboard**, **::OleCreateFromClip**

COleClientItem::Delete

void Delete( );

Remarks Call this function to delete the item. If the item is embedded, the native data for the item is deleted. If the item is an open linked item, this function closes it. Unlike the **Release** member function, this function indicates that the item has been permanently removed. The **COleClientItem** destructor calls **Delete** for embedded items.

See Also **COleClientItem::Release**, **::OleDelete**

COleClientItem::DoVerb

virtual BOOL DoVerb(UINT *nVerb* **);**

nVerb Index of the verb to execute; 0 is the primary verb, 1 is the secondary verb, and so forth.

Remarks

Call this function to execute the specified verb. This function uses the **Activate** member function to execute the verb; it also catches exceptions thrown as a result and alerts the user if an error occurs.

You typically specify the primary verb when the user of the client application double-clicks the item. The action taken in response to each verb depends on the server. If the server supports only one action, it takes that action no matter which value is specified in the *nVerb* parameter.

Return Value

Nonzero if the verb was sucessfully executed; otherwise 0.

See Also

COleClientItem::Activate

COleClientItem::Draw

BOOL Draw(CDC* *pDC*, **LPCRECT** *lpBounds*,
LPCRECT *lpWBounds* = **NULL, CDC*** *pFormatDC* = **NULL);**

pDC Pointer to a **CDC** object used for drawing the item.

lpBounds Pointer to a **CRect** object or **RECT** structure that defines the bounding rectangle in which to draw the object (in logical units determined by the device context).

lpWBounds Pointer to a **CRect** object or **RECT** structure that defines the bounding rectangle if *pDC* specifies a metafile device context. **NULL** if *pDC* points to a screen device context.

pFormatDC Pointer to a **CDC** object describing the target device for which to format the item. This parameter is used only by handler DLLs and is usually **NULL**.

Remarks

Call this function to draw the item into the specified bounding rectangle using the specified device context. The function uses the metafile representation of the item created by the **OnDraw** member function of **COleServerItem**.

Typically you use **Draw** for screen display, passing the screen device context as *pDC*. In this case, you need specify only the first two parameters. If you pass a metafile device context as *pDC*, the rectangle specified by *lpWBounds* must contain the rectangle specified by *lpBounds*. The *pFormatDC* parameter is used for formatting purposes by handler DLLs and must not be a metafile device context.

The *lpBounds* parameter identifies the rectangle in the target device context (relative to its current mapping mode). Rendering may involve scaling the picture and can be used by client applications to impose a view scaling between the displayed view and the final printed image.

Return Value Nonzero if successful; otherwise 0.

See Also **COleClientItem::SetBounds**, **COleServerItem::OnDraw**, **::OleDraw**

COleClientItem::EnumFormats

OLECLIPFORMAT EnumFormats(OLECLIPFORMAT *nFormat* **) const;**

nFormat Specifies the format returned by the previous call to the **EnumFormats** member function. For the first call to this function, this parameter is **NULL**. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows **RegisterClipboardFormat** function.

Remarks Call this function to retrieve the data formats available for the item. Call this function in a loop to retrieve all the formats, each time passing the format returned by the previous call.

Return Value The next (or first) available format; **NULL** if no more formats are available.

See Also **COleClientItem::GetData**, **::OleEnumFormats**

COleClientItem::GetBounds

BOOL GetBounds(LPRECT *lpBounds* **);**

lpBounds Pointer to a **CRect** object or **RECT** structure that will receive the bounds information.

Remarks Call this function to retrieve the extents of the bounding rectangle for the item on the target device. The coordinates are in **MM_HIMETRIC** units and the top and left coordinates are always 0.

Return Value Nonzero if successful; 0 if the item is blank.

See Also **COleClientItem::SetBounds**, **::OleQueryBounds**

COleClientItem::GetData

HANDLE GetData(OLECLIPFORMAT *nFormat*, **BOOL&** *bMustDelete* **);**

nFormat Specifies the format in which data is returned. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows **RegisterClipboardFormat** function.

bMustDelete A reference to a **BOOL** value that the function sets to **TRUE** if you are responsible for the deletion of the retrieved data (through the Windows **GlobalFree** function). If the function sets *bMustDelete* to **FALSE**, then you must copy the data if you need to keep it.

Remarks Call this function to retrieve data from the item in the requested format.

Return Value A handle to an entity that contains the data. If *nFormat* is **CF_METAFILEPICT** or **CF_BITMAP**, then this handle is a Windows graphics device interface (GDI) object handle; otherwise, it is a global memory block handle.

See Also **COleClientItem::RequestData**, **::OleGetData**

COleClientItem::GetDocument

COleClientDoc* GetDocument( ) const;

Remarks Call this function to get a pointer to the document that contains the item. This allows access to the client document that you passed as an argument to the **COleClientItem** constructor.

Return Value A pointer to the document that contains the item. **NULL** if the item is not part of a document.

See Also **COleClientItem::COleClientItem**, **COleClientDoc**

COleClientItem::GetLastStatus

OLESTATUS GetLastStatus( ) const;

Remarks Returns the status of the last OLE operation. For member functions that return a **BOOL** value of **FALSE**, **GetLastStatus** returns more detailed failure information. Be aware that most OLE member functions throw exceptions for more serious errors.

Return Value See **COleException** for a list of return values.

COleClientItem::GetLinkUpdateOptions

OLEOPT_UPDATE GetLinkUpdateOptions( );

Remarks Call this function to get the current value of the link-update option for the item. This is an advanced operation.

Return Value One of the following values:

- **oleupdate_always** Update the linked object whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
- **oleupdate_onsave** Update the linked object when the source document is saved by the server.
- **oleupdate_oncall** Update the linked object only on request from the client application. This option supports the Manual link-update radio button in the Links dialog box.

See Also **::OleGetLinkUpdateOptions**

COleClientItem::GetName

CString GetName( );

Remarks Call this function to get the client name of the item. This is the name passed in when the object was created or last renamed.

Return Value The name of the item.

See Also **::OleQueryName**

COleClientItem::GetSize

DWORD GetSize( );

Remarks Call this function to get the number of bytes in the native representation of the item. You can use this information to determine the space required for saving it.

Return Value Number of bytes required to save the item.

See Also **::OleQuerySize, CObject::Serialize**

COleClientItem::GetType

UINT GetType( );

Remarks Call this function to determine whether the item is embedded, linked, or static.

Return Value An unsigned integer with one of the following values:

- **OT_LINK** The item is a link.
- **OT_EMBEDDED** The item is embedded.
- **OT_STATIC** The item is a static (metafile-based) picture.

See Also **::OleQueryType**

COleClientItem::InWaitForRelease

static BOOL PASCAL InWaitForRelease( );

Remarks Call this function from your main window's **OnCommand** or **OnCmdMsg** member function to disable user commands until all servers respond.

Return Value Nonzero if this client application is still waiting for a server to complete an operation; otherwise 0.

COleClientItem::IsEqual

BOOL IsEqual(COleClientItem* *pOtherItem* **);**

pOtherItem Pointer to an OLE item object that is to be compared with this item.

Remarks Call this function to compare two OLE items. Embedded items are equal if their type name, item name, and native data are identical. Linked items are equal if their type name, item name, and document name are identical.

Return Value Nonzero if the items are equal; otherwise 0.

See Also **::OleEqual**

COleClientItem::IsOpen

BOOL IsOpen( );

Remarks Call this function to see if the item is connected to the OLE system DLL. Typically, an item is connected after a successful call to one of the **COleClientItem** creation functions.

Return Value Nonzero if the item is connected; otherwise 0.

See Also **COleClientItem::CreateFromClipboard**, **COleClientItem::CreateStaticFromClipboard**, **COleClientItem::CreateLinkFromClipboard**, **COleClientItem::CreateNewObject**, **COleClientItem::CreateCloneFrom**, **::OleQueryOpen**

COleClientItem::OnChange

Protected **virtual void OnChange(OLE_NOTIFICATION** *wNotification* **) = 0;** ♦

wNotification Reason the server changed this item. It can have one of the following values:

- **OLE_CHANGED** The user of the server application modified the linked item. This notification is not sent for embedded items.

- **OLE_SAVED** The user of the server application saved the document containing the item.
- **OLE_CLOSED** The user of the server application closed the document containing the item.

The **OLE_RENAMED** notification is handled by the **OnRenamed** member function.

Remarks

Called by the framework when the user of the server application modifies the item or saves or closes the document containing the item. (If the server application is written with the Microsoft Foundation Class Library, this function is called in response to the **Notify** member functions of **COleServerDoc** or **COleServerItem**.) There is no default implementation. You must override this function to respond to changes in the item's state. Typically you update the item's appearance by invalidating the area in which the item is displayed.

See Also

COleClientItem::OnRenamed, **COleServerItem::NotifyChanged**, **COleServerDoc::NotifyChanged**, **COleServerDoc::NotifyClosed**, **COleServerDoc::NotifySaved**

COleClientItem::OnRenamed

Protected

virtual void OnRenamed( ); ♦

Remarks

Called by the framework when the user of the server application renames the document containing the item. (If the server application is written with the Microsoft Foundation Class Library, this function is called in response to the **NotifyRename** member function of **COleServerDoc**.) This function is called only for linked items, not for embedded items. The default implementation does nothing. Override this function if you want to perform special processing when an item is renamed.

See Also

COleClientItem::OnChange, **COleServerDoc::NotifyRename**

COleClientItem::ReconnectLink

void ReconnectLink( );

Remarks

Call this function to reestablish a link between an open linked item and the server. Typically, you call this function after closing a link with the **CloseLink** member function. If the item is not open, **ReconnectLink** does not open it.

See Also

COleClientItem::CloseLink, **::OleReconnect**

COleClientItem::Release

void Release( );

Remarks

Call this function to release the connection to a linked item and close the link if it was open. It does not destroy the item. **Release** is called by the **COleClientItem** destructor for linked items.

See Also

COleClientItem::Delete, **::OleRelease**

COleClientItem::Rename

void Rename(LPCSTR *lpszNewname* **);**

lpszNewname Pointer to the new client name for the item.

Remarks

Call this function to rename the item. The name must be unique within the document and must be preserved when the document is saved.

See Also

::OleRename

COleClientItem::RequestData

void RequestData(OLECLIPFORMAT *nFormat* **);**

nFormat Specifies the format in which data is returned. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows **RegisterClipboardFormat** function.

Remarks

Call this function to retrieve data in a specified format from the server application. An exception is thrown if the server does not support data requests. The client application should be connected to the server application when the client calls **RequestData**. After **RequestData** returns, the client can retrieve the data with the **GetData** member function, and it can examine information through other member functions such as **GetBounds** and **GetSize**.

See Also

COleClientItem::GetData, **COleClientItem::GetBounds**, **COleClientItem::GetSize**, **::OleRequestData**

COleClientItem::SetBounds

void SetBounds(LPCRECT *lpRect* **);**

lpRect Pointer to a **CRect** object or **RECT** structure that contains the bounds information.

Remarks

Call this function to set the bounding rectangle on the target device for the item; this causes the **OnSetBounds** member function of the corresponding **COleServerItem** object to be called. The coordinates must be in **MM_HIMETRIC** units. This function is only meaningful for embedded items. The size of a linked item is determined by the source document for the link. The bounding rectangle does not need to have the same dimensions as the rectangle specified by the **Draw** member function's *lpBounds* parameter. These dimensions may be different because of the view scaling used by the window in which the item is displayed. The client application can call **SetBounds** to make the server reformat the picture to better fit the client's rectangle.

See Also

COleServerItem::OnSetBounds, **::OleSetBounds**

COleClientItem::SetColorScheme

void SetColorScheme(const LOGPALETTE FAR* *lpLogPalette* **);**

lpLogPalette Pointer to a Windows **LOGPALETTE** structure.

Remarks

Call this function to specify a recommended color scheme for the server application to use while displaying the item; this causes the **OnSetColorScheme** member function of the corresponding **COleServerItem** object to be called. The server does not have to use the specified palette. The client does not need to call **SetColorScheme** every time a server is opened.

The first palette entry in the **LOGPALETTE** structure specifies the foreground color recommended by the client application. The second palette entry specifies the background color. The first half of the remaining palette entries are fill colors, and the second half are colors for lines and text. Client applications should specify an even number of palette entries. When there is an uneven number of entries, the server interprets the odd entry as a fill color; that is, if there were five entries, three would be interpreted as fill colors and two as line and text colors. When server applications render metafiles, they should use the suggested palette.

See Also

COleServerItem::OnSetColorScheme, **::OleSetColorScheme**

COleClientItem::SetData

void SetData(OLECLIPFORMAT *nFormat*, **HANDLE** *hData* **);**

nFormat Specifies the format in which data is returned. This parameter can be one of the predefined Clipboard formats or the value returned by the native Windows **RegisterClipboardFormat** function.

hData Identifies a memory object that contains the data in the format specified by the server. Do not free this memory; the server will free it.

Remarks

Call this function to send data to the server application using the specified format; this causes the **OnSetData** member function of the corresponding **COleServerItem** object to be called. An exception is thrown if the server cannot accept the data or the specified data format.

See Also

COleServerItem::OnSetData, **::OleSetData**

COleClientItem::SetHostNames

void SetHostNames(LPCSTR *lpszHost*, **LPCSTR** *lpszHostObj* **);**

lpszHost Pointer to the name of the client application.

lpszHostObj Pointer to the client's name for the item.

Remarks

Call this function to specify the name of the client application and the client's name for the specified object; this calls the **OnSetHostNames** member function of the **COleServerDoc** object that contains the item on the server side. This information can be used in window titles when the server application edits the item. It is not necessary to call **SetHostNames** each time a server is activated.

See Also

COleServerDoc::OnSetHostNames, **::OleSetHostNames**

COleClientItem::SetLinkUpdateOptions

void SetLinkUpdateOptions(OLEOPT_UPDATE *updateOpt* **);**

updateOpt The value of the link-update option for this item. This value must be one of the following:

- **oleupdate_always** Update the linked object whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
- **oleupdate_onsave** Update the linked object when the source document is saved by the server.
- **oleupdate_oncall** Update the linked object only on request from the client application. This option supports the Manual link-update radio button in the Links dialog box.

Remarks

Call this function to set the link-update option for the presentation of the specified linked item. Typically you should not change the update options chosen by the user in the Links dialog box.

See Also

COleClientItem::GetLinkUpdateOptions, **AfxOleLinksDialog**, **::OleSetLinkUpdateOptions**

COleClientItem::SetTargetDevice

void SetTargetDevice(HGLOBAL *hData* **);**

hData Handle to an **OLETARGETDEVICE** structure that describes the target device. Do not free this structure; the server will free it.

Remarks

Call this function to specify an item's target output device; this causes the **OnSetTargetDevice** member function of the corresponding **COleServerItem** object to be called. This function allows a linked or embedded item to be formatted correctly for a target device, even when the item is rendered on a different device. A client application should call this function whenever the target device changes so that servers can be notified to change the rendering of the item if necessary. The client application should call the **UpdateLink** member function after calling **SetTargetDevice** to ensure that the information is sent to the server and that the server can make the necessary changes to the item's presentation. The client application should call the **Draw** member function to redraw the item if it receives a notification from the server that the item has changed. The client does not need to call **SetTargetDevice** every time a server is activated.

See Also

COleClientItem::Draw, **COleClientItem::UpdateLink**, **COleServerItem::OnTargetDevice**, **::OleSetTargetDevice**

COleClientItem::UpdateLink

void UpdateLink( );

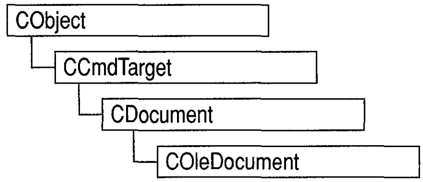
Remarks

Call this function to update the item immediately. The user can also manually update individual links using the Links dialog box.

See Also

AfxOleLinksDialog, **::OleUpdate**

class COleDocument : public CDocument



COleDocument is the base class for Object Linking and Embedding (OLE) documents. **COleDocument** is derived from **CDocument**, allowing your OLE applications to use the document/view architecture provided by the Microsoft Foundation Class Library. In addition, the **COleDocument** class defines an interface that treats a document as a collection of **CDocItem** objects. This interface is needed by both client and server applications because their documents must be able to contain OLE items.

You do not use **COleDocument** directly; instead, use the derived classes **COleClientDoc** and **COleServerDoc**. Use those classes as the base class for documents in your client and server applications, respectively.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **CDocItem**, **COleServerDoc**, **COleClientDoc**, **COleServerItem**, **COleClientItem**

Construction/Destruction—Public Members

COleDocument	Constructs a **COleDocument** object.

Operations—Public Members

AddItem	Adds an item to the list of items maintained by the document.
GetNextItem	Returns all the items in the document when called iteratively.
GetStartPosition	Gets the initial position to begin iteration.
IsOpenClientDoc	Tests if the document is a registered client document.
IsOpenServerDoc	Tests if the document is a registered server document.
RemoveItem	Removes an item from the list of items maintained by the document.

Member Functions

COleDocument::AddItem

void AddItem(CDocItem* *pItem* **);**

pItem Pointer to the document item being added.

Remarks

Call this function to add an item to the document. You typically do not need to call this function explicitly; this function is called by the constructors for **COleClientItem** and **COleServerItem**.

See Also

CDocItem, **COleDocument::RemoveItem**, **COleServerItem::COleServerItem**, **COleClientItem::COleClientItem**

COleDocument::COleDocument

COleDocument( );

Remarks

Constructs a **COleDocument** object.

COleDocument::GetNextItem

virtual CDocItem* GetNextItem(POSITION& *rPosition* **);**

rPosition A reference to a **POSITION** value set by a previous call to **GetNextItem**; the initial value is returned by the **GetStartPosition** member function. This must not be **NULL**.

Remarks

Call this function repeatedly to access each of the items in your document. After each call, the value of *rPosition* is set to the **POSITION** value of the next item in the document. If the retrieved element is the last in the document, the new value of *rPosition* is **NULL**.

Return Value A pointer to the document item at the specified position.

See Also **COleDocument::GetStartPosition**

Example

```
// pDoc points to a COleDocument object
POSITION pos = pDoc->GetStartPosition();
while( pos != NULL )
{
    CDocItem *pItem = pDoc->GetNextItem( pos );
    // use pItem
}
```

COleDocument::GetStartPosition

virtual POSITION GetStartPosition( ) const;

Remarks Call this function to get the position of the first item in the document. Pass the value returned to **GetNextItem**.

Return Value A **POSITION** value that can be used to begin iterating through the document's items; **NULL** if the document is empty.

See Also **COleDocument::GetNextItem**

COleDocument::IsOpenClientDoc

BOOL IsOpenClientDoc( ) const;

Remarks Call this function to see if the document is a registered client document. Note that a document can be both a client document and a server document if your application supports both.

Return Value Nonzero if the document is a registered client document; otherwise 0.

See Also **COleClientDoc::RegisterClientDoc**, **COleDocument::IsOpenServerDoc**

COleDocument::IsOpenServerDoc

BOOL IsOpenServerDoc( ) const;

Remarks Call this function to see if the document is a registered server document. Note that a document can be both a client document and a server document if your application supports both.

Return Value Nonzero if the document is a registered server document; otherwise 0.

See Also **COleServerDoc::RegisterServerDoc**, **COleDocument::IsOpenClientDoc**

COleDocument::RemoveItem

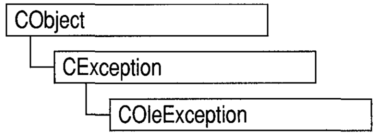
void RemoveItem(CDocItem* *pItem* **);**

pItem Pointer to the document item to be removed.

Remarks Call this function to remove an item from the document. You typically do not need to call this function explicitly; this function is called by the destructors for **COleClientItem** and **COleServerItem**.

See Also **CDocItem**, **COleServerItem**, **COleClientItem**, **COleDocument::AddItem**

class COleException : public CException



A **COleException** object represents an exception condition related to an Object Linking and Embedding (OLE) operation. The **COleException** class includes a public data member that holds the status code indicating the reason for the exception.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

Data Members—Public Members

m_status	Contains the status code that indicates the reason for the exception.

Construction/Destruction—Public Members

COleException	Constructs a **COleException** object.

Member Functions

COleException::COleException

COleException(OLESTATUS *status* **);**

status An enumerated type variable that indicates the reason for the exception. Must be one of the following enumerators:

- **OLE_OK** Function operated correctly (does not throw an exception).
- **OLE_BUSY** Tried to execute a member function while another operation was in progress.
- **OLE_ERROR_STREAM** **OLESTREAM** stream error.
- **OLE_ERROR_STATIC** Nonstatic item expected.

- **OLE_ERROR_BLANK** Critical data missing.
- **OLE_ERROR_DRAW** Error while drawing.
- **OLE_ERROR_METAFILE** Invalid metafile.
- **OLE_ERROR_ABORT** Client chose to abort metafile drawing.
- **OLE_ERROR_CLIPBOARD** Failed to get or set Clipboard data.
- **OLE_ERROR_FORMAT** Requested format not available.
- **OLE_ERROR_GENERIC** General error.
- **OLE_ERROR_DATATYPE** Data format not supported.
- **OLE_ERROR_PALETTE** Invalid color palette.
- **OLE_ERROR_NOT_LINK** Not a linked item.
- **OLE_ERROR_NOT_EMPTY** Client document contains items.
- **OLE_ERROR_SIZE** Incorrect buffer size passed to function that places a string in the caller's buffer.
- **OLE_ERROR_DRIVE** Drive letter in document name invalid.
- **OLE_ERROR_NETWORK** Failed to establish connection to network share on which the document is located.
- **OLE_ERROR_NAME** Invalid name (document name, item name, and so forth) passed to function.
- **OLE_ERROR_TEMPLATE** Server failed to load template.
- **OLE_ERROR_NEW** Server failed to create new document.
- **OLE_ERROR_EDIT** Server failed to create embedded instance.
- **OLE_ERROR_OPEN** Server failed to open document; possible invalid link.
- **OLE_ERROR_NOT_OPEN** Item not open for editing.
- **OLE_ERROR_LAUNCH** Failed to launch server.
- **OLE_ERROR_COMM** Failed to communicate with server.
- **OLE_ERROR_TERMINATE** Error in termination.
- **OLE_ERROR_COMMAND** Error in execution.
- **OLE_ERROR_SHOW** Error in showing.
- **OLE_ERROR_DOVERB** Error in sending do verb, or invalid verb.
- **OLE_ERROR_ADVISE_NATIVE** Item could be missing.
- **OLE_ERROR_ADVISE_PICT** Item could be missing or server doesn't understand this format.
- **OLE_ERROR_ADVISE_RENAME** Server doesn't support rename.
- **OLE_ERROR_POKE_NATIVE** Failure in poking native data to server.

- **OLE_ERROR_REQUEST_NATIVE** Server failed to render native data.
- **OLE_ERROR_REQUEST_PICT** Server failed to render presentation data.
- **OLE_ERROR_SERVER_BLOCKED** Trying to block a blocked server, or trying to revoke a blocked server or document.
- **OLE_ERROR_REGISTRATION** Server not registered in OLE registration database.
- **OLE_ERROR_ALREADY_REGISTERED** Trying to register same document multiple times.
- **OLE_ERROR_TASK** Server or client task invalid.
- **OLE_ERROR_OUTOFDATE** Item out of date.
- **OLE_ERROR_CANT_UPDATE_CLIENT** Client of embedded document doesn't accept updates.
- **OLE_ERROR_UPDATE** Error while trying to update.
- **OLE_WARN_DELETE_DATA** Caller must delete data when done with it (warning).

Remarks Constructs a **COleException** object. Do not use this constructor directly; instead call the global function **AfxThrowOleException**.

See Also **AfxThrowOleException**

Data Members

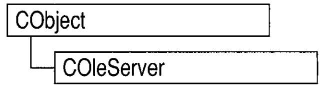
COleException::m_status

OLESTATUS m_status;

Remarks This data member holds the status code that indicates the reason for the exception. This variable is set by the constructor. See the **COleException** constructor documentation for a list of **OLESTATUS** enumerators.

See Also **COleException::COleException**

class COleServer : public CObject



COleServer is the base class for Object Linking and Embedding (OLE) servers. One **COleServer** object is needed for each type of document a server application supports; for example, if your server application supports both worksheets and charts, you need to have two **COleServer** objects. Use the **COleServer** class if you are writing a mini-server (that is, a server application that is only launched by clients to edit embedded items). If you are writing a full server (that is, a server application that supports loading and saving files to and from disk), you can use the **COleTemplateServer** class, which combines a **CDocTemplate** object with a server.

COleServer defines several overridable member functions that are called by the OLE system dynamic-link library (DLL) in response to requests from client applications. Through these member functions, the client instructs the server to open embedded items as documents or open the documents that are the source of linked items.

To use **COleServer**, derive a class from it and implement the **OnCreateDoc** and **OnEditDoc** member functions, which allow your application to open and edit embedded items as documents. Derive a class from **COleServerDoc** to implement the documents edited by your server application and return objects of that class from **OnCreateDoc** and **OnEditDoc**.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **COleTemplateServer**, **COleServerDoc**, **COleServerItem**

Construction/Destruction—Public Members

COleServer	Constructs a **COleServer** object.

Registration/Revocation—Public Members

Register	Registers the server with the OLE system DLL.
BeginRevoke	Begins server shutdown (called by the destructor).

Status—Public Members

IsOpen	Indicates whether the server is currently operational and registered.
GetServerName	Returns the name of the server registered with the OLE system DLL.

Overridables—Protected Members

OnCreateDoc	Called to create a document for a new embedded item. Implementation required.
OnEditDoc	Called to create a document to edit an existing embedded item. Implementation required.
OnOpenDoc	Called to open an existing document containing the source of a linked item.
OnCreateDocFromTemplateFile	Called to create a new document based on another file.
OnExecute	Called to handle dynamic data exchange (DDE) **WM_DDE_EXECUTE** messages.
OnExit	Called to instruct the server to quit.

Member Functions

COleServer::BeginRevoke

void BeginRevoke( );

Remarks Call this function to close any registered documents and begin the server shutdown procedure. You typically call this function when the user exits your application. This function is also called by the **COleServer** destructor. This function does not wait for the OLE system DLL to complete the revoke operation; the DLL calls the implementation member function **OnRelease** when it is safe for the application to quit.

See Also **::OleRevokeServer**

COleServer::COleServer

COleServer(BOOL *bLaunchEmbedded* **);**

bLaunchEmbedded **TRUE** if the server application was launched with the "/Embedded" command-line argument.

Remarks Constructs a **COleServer** object. The server cannot receive requests from clients until you call the **Register** member function.

See Also **COleServer::Register**

COleServer::GetServerName

const CString& GetServerName( ) const;

Remarks Call this function to get the name of the server registered with the OLE system DLL. This is the name that was passed to the **Register** member function.

Return Value The registered name of the server.

See Also **COleServer::Register**

COleServer::IsOpen

BOOL IsOpen( ) const;

Remarks Call this function to see if the server is registered with the OLE system DLL.

Return Value Returns nonzero if the server has been successfully registered; otherwise 0.

See Also **COleServer::Register**

COleServer::OnCreateDoc

Protected

virtual COleServerDoc* OnCreateDoc(LPCSTR *lpszTypeName*, **LPCSTR** *lpszDoc* **) = 0; ♦**

lpszTypeName Pointer to the type name of the document being created.

lpszDoc Pointer to the name of the document being created; note that this is not a filename because embedded items are not stored as their own files. This name can be used to identify the document in window titles.

Remarks

Called by the framework when a new embedded item is being created, that is, when the user of a client application executes the Insert New Object command. There is no default implementation. You must override this function to create a new document object of the specified type or return a pointer to an existing document object. The document object must be an object of a **COleServerDoc**-derived class.

This function is overriden for you in the derived class **COleTemplateServer** to use the document creation facilities of a **CDocTemplate** object.

Return Value

If successful, a pointer to a server document; otherwise **NULL**.

See Also

COleServer::OnEditDoc, **COleServerDoc**, **COleTemplateServer**, **COleClientItem::CreateNewObject**

COleServer::OnCreateDocFromTemplateFile

Protected

virtual COleServerDoc* OnCreateDocFromTemplateFile(LPCSTR *lpszTypeName*, **LPCSTR** *lpszDoc*, **LPCSTR** *lpszTemplate* **); ♦**

lpszTypeName Pointer to the type name of the document being created.

lpszDoc Pointer to the name of the document being created. Note that this is not a filename because embedded items are not stored as their own files. This name can be used to identify the document in window titles.

lpszTemplate Pointer to the fully qualified name of a file on which the new document should be based.

Remarks

Called by the framework to create a server document for a new embedded item and initialize it with the contents of the specified file. The default implementation does nothing and returns **NULL**. Override this function if you want to use an existing file to initialize new embedded items. In such a situation, you must determine your own format for initializing the item from the template file. The document object you create must be an object of a **COleServerDoc**-derived class.

Note that the file used as the template for the embedded item is unrelated to the **CDocTemplate** classes defined by the Microsoft Foundation Class Library.

Return Value If successful, a pointer to a server document; otherwise **NULL**. Returns **NULL** if the server does not support this feature.

COleServer::OnEditDoc

Protected **virtual COleServerDoc* OnEditDoc(LPCSTR** *lpszTypeName*, **LPCSTR** *lpszDoc*) **= 0;** ♦

lpszTypeName Pointer to the type name of the document being opened.

lpszDoc Pointer to the name of an existing document; note that this is not a filename because embedded items are not stored as their own files. This name can be used to identify the document in window titles.

Remarks Called by the framework when an existing embedded item is opened for editing, that is, when the user of a client application edits an embedded item. There is no default implementation. You must override this function to create a new document object of the specified type or to return a pointer to an existing document object. The document object you create must be an object of a **COleServerDoc**-derived class. Note that this function is called only for embedded items; the **OnOpenDoc** member function is called for linked items.

This function is overriden for you in the derived class **COleTemplateServer** to use the document creation facilities of a **CDocTemplate** object.

Return Value If successful, a pointer to a server document; otherwise **NULL**.

See Also **COleServer::OnOpenDoc**, **COleServerDoc**, **COleTemplateServer**

COleServer::OnExecute

Protected **virtual OLESTATUS OnExecute(LPVOID** *lpCommands* **);** ♦

lpCommands Points to a block of memory that contains dynamic data exchange (DDE) **WM_DDE_EXECUTE** command strings.

Remarks Called by the framework when the client sends DDE **WM_DDE_EXECUTE** command strings to the server document. The default implementation does nothing and returns **OLE_ERROR_COMMAND**. Override this funtion to handle DDE **WM_DDE_EXECUTE** messages. Do not delete memory referenced by *lpCommands*.

Return Value **OLE_OK** if successful; any other value indicates failure. See the **COleException** class for a list of possible values.

COleServer::OnExit

Protected **virtual OLESTATUS OnExit( );** ♦

Remarks Called by the framework to tell the server to close documents and quit. The default implementation calls the **BeginRevoke** member function to start shutting down the server application. Override this function if you want to perform special processing when you exit.

Return Value See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also **COleServer::BeginRevoke**

COleServer::OnOpenDoc

Protected **virtual COleServerDoc* OnOpenDoc(LPCSTR** *lpszDoc* **);** ♦

lpszDoc Pointer to the filename of an existing document, which is the source of the linked item.

Remarks Called by the framework when an existing linked item is opened; that is, when the user of a client application edits a linked item. The default implementation does nothing and returns **NULL**. You must override this function if you are supporting linked items; override this function to open the document with the specified name.

The document object you create must be an object of a **COleServerDoc**-derived class. Note that this function is called only for linked items; the **OnEditDoc** member function is called for embedded items.

This function is overriden in the derived class **COleTemplateServer** to open the document with the specified name using the document-creation facilities of a **CDocTemplate** object.

Return Value If successful, a pointer to a server document; otherwise **NULL**.

See Also **COleServer::OnEditDoc**, **COleServerDoc**, **COleTemplateServer**

COleServer::Register

BOOL Register(**LPCSTR** *lpszTypeName*, **BOOL** *bMultiInstance* **);**

lpszTypeName Pointer to the name of the server document type. This must be the type name passed to **AfxOleRegisterServerName** when registering the server with the Windows registration database.

bMultiInstance Flag indicating whether multiple instances of the server application can be run simultaneously. Pass **TRUE** if your server is a single document interface (SDI) application; **TRUE** causes a separate instance of your application to run for each client. Pass **FALSE** if it is a multiple document interface (MDI) application since one instance of an MDI application can support multiple clients using separate document windows. Note that mini-servers are typically SDI applications and full servers are typically MDI applications.

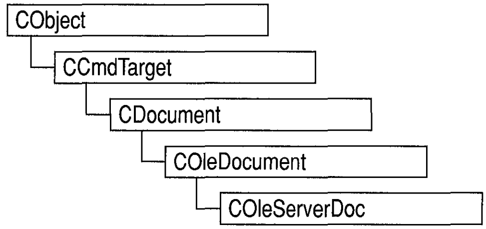
Remarks Call this function to register the server with the OLE system DLL so that it can receive requests from clients. You typically call this function for every **COleServer** object your application maintains when the application starts. The **BeginRevoke** function terminates the connection with the OLE system DLL.

Note that this operation is separate from the operation needed to create an entry for the server in the Windows registration database.

Return Value Nonzero if the server was successfully registered; otherwise 0.

See Also **COleServer::BeginRevoke**, **::OleRegisterServer**, **AfxOleRegisterServerName**

class COleServerDoc : public COleDocument



COleServerDoc is the base class for Object Linking and Embedding (OLE) server documents. A server document is a document that can contain **COleServerItem** objects, which represent the server interface to embedded or linked items. When a server application is launched by a client to edit an embedded item, the item is loaded as its own server document; the **COleServerDoc** object contains just one **COleServerItem** object, consisting of the entire document. When a server application is launched by a client to edit a linked item, an existing document is loaded from disk; this document has a portion of its contents highlighted to indicate the linked item.

Note that in server applications that support only embedding, a server document can contain only a single item. In server applications that support linking, a server document can contain zero or more linked items.

To use **COleServerDoc**, derive a class from it and implement the **OnGetEmbeddedItem** member function; this function lets your server support embedded items. Derive a class from **COleServerItem** to implement the items in your documents, and return objects of that class from **OnGetEmbeddedItem**.

To support linked items, **COleServerDoc** provides the **OnGetLinkedItem** member function. You can use the default implementation or override it if you have your own method to manage document items.

You need one **COleServerDoc**-derived class for every type of server document your application supports. For example, if your server application supports worksheets and charts, you need two **COleServerDoc**-derived classes.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **COleDocument**, **COleServer**, **COleTemplateServer**, **COleServerItem**

Construction/Destruction—Public Members

COleServerDoc	Constructs a **COleServerDoc** object.

Registration/Revocation—Public Members

RegisterServerDoc	Registers the document and informs the OLE system dynamic-link library (DLL) that it is ready for communication.
Revoke	Revokes the server document registration and waits to finish.

Operations—Public Members

NotifyRename	Notifies clients that the user has renamed the document.
NotifyRevert	Notifies clients that the user has reverted the document to the last saved state.
NotifySaved	Notifies clients that the user has saved the document.
NotifyClosed	Notifies clients that the user has closed the document.
NotifyChanged	Notifies clients that the user has changed the document.

Overridables—Protected Members

OnGetEmbeddedItem	Called to get a **COleServerItem** that represents the entire document; used to get an embedded item. Implementation required.
OnGetLinkedItem	Called to return a **COleServerItem** with the specified name; used to get a linked item.
OnClose	Called when a client requests to close the document.
OnExecute	Called when a client sends dynamic data exchange (DDE) **WM_DDE_EXECUTE** strings.
OnSetDocDimensions	Called when a client requests to change the document dimensions.
OnSetHostNames	Called when a client sets the window title for an embedded object.
OnSetColorScheme	Called when a client specifies a color palette for the document.
OnUpdateDocument	Called when a server document that is an embedded item is saved, updating the client's copy of the item.

Member Functions

COleServerDoc::COleServerDoc

COleServerDoc( );

Remarks Constructs a **COleServerDoc** object; it does not begin communications with the OLE system DLL. If your server application supports links, you must call the **RegisterServerDoc** member function; this informs clients who may be linked to the document that it is open.

See Also **COleServerDoc::RegisterServerDoc**

COleServerDoc::NotifyChanged

void NotifyChanged( );

Remarks Call this function to notify all linked items connected to the document that the document has changed. You typically call this function after the user changes some global attribute such as the dimensions of the server document. If a client item is linked to the document with an automatic link, the item is updated to reflect the changes. In client applications written with the Microsoft Foundation Class Library, the **OnChange** member function of **COleClientItem** is called. Do not call this function if the document is an embedded item.

See Also **COleServerDoc::NotifyClosed**, **COleServerDoc::NotifySaved**, **COleClientItem::OnChange**

COleServerDoc::NotifyClosed

void NotifyClosed( );

Remarks Call this function to notify the client(s) that the document has been closed. In the case where the user chooses the Close command from the File menu, **NotifyRename** is called for you by **COleServerDoc**'s implementation of the

OnCloseDocument member function. In client applications written with the Microsoft Foundation Class Library, the **OnChange** member function of **COleClientItem** is called.

See Also **COleServerDoc::NotifyChanged**, **COleServerDoc::NotifySaved**, **COleClientItem::OnChange**, **CDocument::OnCloseDocument**

COleServerDoc::NotifyRename

void NotifyRename(LPCSTR *lpszNewName* **);**

lpszNewName Pointer to a string specifying the new name of the server document; this is typically a fully qualified path.

Remarks Call this function after the user renames the server document. In the case where the user chooses the Save As command from the File menu, **NotifyRename** is called for you by **COleServerDoc**'s implementation of the **OnSaveDocument** member function. This function notifies the OLE system DLL, which in turn notifies the clients. In client applications written with the Microsoft Foundation Class Library, the **OnRenamed** member function of **COleClientItem** is called.

See Also **::OleRenameServerDoc**, **COleServerDoc::NotifySaved**, **COleClientItem::OnRenamed**, **CDocument::OnSaveDocument**

COleServerDoc::NotifyRevert

void NotifyRevert( );

Remarks Call this function to inform the OLE system DLL that the server has restored a document to its last saved state without closing it; the OLE system DLL notifies the clients. You typically call this function after the user reverts a server document to its last saved form. The framework calls this function in **COleServerDoc**'s implementation of the **OnCloseDocument** member function if the document has been modified.

See Also **::OleRevertServerDoc**, **COleServerDoc::NotifyRename**, **COleServerDoc::NotifySaved**, **CDocument::OnCloseDocument**

COleServerDoc::NotifySaved

void NotifySaved( );

Remarks

Call this function after the user saves the server document. In the case where the user chooses the Save command from the File menu, **NotifySaved** is called for you by **COleServerDoc**'s implementation of **OnSaveDocument**. This function notifies the OLE system DLL, which in turn notifies the clients. In client applications written with the Microsoft Foundation Class Library, the **OnChanged** member function of **COleClientItem** is called.

See Also

::OleSavedServerDoc, COleServerDoc::NotifyChanged, COleServerDoc::NotifyClosed, COleClientItem::OnChange, CDocument::OnSaveDocument

COleServerDoc::OnClose

Protected

virtual OLESTATUS OnClose( ); ♦

Remarks

Called by the framework when a client requests that the server document be closed.

Return Value

OLE_OK if successful; any other value indicates failure. See the **COleException** class for a list of possible values.

COleServerDoc::OnExecute

Protected

virtual OLESTATUS OnExecute(LPVOID *lpCommands* **);** ♦

lpCommands Points to a block of memory that contains dynamic data exchange (DDE) **WN_DDE_EXECUTE** command strings.

Remarks

Called by the framework when the client sends DDE **WN_DDE_EXECUTE** command strings to the document. The default implementation does nothing and returns **OLE_ERROR_COMMAND**. Override this function to handle DDE **WN_DDE_EXECUTE** commands. Do not delete memory referenced by *lpCommands*.

Return Value **OLE_OK** if successful; any other value indicates failure. See the **COleException** class for a list of possible values.

See Also **COleServer::OnExecute**

COleServerDoc::OnGetEmbeddedItem

Protected **virtual COleServerItem* OnGetEmbeddedItem( ) = 0;** ♦

Remarks Called by the framework when a client application invokes the server application to create or edit an embedded item. There is no default implementation. You must override this function to return an item representing the entire document. This should be an instance of a **COleServerItem**-derived class.

Return Value A pointer to an item representing the entire document; **NULL** if the operation failed.

See Also **COleServerDoc::OnGetLinkedItem**, **COleServerItem**

COleServerDoc::OnGetLinkedItem

Protected **virtual COleServerItem* OnGetLinkedItem(LPCSTR** *lpszItemName* **);** ♦

lpszItemName The name of an existing linked item.

Remarks Called by the framework when a client application invokes the server application to edit a linked item. The default implementation searches for the item with the specified name in the collection of items contained in the document. Override this function if you want to implement your own method of storing or retrieving linked items. The **OnGetLinkedItem** function is called only for documents that support links. If the document is an embedded item, the function should return **NULL**.

Return Value A pointer to the specified item; **NULL** if the item is not found.

See Also **COleServerDoc::OnGetEmbeddedItem**, **COleServerItem**

COleServerDoc::OnSetColorScheme

Protected

virtual OLESTATUS OnSetColorScheme(const LOGPALETTE FAR* *lpLogPalette* **);** ♦

lpLogPalette Pointer to a Windows **LOGPALETTE** structure.

Remarks

Called by the framework when a client sets the color palette for this server document. The default implementation does nothing. Override this function if you want to use the color palette specified by the client. See **COleClientItem::SetColorScheme** for information on how your server should interpret the colors in the palette.

Return Value

OLE_OK if successful; any other value indicates failure. See the **COleException** class for a list of possible values.

See Also

COleClientItem::SetColorScheme

COleServerDoc::OnSetDocDimensions

Protected

virtual OLESTATUS OnSetDocDimensions(LPCRECT *lpRect* **);** ♦

lpRect A pointer to a **RECT** structure that contains the new window dimensions.

Remarks

Called by the framework when a client changes the size of the server's document window. The default implementation does nothing and returns **OLE_OK**. Override this function if your server can resize or move its document windows. This function is called only for documents that are embedded items.

Return Value

OLE_OK if successful; any other value indicates failure. See the **COleException** class for a list of possible values.

See Also

COleClientItem::SetBounds

COleServerDoc::OnSetHostNames

Protected **virtual OLESTATUS OnSetHostNames(LPCSTR** *lpszHost*, **LPCSTR** *lpszHostObj* **);** ♦

lpszHost Pointer to a string that specifies the name of the client application.

lpszHostObj Pointer to a string that specifies the client's name for the document.

Remarks Called by the framework when the client sets or changes the host names for this item. The default implementation does nothing and returns **OLE_OK**. Override this function if you need to save these names.

Return Value **OLE_OK** if successful; any other value indicates failure. See the **COleException** class for a list of possible values.

See Also **COleClientItem::SetHostNames**

COleServerDoc::OnUpdateDocument

Protected **virtual BOOL OnUpdateDocument( );** ♦

Remarks Called by the framework when saving a document that is an embedded item, that is, when updating an item in a compound document. The default implementation calls the **NotifySaved** member function and then marks the document as clean. Override this function if you want to perform special processing when updating an embedded item.

Return Value Nonzero if the document was successfully updated; otherwise 0.

See Also **COleServerDoc::NotifySaved**, **CDocument::OnSaveDocument**

COleServerDoc::RegisterServerDoc

BOOL RegisterServerDoc(COleServer* *pServer*, **LPCSTR** *lpszDoc* **);**

pServer Pointer to an OLE server that is already registered.

lpszDoc Pointer to the fully qualified path of the server document.

Remarks

Call this function to register the document with the OLE system DLL. You need to call this function only if your server application supports links; this registration lets clients know that the document is open. Call this function when creating or opening a named file; however, if you are using **COleTemplateServer** to implement your server, **RegisterServerDoc** is called for you by **COleServerDoc**'s implementation of **OnNewDocument** or **OnOpenDocument**, respectively. There is no need to call this function if the document represents an embedded item.

Return Value

Nonzero if the document was successfully registered; otherwise 0.

See Also

COleServer, **COleTemplateServer**, **CDocument::OnNewDocument**, **CDocument::OnOpenDocument**

COleServerDoc::Revoke

void Revoke( );

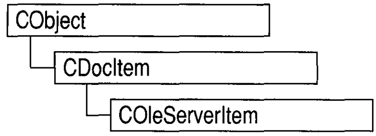
Remarks

Revokes, or shuts down, the server document and waits for any pending operation to finish. The **Revoke** member function is called by the **COleServerDoc** destructor; it is seldom called explicitly elsewhere.

See Also

::OleRevokeServerDoc

class COleServerItem : public CDocItem



The **COleServerItem** class provides the server interface to Object Linking and Embedding (OLE) items. A linked item can represent some or all of a server document. An embedded item always represents an entire server document.

The **COleServerItem** class defines several overridable member functions that are called by the OLE system dynamic-link library (DLL), usually in response to requests from the client application. These member functions allow the client application to indirectly manipulate the item in various ways, such as displaying it, executing its verbs, or retrieving its data in various formats.

To use **COleServerItem**, derive a class from it and implement the **OnDraw** and **Serialize** member functions. The **OnDraw** function provides the metafile representation of an item, allowing it to be displayed when a client application opens a compound document. The **Serialize** function of **CObject** provides the Native representation of an item, allowing an embedded item to be transferred between the server and client applications.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **COleClientItem**, **COleServer**, **COleServerDoc**, **COleTemplateServer**, **CObject::Serialize**

Status—Public Members

GetDocument	Returns the server document that contains the item.
GetItemName	Returns the name of the item. Used for linked items only.
SetItemName	Sets the name of the item. Used for linked items only.

Operations—Public Members

CopyToClipboard	Copies the item to the Clipboard.
NotifyChanged	Updates all clients with automatic link update.
Revoke	Terminates the connection between the item and the OLE system DLL.

Construction/Destruction—Protected Members

COleServerItem	Constructs a **COleServerItem** object.

Status—Protected Members

IsConnected	Indicates whether the item is currently attached to an active client.

Overridables—Protected Members

OnShow	Called when the client requests to show the item.
OnDraw	Called when the client requests to draw the item; implementation required.
OnExtraVerb	Called to execute verbs other than the primary verb.
OnSetTargetDevice	Called to set the item's target device.
OnSetBounds	Called to set the item's bounding rectangle.
OnGetTextData	Called to get item data as a text string.
OnSetColorScheme	Called to set the item's color scheme.
OnEnumFormats	Called to enumerate available data formats.
OnGetData	Called to retrieve the item's data.
OnSetData	Called to set the item's data.
OnDoVerb	Called to execute the primary verb.

Member Functions

COleServerItem::COleServerItem

Protected **COleServerItem(COleServerDoc*** *pContainerDoc* **);** ♦

pContainerDoc Pointer to the document that contains the item.

Remarks Constructs a **COleServerItem** object and adds it to the container document's collection of document items.

See Also **COleDocument::AddItem**

COleServerItem::CopyToClipboard

BOOL CopyToClipboard(BOOL *bIncludeNative*, **BOOL** *bIncludeLink* **);**

bIncludeNative Set this to **TRUE** if Native data should be copied to the Clipboard. Set this to **FALSE** if your server application supports only links (this is rare).

bIncludeLink Set this to **TRUE** if ObjectLink data should be copied to the Clipboard. Set this to **FALSE** if your server application does not support links.

Remarks

Call this function to copy the item to the Clipboard. The function first copies the item to the Clipboard using the formats returned by the **OnEnumFormats** member function. These typically include Native format followed by presentation formats. This causes the **Serialize**, **OnDraw**, and **OnGetTextData** member functions to be called. The function then checks whether the document containing the item is connected to a server; if so, the function copies OwnerLink format and, if specified, ObjectLink format.

Return Value

Nonzero if the item was successfully copied to the Clipboard; otherwise 0.

See Also

COleClientItem::CopyToClipboard, **COleServerItem::OnEnumFormats**, **COleServerItem::OnDraw**, **COleServerItem::OnGetTextData**, **CObject::Serialize**

COleServerItem::GetDocument

COleServerDoc* GetDocument( ) const;

Remarks

Call this function to get a pointer to the document that contains the item. This allows access to the server document that you passed as an argument to the **COleServerItem** constructor.

Return Value

A pointer to the document that contains the item, **NULL** if the item is not part of a document.

See Also

COleServerItem::COleServerItem, **COleServerDoc**

COleServerItem::GetItemName

const CString& GetItemName( ) const;

Remarks Call this function to get the name of the item. You typically call this function only for linked items.

Return Value The name of the item.

See Also **COleServerItem::SetItemName**, **COleServerDoc::OnGetLinkedItem**

COleServerItem::IsConnected

Protected **BOOL IsConnected( ) const;** ♦

Remarks Call this function to determine if the item is connected to its corresponding client item.

Return Value Nonzero if the item is connected; otherwise 0.

COleServerItem::NotifyChanged

void NotifyChanged( );

Remarks Call this function after the linked item has been changed. If a client item is linked to the document with an automatic link, the item is updated to reflect the changes. In client applications written with the Microsoft Foundation Class Library, the **OnChange** member function of **COleClientItem** is called in response.

See Also **COleClientItem::OnChange**, **COleServerDoc::NotifyChanged**

COleServerItem::OnDoVerb

Protected

virtual OLESTATUS OnDoVerb(UINT *nVerb*, **BOOL** *bShow*,
BOOL *bTakeFocus* **);** ♦

nVerb Server verb index; 0 is the primary index, 1 is the secondary index, and so forth.

bShow **TRUE** if the server should show the item when it performs the operation.

bTakeFocus **TRUE** if the server should set the input focus.

Remarks

Called by the framework when the **COleClientItem::Activate** function is called. The default implementation calls the **OnShow** member function for the primary verb if *bShow* is **TRUE** and calls **OnExtraVerb** for nonprimary verbs. Override this function if your primary verb does not show the item. For example, suppose the item were a sound recording and its primary verb were Play; in this case, you would not have to display the server application to play back the item.

Return Value

See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also

COleClientItem::Activate, **COleServerItem::OnShow**, **COleServerItem::OnExtraVerb**

COleServerItem::OnDraw

Protected

virtual BOOL OnDraw(CDC* *pDC* **) = 0;** ♦

pDC A pointer to the **CDC** object on which to draw the item. This is an output-only **CMetafileDC** object; do not call any attribute member functions of **CDC** for this parameter.

Remarks

Called by the framework to render the item into a metafile. The metafile representation of the item is used by the **COleClientItem::Draw** function to display the item in the client application. There is no default implementation. You must override this function to draw the item into the device context specified.

Return Value

Nonzero if the item was successfully drawn; otherwise 0.

See Also

COleServerItem::OnGetData, **COleClientItem::Draw**

COleServerItem::OnEnumFormats

Protected **virtual OLECLIPFORMAT OnEnumFormats(OLECLIPFORMAT** *nFormat* **) const;** ♦

nFormat Specifies the format returned by the previous call to the **OnEnumFormats** member function. For the first call to **OnEnumFormats**, this parameter is **NULL**. This parameter can be one of the predefined Clipboard formats or the value returned by the Windows **RegisterClipboardFormat** function.

Remarks Called by the framework to determine what formats are available for the item. This is called in response to the **COleClientItem::EnumFormats** function; it is also called by the OLE system DLL. When called iteratively, this function returns all the Clipboard formats that are supported by this server. The default implementation returns Native, **CF_METAFILEPICT**, and **CF_TEXT** formats. Override this function if you want to specify the formats supported by your server; for example, if you wanted to support the Paste Special command in client applications. Note that if you want to support the **CF_TEXT** format, you must override the **OnGetTextData** member function.

Return Value The next (or first) available format; **NULL** if no more formats are available.

See Also **COleClientItem::EnumFormats**, **COleServerItem::OnGetTextData**, **::OleEnumFormats**, **::RegisterClipboardFormats**

COleServerItem::OnExtraVerb

Protected **virtual OLESTATUS OnExtraVerb(UINT** *nVerb* **);** ♦

nVerb Index of the verb to execute; 1 is the secondary verb, 2 is the tertiary verb, and so forth.

Remarks Called by the framework when a client makes a request to execute a nonprimary verb. The default implementation returns **OLE_ERROR_DOVERB**. Override this function if the item supports more than one verb. You must provide the names of all supported verbs to the client applications through the Windows registration database.

Return Value See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also **COleServerItem::OnShow**, **COleServerItem::OnDoVerb**, **COleClientItem::Activate**

COleServerItem::OnGetData

Protected

virtual OLESTATUS OnGetData(OLECLIPFORMAT *nFormat*, **LPHANDLE** *lphReturn* **);** ♦

nFormat Specifies the format of the data. This parameter can be one of the predefined Clipboard formats or the value returned by the Windows **RegisterClipboardFormats** function.

lphReturn Pointer to a handle to the block of memory that contains the requested data when the function returns.

Remarks

Called by the framework to retrieve the contents of the item in a specified format. This is called in response to the **COleClientItem::GetData** function. The default implementation supports Native and metafile formats; it uses the implementations of the **Serialize** and **OnDraw** member functions that you provide. This function also supports the **CF_TEXT** format if you have overridden the **OnGetTextData** member function. Override this function if you want to handle other formats. Allocate a memory object, fill it with the data in the desired format, and return it via the *lphReturn* parameter. This is an advanced overridable.

Return Value

See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also

COleServerItem::OnGetTextData, **COleServerItem::OnSetData**, **COleServerItem::OnDraw**, **CObject::Serialize**, **COleClientItem::GetData**

COleServerItem::OnGetTextData

Protected

virtual BOOL OnGetTextData(CString& *rStringReturn* **) const;** ♦

rStringReturn A reference to a **CString** that receives the text data when the function returns.

Remarks

Called by the framework to get the contents of the item in text (**CF_TEXT**) format. The default implementation returns **FALSE**. Override this function if the item can return its data in text form.

Return Value

Nonzero if text data is supported; otherwise 0.

COleServerItem::OnSetBounds

Protected **virtual OLESTATUS OnSetBounds(LPCRECT** *lpRect* **);** ♦

lpRect A pointer to a **RECT** structure specifying the new bounding rectangle.

Remarks Called by the framework when the **COleClientItem::SetBounds** function is called. The default implementation updates the item's bounding rectangle with the specified rectangle. Override this function to perform special processing when you change the bounding rectangle for the item.

Return Value See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also **COleClientItem::SetBounds**

COleServerItem::OnSetColorScheme

Protected **virtual OLESTATUS OnSetColorScheme(const LOGPALETTE FAR*** *lpLogPalette* **);** ♦

lpLogPalette Pointer to a Windows **LOGPALETTE** structure.

Remarks Called by the framework when the **COleClientItem::SetColorScheme** function is called. The default implementation does nothing. Override this function if you want to use the recommended palette.

Return Value See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also **COleClientItem::SetColorScheme**

COleServerItem::OnSetData

Protected

virtual OLESTATUS OnSetData(OLECLIPFORMAT *nFormat*, **HANDLE** *hData* **);** ♦

nFormat Specifies the format of the data. This parameter can be one of the predefined Clipboard formats or the value returned by the Windows **RegisterClipboardFormats** function.

hData Handle to a memory object that contains the data in the specified format.

Remarks

Called by the framework to provide the server application with the data for the item, typically when an embedded item is opened for editing. This is called in response to the **COleClientItem::SetData** function; it is also called by the OLE system DLLs. The default implementation handles only Native format; it calls the **Serialize** member function to load the contents of the specified block of memory into the item. Override this function to process non-Native formats. You must free the memory object after you have used it.

Return Value

See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also

COleClientItem::SetData, **COleServerItem::OnGetData**, **CObject::Serialize**

COleServerItem::OnSetTargetDevice

Protected

virtual OLESTATUS OnSetTargetDevice(LPOLETARGETDEVICE *lpTargetDevice* **);** ♦

lpTargetDevice Points to a Windows **OLETARGETDEVICE** structure that describes the target device for the item. If **NULL**, the target device is the video display. Do not free this structure after you have used it.

Remarks

Called by the framework to provide the server application with information about the client application's target device for the item. This is called in response to the **COleClientItem::SetTargetDevice** function. The default implementation does nothing. Override this function if you want to know what kind of device the item will be rendered on. You can use this information to optimize the format of the information that you supply the client through the **OnGetData** member function.

Return Value See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also **COleServerItem::OnGetData**, **COleClientItem::SetTargetDevice**

COleServerItem::OnShow

Protected **virtual OLESTATUS OnShow(BOOL** *bTakeFocus* **);** ♦

bTakeFocus **TRUE** if the item should take the input focus; otherwise **FALSE**.

Remarks Called by the framework to instruct the server application to display the item. This function is typically called when the user of the client application creates an item or executes a verb, such as Edit, that requires the item to be shown. The default implementation activates the first frame window displaying the document that contains the item and, if *bTakeFocus* is **TRUE**, gives the window the focus. Override this function to make the item visible in the window (for example, by scrolling) and to select the item, if possible.

Return Value See the **COleException** class for a list of return values. The value **OLE_OK** indicates that the function operated correctly.

See Also **COleServerItem::OnDoVerb**, **COleServerItem::OnExtraVerb**, **COleClientItem::Activate**

COleServerItem::Revoke

void Revoke( );

Remarks Call this function to revoke the client's access to the item. You should call this function when the user of the server application destroys an item. This function does not return until the revoke operation is complete, but it allows other messages to be processed while waiting.

See Also **::OleRevokeObject**

COleServerItem::SetItemName

void SetItemName(const char* *pszItemName* **);**

pszItemName Pointer to the new name of the item.

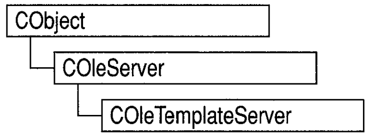
Remarks

Call this function to set the name of the item. You should call this function when you create a linked item; the name must be unique within the document. When a server application is invoked to edit a linked item, the application uses this name to find the item. You do not need to call this function for embedded items.

See Also

COleServerItem::GetItemName, **COleServerDoc::OnGetLinkedItem**

class COleTemplateServer : public COleServer



The **COleTemplateServer** class defines an Object Linking and Embedding (OLE) server. It is derived from the abstract class **COleServer**; however, you can use **COleTemplateServer** directly rather than having to derive a class. **COleTemplateServer** uses a **CDocTemplate** object to manage the server documents. Use **COleTemplateServer** when implementing a full server, that is, a server that can be run as a stand-alone application. Full servers are typically multiple document interface (MDI) applications, although single document interface (SDI) applications are supported. One **COleTemplateServer** object is needed for each type of server document an application supports; that is, if your server application supports both worksheets and charts, you must have two **COleTemplateServer** objects.

COleTemplateServer overrides the **OnCreateDoc**, **OnEditDoc**, and **OnOpenDoc** member functions defined by **COleServer**. These member functions are called by the OLE system dynamic-link library (DLL) in response to requests from client applications. Through these member functions, the OLE system DLL instructs the server to open embedded items as documents or open the documents that are the source of linked items. See the descriptions for these functions under **COleServer** for more information on when these member functions are called.

COleTemplateServer implements these member functions by using the document-creation facilities of its associated **CDocTemplate** object. This lets your server application take advantage of the document/view architecture provided by the Microsoft Foundation Class Library. To use **COleTemplateServer**, create a **CDocTemplate** object, specifying a **COleServerDoc**-derived class as the document class, and add it to your application by passing it to the **AddDocTemplate** member function of **CWinApp**. To execute the server, pass the document template to the **RunEmbedded** member function of **COleTemplateServer**.

Note The OLE documentation for Windows version 3.1 refers to embedded and linked items as "objects" and refers to types of items as "classes." This reference uses the term "item" to distinguish the OLE entity from the corresponding C++ object and the term "type" to distinguish the OLE category from the C++ class.

#include <afxole.h>

See Also **CDocTemplate**, **COleServer**, **COleServerDoc**, **COleServerItem**

Construction/Destruction — Public Members

COleTemplateServer	Constructs a **COleTemplateServer** object.

Operations — Public Members

RunEmbedded	Launches the server in embedded mode.

Member Functions

COleTemplateServer::COleTemplateServer

COleTemplateServer( );

Remarks Constructs a **COleTemplateServer** object. Call the **RunEmbedded** member function to run the server.

See Also **COleTemplateServer::RunEmbedded**

COleTemplateServer::RunEmbedded

BOOL RunEmbedded(CDocTemplate* *pDocTemplate*,
BOOL *bMultiInstance*, **LPCSTR** *lpszCmdLine* **);**

pDocTemplate Pointer to a **CDocTemplate** object describing the document type. The document class should be derived from **COleServerDoc**.

bMultiInstance Flag indicating whether multiple instances of the server application can be run simultaneously. Pass **TRUE** if your server is an SDI application; **TRUE** causes a separate instance of your application to run for each client. Pass **FALSE** if it is an MDI application since one instance of an MDI application can support multiple clients using separate document windows. Note that mini servers are typically SDI applications and full servers are typically MDI applications.

lpszCmdLine Pointer to the command line.

Remarks

Call this function from the **InitInstance** member function of your **CWinApp**-derived application class, passing the command line from **CWinApp::m_lpCmdLine**. This function parses the command line to see whether the "/Embedding" or "-Embedding" option is present; either option indicates that the server application was launched by a client application. The function then registers the server with the OLE system DLL so that it can receive requests from clients. (If the application was launched as a stand-alone application, the function registers the server application with the Windows registration database before registering it with the OLE system DLL.) If a filename appeared after the "/Embedding" or "-Embedding" option (referring to the source of a linked item), the function then opens the specified file.

Return Value

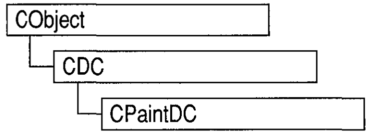
Nonzero if the server was launched successfully; otherwise 0.

See Also

COleServer::Register, **CDocTemplate**, **COleServerDoc**, **CWinApp::InitInstance**, **CWinApp::m_lpCmdLine**

class CPaintDC : public CDC

The **CPaintDC** class is a device-context class derived from **CDC**. It performs a **CWnd::BeginPaint** at construction time and **CWnd::EndPaint** at destruction time. A **CPaintDC** object can only be used when responding to a **WM_PAINT** message, usually in your **OnPaint** message-handler member function.



#include <afxwin.h>

See Also **CDC**

Data Members—Public Members

m_ps Contains the **PAINTSTRUCT** used to paint the client area.

Construction/Destruction—Public Members

CPaintDC Constructs a **CPaintDC** connected to the specified **CWnd**.

Data Members—Protected Members

m_hWnd The **HWND** to which this **CPaintDC** object is attached.

Member Functions

CPaintDC::CPaintDC

CPaintDC(CWnd* *pWnd* **)**
throw(CResourceException);

pWnd Points to the **CWnd** object to which the **CPaintDC** object belongs.

Remarks

Constructs a **CPaintDC** object, prepares the application window for painting, and stores the **PAINTSTRUCT** structure in the **m_ps** member variable. An exception (of type **CResourceException**) is thrown if the Windows **GetDC** call fails. A device context may not be available if Windows has already allocated all of its available device contexts. Your application competes for the five common display contexts available at any given time under the Windows operating system.

Data Members

CPaintDC::m_hWnd

Remarks

The **HWND** to which this **CPaintDC** object is attached. **m_hWnd** is a protected variable of type **HWND**.

CPaintDC::m_ps

Remarks

m_ps is a public member variable of type **PAINTSTRUCT**. It is the **PAINTSTRUCT** that is passed to and filled out by **CWnd::BeginPaint**. The **PAINTSTRUCT** contains information that the application uses to paint the client area of the window associated with a **CPaintDC** object. Note that you can access the device-context handle through the **PAINTSTRUCT**. However, you can access the handle more directly through the **m_hDC** member variable that **CPaintDC** inherits from **CDC**.

PAINTSTRUCT Structure

The **PAINTSTRUCT** structure looks like this:

```
typedef struct tagPAINTSTRUCT {
   HDC  hdc;
   BOOL fErase;
   RECT rcPaint;
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved[16];
} PAINTSTRUCT;
```

The **PAINTSTRUCT** structure contains information that can be used to paint the client area of a window.

Members

hdc Identifies the display context to be used for painting.

fErase Specifies whether the background needs to be redrawn. It is not 0 if the application should redraw the background. The application is responsible for drawing the background if a Windows window-class is created without a background brush (see the description of the **hbrBackground** member of the **WNDCLASS** structure).

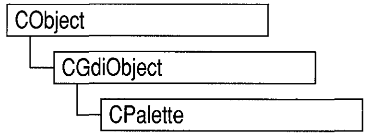
rcPaint Specifies the upper-left and lower-right corners of the rectangle in which the painting is requested.

fRestore Reserved member. It is used internally by Windows.

fIncUpdate Reserved member. It is used internally by Windows.

rgbReserved[16] Reserved member. A reserved block of memory used internally by Windows.

class CPalette : public CGdiObject



The **CPalette** class encapsulates a Windows color palette. A palette provides an interface between an application and a color output device (such as a display device). The interface allows the application to take full advantage of the color capabilities of the output device without severely interfering with the colors displayed by other applications. The Windows operating system uses the application's logical palette (a list of needed colors) and the system palette (which defines available colors) to determine the colors used.

A **CPalette** object provides member functions for manipulating the palette referred to by the object. Construct a **CPalette** object and use its member functions to create the actual palette, a graphics device interface (GDI) object, and to manipulate its entries and other properties.

#include <afxwin.h>

Construction/Destruction—Public Members

CPalette	Constructs a **CPalette** object with no attached Windows palette. You must initialize the **CPalette** object with one of the other member functions before it can be used.

Initialization—Public Members

CreatePalette	Initializes a **CPalette** object by creating a Windows color palette and attaching the palette to the **CPalette** object.

Operations—Public Members

FromHandle	Returns a pointer to a **CPalette** object when given a handle to a Windows palette object. If a **CPalette** object is not already attached to the Windows palette, a temporary **CPalette** object is created and attached.
GetPaletteEntries	Retrieves a range of palette entries in a logical palette.
SetPaletteEntries	Sets RGB color values and flags in a range of entries in a logical palette.

AnimatePalette	Replaces entries in the logical palette identified by the **CPalette** object. The application does not have to update its client area because the Windows operating system maps the new entries into the system palette immediately.
GetNearestPaletteIndex	Returns the index of the entry in the logical palette that most closely matches a color value.
ResizePalette	Changes the size of the logical palette specified by the **CPalette** object to the specified number of entries.

Member Functions

CPalette::AnimatePalette

void AnimatePalette(UINT *nStartIndex*, **UINT** *nNumEntries*, **LPPALETTEENTRY** *lpPaletteColors* **);**

nStartIndex Specifies the first entry in the palette to be animated.

nNumEntries Specifies the number of entries in the palette to be animated.

lpPaletteColors Points to the first member of an array of **PALETTEENTRY** structures to replace the palette entries identified by *nStartIndex* and *nNumEntries*.

Remarks

Replaces entries in the logical palette attached to the **CPalette** object. When an application calls **AnimatePalette**, it does not have to update its client area because Windows maps the new entries into the system palette immediately. The **AnimatePalette** function will only change entries with the **PC_RESERVED** flag set in the corresponding **palPaletteEntry** member of the **LOGPALETTE** structure that is attached to the **CPalette** object.

See Also

CPalette::CreatePalette, **::AnimatePalette**

CPalette::CPalette

CPalette( );

Remarks Constructs a **CPalette** object. The object has no attached palette until you call **CreatePalette** to attach one.

See Also **CPalette::CreatePalette**

CPalette::CreatePalette

BOOL CreatePalette(LPLOGPALETTE *lpLogPalette* **);**

lpLogPalette Points to a **LOGPALETTE** structure that contains information about the colors in the logical palette.

The **LOGPALETTE** structure has the following form:

```
typedef struct tagLOGPALETTE {
    WORD         palVersion;
    WORD         palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

y

Remarks Initializes a **CPalette** object by creating a Windows logical color palette and attaching it to the **CPalette** object.

Return Value Nonzero if successful; otherwise 0.

See Also **::CreatePalette**

CPalette::FromHandle

static CPalette* PASCAL FromHandle(HPALETTE *hPalette* **);**

hPalette A handle to a Windows GDI color palette.

Remarks Returns a pointer to a **CPalette** object when given a handle to a Windows palette object. If a **CPalette** object is not already attached to the Windows palette, a temporary **CPalette** object is created and attached. This temporary **CPalette** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

Return Value A pointer to a **CPalette** object if successful; otherwise **NULL**.

CPalette::GetNearestPaletteIndex

UINT GetNearestPaletteIndex(COLORREF *crColor* **) const;**

crColor Specifies the color to be matched.

Remarks Returns the index of the entry in the logical palette that most closely matches the specified color value.

Return Value The index of an entry in a logical palette. The entry contains the color that most nearly matches the specified color.

See Also **::GetNearestPaletteIndex**

CPalette::GetPaletteEntries

UINT GetPaletteEntries(UINT *nStartIndex*, **UINT** *nNumEntries*, **LPPALETTEENTRY** *lpPaletteColors* **) const;**

nStartIndex Specifies the first entry in the logical palette to be retrieved.

nNumEntries Specifies the number of entries in the logical palette to be retrieved.

lpPaletteColors Points to an array of **PALETTEENTRY** data structures to receive the palette entries. The array must contain at least as many data structures as specified by *nNumEntries*.

Remarks Retrieves a range of palette entries in a logical palette.

Return Value The number of entries retrieved from the logical palette; 0 if the function failed.

See Also **::GetPaletteEntries**

CPalette::ResizePalette

BOOL ResizePalette(UINT *nNumEntries* **);**

nNumEntries Specifies the number of entries in the palette after it has been resized.

Remarks Changes the size of the logical palette attached to the **CPalette** object to the number of entries specified by *nNumEntries*. If an application calls **ResizePalette** to reduce the size of the palette, the entries remaining in the resized palette are unchanged. If the application calls **ResizePalette** to enlarge the palette, the additional palette entries are set to black (the red, green, and blue values are all 0), and the flags for all additional entries are set to 0.

Return Value Nonzero if the palette was successfully resized; otherwise 0.

See Also **::ResizePalette**

CPalette::SetPaletteEntries

UINT SetPaletteEntries(UINT *nStartIndex*, **UINT** *nNumEntries*, **LPPALETTEENTRY** *lpPaletteColors* **);**

nStartIndex Specifies the first entry in the logical palette to be set.

nNumEntries Specifies the number of entries in the logical palette to be set.

lpPaletteColors Points to an array of **PALETTEENTRY** data structures to receive the palette entries. The array must contain at least as many data structures as specified by *nNumEntries*.

Remarks Sets RGB color values and flags in a range of entries in a logical palette. If the logical palette is selected into a device context when the application calls **SetPaletteEntries**, the changes will not take effect until the application calls **CDC::RealizePalette**.

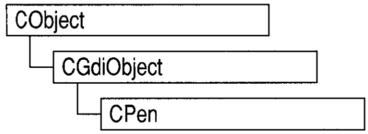
Return Value The number of entries set in the logical palette; 0 if the function failed.

See Also **CDC::RealizePalette**, **::SetPaletteEntries**

class CPen : public CGdiObject

The **CPen** class encapsulates a Windows graphics device interface (GDI) pen.

#include <afxwin.h>



Construction/Destruction — Public Members

CPen	Constructs a **CPen** object.

Initialization — Public Members

CreatePen	Initializes a pen with the specified style, width, and color.
CreatePenIndirect	Initializes a pen with the style, width, and color given in a **LOGPEN** structure.

Operations — Public Members

FromHandle	Returns a pointer to a **CPen** object when given a Windows **HPEN**.

Member Functions

CPen::CPen

CPen( );

CPen(int *nPenStyle*, **int** *nWidth*, **COLORREF** *crColor*)
throw(CResourceException);

nPenStyle   Specifies the pen style. This parameter can be one of the following values:

- **PS_SOLID**   Creates a solid pen.
- **PS_DASH**   Creates a dashed pen. Valid only when the pen width is 1.
- **PS_DOT**   Creates a dotted pen. Valid only when the pen width is 1.
- **PS_DASHDOT**   Creates a pen with alternating dashes and dots. Valid only when the pen width is 1.

- **PS_DASHDOTDOT** Creates a pen with alternating dashes and double dots. Valid only when the pen width is 1.
- **PS_NULL** Creates a null pen.
- **PS_INSIDEFRAME** Creates a pen that draws a line inside the frame of closed shapes produced by the Windows GDI output functions that specify a bounding rectangle (for example, the **Ellipse**, **Rectangle**, **RoundRect**, **Pie**, and **Chord** member functions). When this style is used with Windows GDI output functions that do not specify a bounding rectangle (for example, the **LineTo** member function), the drawing area of the pen is not limited by a frame.

nWidth Specifies the width, in logical units, of the pen. If this value is 0, the width in device units is always 1 pixel, regardless of the mapping mode.

crColor Contains an RGB color for the pen.

Remarks

If you use the constructor with no arguments, you must initialize the resulting **CPen** object with the **CreatePen**, **CreatePenIndirect**, or **CreateStockObject** member functions. If you use the constructor that takes arguments, then no further initialization is necessary. The constructor with arguments can throw an exception if errors are encountered, while the constructor with no arguments will always succeed.

See Also

CPen::CreatePen, **CPen::CreatePenIndirect**, **CGdiObject::CreateStockObject**

CPen::CreatePen

BOOL CreatePen(int *nPenStyle*, **int** *nWidth*, **COLORREF** *crColor* **);**

nPenStyle Specifies the style for the pen. For a list of possible values, see the *nPenStyle* parameter to the **CPen** constructor.

nWidth Specifies the width of the pen (in logical units). If this value is 0, the width in device units is always 1 pixel, regardless of the mapping mode.

crColor Contains an RGB color for the pen.

Remarks

Initializes a pen with the specified style, width, and color. The pen can be subsequently selected as the current pen for any device context. Pens that have a width greater than 1 pixel should always have either the **PS_NULL**, **PS_SOLID**, or **PS_INSIDEFRAME** style. If a pen has the **PS_INSIDEFRAME** style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The **PS_SOLID** pen style cannot be used to create a pen with a dithered color. The style **PS_INSIDEFRAME** is identical to **PS_SOLID** if the pen width is less than or equal to 1.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CPen::CreatePenIndirect**, **CPen::CPen**

CPen::CreatePenIndirect

BOOL CreatePenIndirect(LPLOGPEN *lpLogPen* **);**

lpLogPen Points to the Windows **LOGPEN** structure that contains information about the pen.

Remarks Initializes a pen that has the style, width, and color given in the structure pointed to by *lpLogPen*. Pens that have a width greater than 1 pixel should always have either the **PS_NULL**, **PS_SOLID**, or **PS_INSIDEFRAME** style. If a pen has the **PS_INSIDEFRAME** style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The **PS_INSIDEFRAME** style is identical to **PS_SOLID** if the pen width is less than or equal to 1.

Return Value Nonzero if the function is successful; otherwise 0.

LOGPEN Structure A **LOGPEN** structure has this form:

```
typedef struct tagLOGPEN {  /* lgpn */
    UINT     lopnStyle;
    POINT    lopnWidth;
    COLORREF lopnColor;
} LOGPEN;
```

The **LOGPEN** structure defines the style, width, and color of a pen, a drawing object used to draw lines and borders. The **CreatePenIndirect** function uses the **LOGPEN** structure.

Members **lopnStyle** Specifies the pen type. This member can be one of the following values:

- **PS_SOLID** Creates a solid pen.
- **PS_DASH** Creates a dashed pen. (Valid only when the pen width is 1.)
- **PS_DOT** Creates a dotted pen. (Valid only when the pen width is 1.)
- **PS_DASHDOT** Creates a pen with alternating dashes and dots. (Valid only when the pen width is 1.)

- **PS_DASHDOTDOT** Creates a pen with alternating dashes and double dots. (Valid only when the pen width is 1.)
- **PS_NULL** Creates a null pen.
- **PS_INSIDEFRAME** Creates a pen that draws a line inside the frame of closed shapes produced by GDI output functions that specify a bounding rectangle (for example, the **Ellipse**, **Rectangle**, **RoundRect**, **Pie**, and **Chord** member functions). When this style is used with GDI output functions that do not specify a bounding rectangle (for example, the **LineTo** member function), the drawing area of the pen is not limited by a frame.

  If a pen has the **PS_INSIDEFRAME** style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The **PS_SOLID** pen style cannot be used to create a pen with a dithered color. The **PS_INSIDEFRAME** style is identical to **PS_SOLID** if the pen width is less than or equal to 1.

  When the **PS_INSIDEFRAME** style is used with GDI objects produced by functions other than **Ellipse**, **Rectangle**, and **RoundRect**, the line may not be completely inside the specified frame.

lopnWidth Specifies the pen width, in logical units. If the **lopnWidth** member is 0, the pen is 1 pixel wide on raster devices regardless of the current mapping mode.

lopnColor Specifies the pen color.

Comments The *y* value in the **POINT** structure for the **lopnWidth** member is not used.

See Also **CPen::CreatePen**, **CPen::CPen**

CPen::FromHandle

static CPen* PASCAL FromHandle(HPEN *hPen* **);**

hPen **HPEN** handle to Windows GDI pen.

Remarks Returns a pointer to a **CPen** object given a handle to a Windows GDI pen object. If a **CPen** object is not attached to the handle, a temporary **CPen** object is created and attached. This temporary **CPen** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. In other words, the temporary object is only valid during the processing of one window message.

Return Value A pointer to a **CPen** object if successful; otherwise **NULL**.

class CPoint : public tagPOINT

The **CPoint** class is similar to the Windows **POINT** structure and also includes member functions to manipulate **CPoint** and **POINT** structures. A **CPoint** object can be used wherever a **POINT** structure is used. The operators of this class that interact with a "size" accept either **CSize** objects or **SIZE** structures, as the two are interchangeable.

#include <afxwin.h>

POINT Structure

The **POINT** data structure looks like this:

```
typedef struct tagPOINT {
   int x;
   int y;
} POINT;
```

The **POINT** structure defines the x- and y-coordinates of a point.

Members

x Specifies the x-coordinate of a point.

y Specifies the y-coordinate of a point.

See Also

CRect, **CSize**

Construction/Destruction—Public Members

CPoint	Constructs a **CPoint**.

Operations—Public Members

Offset	Adds separate values to the **x** and **y** members of the **CPoint**.
operator ==	Checks for equality between two points.
operator !=	Checks for inequality between two points.
operator +=	Offsets a **CPoint** by a size.
operator –=	Subtracts a size from the **CPoint**.

Operators Returning CPoint Values—Public Members

operator +	Returns a **CPoint** offset by a size.
operator –	Returns a **CPoint** offset by a negative size.

Operators Returning CSize Values—Public Members

operator –	Returns the size difference between two points.

Member Functions

CPoint::CPoint

CPoint( );

CPoint(int *initX*, **int** *initY* **);**

CPoint(POINT *initPt* **);**

CPoint(SIZE *initSize* **);**

CPoint(DWORD *dwPoint* **);**

initX Sets the **x** member for the **CPoint**.

initY Sets the **y** member for the **CPoint**.

initPt Windows **POINT** structure or **CPoint** used to initialize **CPoint**.

initSize Sets the **x** and **y** members equal to the corresponding values in **cx** and **cy** values in *initSize*.

dwPoint Sets the low-order word to the **x** member and the high-order word to the **y** member.

Remarks

Constructs a **CPoint** object. If no arguments are given, **x** and **y** members are not initialized.

CPoint::Offset

void Offset(int *xOffset*, **int** *yOffset* **);**

void Offset(POINT *point* **);**

void Offset(SIZE *size* **);**

xOffset Specifies the amount to offset the **x** member of the **CPoint**.

yOffset Specifies the amount to offset the **y** member of the **CPoint**.

point Specifies the amount (**POINT** or **CPoint**) to offset the **CPoint**.

size Specifies the amount (**SIZE** or **CSize**) to offset the **CPoint**.

Remarks Adds separate values to the **x** and **y** members of the **CPoint**.

Return Value A **CPoint** offset by a **POINT**, **CPoint**, **CSize**, or **SIZE**.

Operators

CPoint::operator ==

BOOL operator ==(POINT *point* **) const;**

point Contains a **POINT** structure or **CPoint** object.

Remarks Checks for equality between two points.

Return Value Nonzero if the points are equal; otherwise 0.

CPoint::operator !=

BOOL operator !=(POINT *point* **) const;**

point Contains a **POINT** structure or **CPoint** object.

Remarks Checks for inequality between two points.

Return Value Nonzero if the points are not equal; otherwise 0.

CPoint::operator +=

void operator +=(SIZE *size* **);**

size Contains a **SIZE** structure or **CSize** object.

Remarks Offsets a **CPoint** by a size.

CPoint::operator –=

void operator –=(SIZE *size* **);**

size Contains a **SIZE** structure or **CSize** object.

Remarks Subtracts a size from the **CPoint**.

CPoint::operator +

CPoint operator +(SIZE *size* **) const;**

size Contains a **SIZE** structure or **CSize** object.

Return Value A **CPoint** that is offset by a size.

CPoint::operator –

CSize operator –(POINT *point* **) const;**

CPoint operator –(SIZE *size* **) const;**

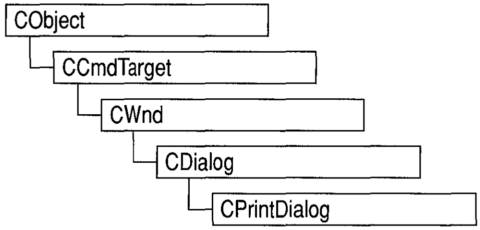
CPoint operator –() const;

point Contains a **POINT** structure or **CPoint** object.

size Contains a **SIZE** structure or **CSize** object.

Return Value A **CSize** that is the difference between two points, or returns a **CPoint** that is offset by a negative size.

class CPrintDialog : public CDialog



The **CPrintDialog** class encapsulates the services provided by the Windows common dialog box for printing. Common print dialog boxes provide an easy way to implement Print and Print Setup dialog boxes in a manner consistent with Windows standards.

If you wish, you can rely on the framework to handle many aspects of the printing process for your application. In this case, the framework automatically displays the Windows common dialog box for printing. You can also have the framework handle printing for your application but override the common Print dialog box with your own Print dialog box. For more information on using the framework to handle printing tasks, see Chapter 9 of the *Class Library User's Guide.*

If you want your application to handle printing without the framework's involvement, you can use the **CPrintDialog** class "as is" with the constructor provided, or you can derive your own dialog class from **CPrintDialog** and write a constructor to suit your needs. In either case, these dialog boxes will behave like standard Microsoft Foundation class dialog boxes because they are derived from class **CDialog**.

To use a **CPrintDialog** object, first create the object using the **CPrintDialog** constructor. Once the dialog box has been constructed, you can set or modify any values in the **m_pd** structure to initialize the values of the dialog box's controls. The **m_pd** structure is of type **PRINTDLG**. For more information on this structure, see the *Windows Software Development Kit* (SDK) documentation.

If you do not supply your own handles in **m_pd** for the **hDevMode** and **hDevNames** members, be sure to call the Windows function **GlobalFree** for these handles when you are done with the dialog box.

After initializing the dialog box controls, call the **DoModal** member function to display the dialog box and allow the user to select the path and file. **DoModal** returns whether the user selected the OK (**IDOK**) or the Cancel (**IDCANCEL**) button.

If **DoModal** returns **IDOK**, you can use one of **CPrintDialog**'s member functions to retrieve the information input by the user.

The **CPrintDialog::GetDefaults** member function is useful for retrieving the current printer defaults without displaying a dialog box. This member function requires no user interaction.

You can use the Windows **CommDlgExtendedError** function to determine if an error occurred during initialization of the dialog box and to learn more about the error. For more information on this function, see the Windows SDK documentation.

CPrintDialog relies on the COMMDLG.DLL file that ships with Windows version 3.1. For details about redistributing COMMDLG.DLL to Windows version 3.0 users, see the *Getting Started* manual for the Windows version 3.1 SDK.

To customize the dialog box, derive a class from **CPrintDialog**, provide a custom dialog template, and add a message map to process the notification messages from the extended controls. Any unprocessed messages should be passed on to the base class. Customizing the hook function is not required.

To process the same message differently depending on whether the dialog box is Print or Print Setup, you must derive a class for each dialog box. You must also override the Windows **AttachOnSetup** function, which handles the creation of a new dialog box when the Print Setup button is selected within a Print dialog box.

#include <afxdlgs.h>

Data Members—Public Members

m_pd	A structure used to customize a **CPrintDialog** object.

Construction/Destruction—Public Members

CPrintDialog	Constructs a **CPrintDialog** object.

Operations—Public Members

DoModal	Displays the dialog box and allows the user to make a selection.
GetCopies	Retrieves the number of copies requested.
GetDefaults	Retrieves device defaults without displaying a dialog box.
GetDeviceName	Retrieves the name of the currently selected printer device.
GetDevMode	Retrieves the **DEVMODE** structure.
GetDriverName	Retrieves the name of the currently selected printer driver.
GetFromPage	Retrieves the starting page of the print range.
GetToPage	Retrieves the ending page of the print range.
GetPortName	Retrieves the name of the currently selected printer port.
GetPrinterDC	Retrieves a handle to the printer device context.

PrintAll	Determines whether to print all pages of the document.
PrintCollate	Determines whether collated copies are requested.
PrintRange	Determines whether to print only a specified range of pages.
PrintSelection	Determines whether to print only the currently selected items.

Member Functions

CPrintDialog::CPrintDialog

CPrintDialog(BOOL *bPrintSetupOnly*, **DWORD** *dwFlags* = **PD_ALLPAGES | PD_USEDEVMODECOPIES | PD_NOPAGENUMS | PD_HIDEPRINTTOFILE | PD_NOSELECTION, CWnd*** *pParentWnd* = **NULL);**

bPrintSetupOnly Specifies whether the standard Windows Print dialog box or Print Setup dialog box is displayed. Set this parameter to **TRUE** to display the standard Windows Print Setup dialog box. Set it to **FALSE** to display the Windows Print dialog box. If *bPrintSetupOnly* is **FALSE**, a Print Setup option button is still displayed in the Print dialog box.

dwFlags One or more flags you can use to customize the settings of the dialog box, combined using the bitwise-OR operator. For example, the **PD_ALLPAGES** flag sets the default print range to all pages of the document. See the **PRINTDLG** structure in the Windows SDK for more information on these flags.

pParentWnd A pointer to the dialog box's parent or owner window.

Remarks

Constructs either a Windows Print or Print Setup dialog object. This member function only constructs the object. Use the **DoModal** member function to invoke the dialog box.

See Also

CPrintDialog::DoModal, **::PrintDlg**, **PRINTDLG**

CPrintDialog::DoModal

virtual int DoModal( );

Remarks

Call this function to display the Windows common print dialog box and allow the user to select various printing options such as the number of copies, page range, and whether copies should be collated.

If you want to initialize the various print dialog options by setting members of the **m_pd** structure, you should do this before calling **DoModal**, but after the dialog object is constructed.

After calling **DoModal**, you can call other member functions to retrieve the settings or information input by the user into the dialog box.

Return Value

IDOK or **IDCANCEL** if the function is successful; otherwise 0. **IDOK** and **IDCANCEL** are constants that indicate whether the user selected the OK or Cancel button.

If **IDCANCEL** is returned, you can call the Windows **CommDlgExtendedError** function to determine if an error occurred.

See Also

CPrintDialog::CPrintDialog, **CDialog::DoModal**

CPrintDialog::GetCopies

int GetCopies( ) const;

Remarks

Call this function after calling **DoModal** to retrieve the number of copies requested.

Return Value

The number of copies requested.

See Also

CPrintDialog::PrintCollate

CPrintDialog::GetDefaults

BOOL GetDefaults( );

Remarks

Call this function to retrieve the device defaults of the default printer without displaying a dialog box. The retrieved values are placed in the **m_pd** structure.

Return Value

Nonzero if the function was successful; otherwise 0.

See Also

CPrintDialog::m_pd

CPrintDialog::GetDeviceName

CString GetDeviceName( ) const;

Remarks

Call this function after calling **DoModal** to retrieve the name of the currently selected printer.

Return Value

The name of the currently selected printer.

See Also

CPrintDialog::GetDriverName, **CPrintDialog::GetDevMode**, **CPrintDialog::GetPortName**

CPrintDialog::GetDevMode

LPDEVMODE GetDevMode( ) const;

Remarks

Call this function after calling **DoModal** to retrieve information about the printing device.

Return Value

The **DEVMODE** data structure, which contains information about the device initialization and environment of a print driver. You must free the memory taken by this structure with the Windows **GlobalFree** function. See **PRINTDLG** in the Windows SDK reference for more information about using **GlobalFree**.

A **DEVMODE** data structure has this form:

```
#include <print.h>
typedef struct tagDEVMODE {   /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    UINT  dmSpecVersion;
    UINT  dmDriverVersion;
    UINT  dmSize;
    UINT  dmDriverExtra;
    DWORD dmFields;
    int   dmOrientation;
    int   dmPaperSize;
    int   dmPaperLength;
    int   dmPaperWidth;
    int   dmScale;
    int   dmCopies;
    int   dmDefaultSource;
    int   dmPrintQuality;
    int   dmColor;
    int   dmDuplex;
    int   dmYResolution;
    int   dmTTOption;
} DEVMODE;
```

For more complete information about this structure, see **DEVMODE** in the Windows SDK documentation.

See Also **CDC::GetDeviceCaps**

CPrintDialog::GetDriverName

CString GetDriverName( ) const;

Remarks Call this function after calling **DoModal** to retrieve the name of the currently selected printer device driver.

Return Value The name of the currently selected printer device driver.

See Also **CPrintDialog::GetDeviceName**, **CPrintDialog::GetDevMode**, **CPrintDialog::GetPortName**

CPrintDialog::GetFromPage

int GetFromPage( ) const;

Remarks

Call this function after calling **DoModal** to retrieve the starting page number in the range of pages to be printed.

Return Value

The starting page number in the range of pages to be printed.

See Also

CPrintDialog::GetToPage, **CPrintDialog::PrintRange**

CPrintDialog::GetPortName

CString GetPortName( ) const;

Remarks

Call this function after calling **DoModal** to retrieve the name of the currently selected printer port.

Return Value

The name of the currently selected printer port.

See Also

CPrintDialog::GetDriverName, **CPrintDialog::GetDeviceName**

CPrintDialog::GetPrinterDC

HDC GetPrinterDC( ) const;

Remarks

If the *bPrintSetupOnly* parameter of the **CPrintDialog** constructor was **FALSE** (indicating that the Print dialog box is displayed), then **GetPrinterDC** returns a handle to the printer device context. You must call the Windows **DeleteDC** function to delete the device context when you are done using it.

Return Value

A handle to the printer device context if successful; otherwise **NULL**.

CPrintDialog::GetToPage

int GetToPage( ) const;

Remarks Call this function after calling **DoModal** to retrieve the ending page number in the range of pages to be printed.

Return Value The ending page number in the range of pages to be printed.

See Also **CPrintDialog::GetFromPage**, **CPrintDialog::PrintRange**

CPrintDialog::PrintAll

BOOL PrintAll( ) const;

Remarks Call this function after calling **DoModal** to determine whether to print all pages in the document.

Return Value Nonzero if all pages in the document are to be printed; otherwise 0.

See Also **CPrintDialog::PrintRange**, **CPrintDialog::PrintSelection**

CPrintDialog::PrintCollate

BOOL PrintCollate( ) const;

Remarks Call this function after calling **DoModal** to determine whether the printer should collate all printed copies of the document.

Return Value Nonzero if the user selects the collate check box in the dialog box; otherwise 0.

See Also **CPrintDialog::GetCopies**

CPrintDialog::PrintRange

BOOL PrintRange( ) const;

Remarks Call this function after calling **DoModal** to determine whether to print only a range of pages in the document.

Return Value Nonzero if only a range of pages in the document are to be printed; otherwise 0.

See Also **CPrintDialog::PrintAll**, **CPrintDialog::PrintSelection**, **CPrintDialog::GetFromPage**, **CPrintDialog::GetToPage**

CPrintDialog::PrintSelection

BOOL PrintSelection( ) const;

Remarks Call this function after calling **DoModal** to determine whether to print only the currently selected items.

Return Value Nonzero if only the selected items are to be printed; otherwise 0.

See Also **CPrintDialog::PrintRange**, **CPrintDialog::PrintAll**

Data Members

CPrintDialog::m_pd

PRINTDLG FAR& m_pd;

Remarks A structure whose members store the characteristics of the dialog object. After constructing a **CPrintDialog** object, you can use **m_pd** to set various aspects of the dialog box before calling the **DoModal** member function. For more information on the **m_pd** structure, see **PRINTDLG** in the Windows SDK documentation.

If you modify the **m_pd** data member directly, you will override any default behavior.

struct CPrintInfo

CPrintInfo stores information about a print or print-preview job. The framework creates an object of **CPrintInfo** each time the Print or Print Preview command is chosen and destroys it when the command is completed.

CPrintInfo contains information about both the print job as a whole, such as the range of pages to be printed, and the current status of the print job, such as the page currently being printed. Some information is stored in an associated **CPrintDialog** object; this object contains the values entered by the user in the Print dialog box.

A **CPrintInfo** object is passed between the framework and your view class during the printing process and is used to exchange information between the two. For example, the framework informs the view class which page of the document to print by assigning a value to the **m_nCurPage** member of **CPrintInfo**; the view class retrieves the value and performs the actual printing of the specified page.

Another example is the case when the length of the document is not known until it is printed. In this situation, the view class tests for the end of the document each time a page is printed. When the end is reached, the view class sets the **m_bContinuePrinting** member of **CPrintInfo** to **FALSE**; this informs the framework to stop the print loop.

CPrintInfo is used by the member functions of **CView** that are listed under "See Also." For more information about the printing architecture provided by the Microsoft Foundation Class Library, see Chapter 4 in this manual and Chapter 9 of the *Class Library User's Guide*.

#include <afxext.h>

See Also **CView::OnBeginPrinting**, **CView::OnEndPrinting**, **CView::OnEndPrintPreview**, **CView::OnPrepareDC**, **CView::OnPreparePrinting**, **CView::OnPrint**

Data Members—Public Members

m_pPD	Contains a pointer to **CPrintDialog** object used for the Print dialog box.
m_bPreview	Contains a flag indicating whether the document is being previewed.
m_bContinuePrinting	Contains a flag indicating whether the framework should continue the print loop.
m_nCurPage	Identifies the number of the page currently being printed.
m_nNumPreviewPages	Identifies the number of pages displayed in the preview window; either 1 or 2.

m_lpUserData	Contains a pointer to a user-created structure.
m_rectDraw	Specifies a rectangle defining the current usable page area.
m_strPageDesc	Contains a format string for page-number display.

Attributes—Public Members

SetMinPage	Sets the number of the first page of the document.
SetMaxPage	Sets the number of the last page of the document.
GetMinPage	Returns the number of the first page of the document.
GetMaxPage	Returns the number of the last page of the document.
GetFromPage	Returns the number of the first page being printed.
GetToPage	Returns the number of the last page being printed.

Member Functions

CPrintInfo::GetFromPage

UINT GetFromPage( );

Remarks Call this function to retrieve the number of the first page to be printed. This is the value specified by the user in the Print dialog box, and it is stored in the **CPrintDialog** object referenced by the **m_pPD** member. If the user has not specified a value, the default is the first page of the document.

See Also **CPrintInfo::m_nCurPage**, **CPrintInfo::m_pPD**, **CPrintInfo::GetToPage**

CPrintInfo::GetMaxPage

UINT GetMaxPage( );

Remarks Call this function to retrieve the number of the last page of the document. This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member.

See Also **CPrintInfo::m_nCurPage**, **CPrintInfo::m_pPD**, **CPrintInfo::GetMinPage**, **CPrintInfo::SetMaxPage**, **CPrintInfo::SetMinPage**

CPrintInfo::GetMinPage

UINT GetMinPage( );

Remarks

Call this function to retrieve the number of the first page of the document. This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member.

See Also

CPrintInfo::m_nCurPage, **CPrintInfo::m_pPD**, **CPrintInfo::GetMaxPage**, **CPrintInfo::SetMaxPage**, **CPrintInfo::SetMinPage**

CPrintInfo::GetToPage

UINT GetToPage( );

Remarks

Call this function to retrieve the number of the last page to be printed. This is the value specified by the user in the Print dialog box, and it is stored in the **CPrintDialog** object referenced by the **m_pPD** member. If the user has not specified a value, the default is the last page of the document.

See Also

CPrintInfo::m_nCurPage, **CPrintInfo::m_pPD**, **CPrintInfo::GetFromPage**

CPrintInfo::SetMaxPage

void SetMaxPage(UINT *nMaxPage* **);**

nMaxPage Number of the last page of the document.

Remarks

Call this function to specify the number of the last page of the document. This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member. If the length of the document is known before it is printed, call this function from your override of **CView::OnPreparePrinting**. If the length of the document depends on a setting specified by the user in the Print dialog box, call this function from your override of **CView::OnBeginPrinting**. If the length of the document is not known until it is printed, use the **m_bContinuePrinting** member to control the print loop.

See Also

CPrintInfo::m_bContinuePrinting, **CPrintInfo::m_nCurPage**, **CPrintInfo::m_pPD**, **CPrintInfo::GetMinPage**, **CPrintInfo::GetToPage**, **CPrintInfo::SetMinPage**, **CView::OnBeginPrinting**, **CView::OnPreparePrinting**

CPrintInfo::SetMinPage

void SetMinPage(UINT *nMinPage* **);**

nMinPage Number of the first page of the document.

Remarks

Call this function to specify the number of the first page of the document. Page numbers normally start at 1. This value is stored in the **CPrintDialog** object referenced by the **m_pPD** member.

See Also

CPrintInfo::m_nCurPage, **CPrintInfo::m_pPD**, **CPrintInfo::GetMaxPage**, **CPrintInfo::GetMinPage**, **CPrintInfo::SetMaxPage**

Data Members

CPrintInfo::m_bContinuePrinting

Remarks

Contains a flag indicating whether the framework should continue the print loop. If you are doing print-time pagination, you can set this member to **FALSE** in your override of **CView::OnPrepareDC** once the end of the document has been reached. You do not have to modify this variable if you have specified the length of the document at the beginning of the print job using the **SetMaxPage** member function. The **m_bContinuePrinting** member is a public variable of type **BOOL**.

See Also

CPrintInfo::SetMaxPage, **CView::OnPrepareDC**

CPrintInfo::m_bPreview

Remarks

Contains a flag indicating whether the document is being previewed. This is set by the framework depending on which command the user executed. The Print dialog box is not displayed for a print-preview job. The **m_bPreview** member is a public variable of type **BOOL**.

See Also

CView::DoPreparePrinting, **CView::OnPreparePrinting**

CPrintInfo::m_lpUserData

Remarks

Contains a pointer to a user-created structure. You can use this to store printing-specific data that you don't want to store in your view class. The **m_lpUserData** member is a public variable of type **LPVOID**.

CPrintInfo::m_nCurPage

Remarks

Contains the number of the current page. The framework calls **CView::OnPrepareDC** and **CView::OnPrint** once for each page of the document, specifying a different value for this member each time; its values range from the value returned by **GetFromPage** to that returned by **GetToPage**. Use this member in your overrides of **CView::OnPrepareDC** and **CView::OnPrint** to print the specified page of the document.

When preview mode is first invoked, the framework reads the value of this member to determine which page of the document should be previewed initially. You can set the value of this member in your override of **CView::OnPreparePrinting** to maintain the user's current position in the document when entering preview mode. The **m_nCurPage** member is a public variable of type **UINT**.

See Also

CPrintInfo::GetFromPage, **CPrintInfo::GetToPage**, **CView::OnPrepareDC**, **CView::OnPreparePrinting**, **CView::OnPrint**

CPrintInfo::m_nNumPreviewPages

Remarks

Contains the number of pages displayed in preview mode; it can be either 1 or 2. The **m_nNumPreviewPages** member is a public variable of type **UINT**.

See Also

CPrintInfo::m_strPageDesc

CPrintInfo::m_pPD

Remarks

Contains a pointer to the **CPrintDialog** object used to display the Print dialog box for the print job. The **m_pPD** member is a public variable of type **CPrintDialog***.

See Also

CPrintDialog

CPrintInfo::m_rectDraw

Remarks Specifies the usable drawing area of the page in logical coordinates. You may want to refer to this in your override of **CView::OnPrint**. You can use this member to keep track of what area remains usable after you print headers, footers, etc. The **m_rectDraw** member is a public variable of type **CRect**.

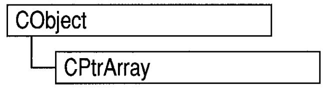
See Also **CView::OnPrint**

CPrintInfo::m_strPageDesc

Remarks Contains a format string used to display the page numbers during print preview; this string consists of two substrings, one for single-page display and one for double-page display, each terminated by a '\n' character. The framework uses "Page %u\nPages %u-%u\n" as the default value. If you want a different format for the page numbers, specify a format string in your override of **CView::OnPreparePrinting**. The **m_strPageDesc** member is a public variable of type **CString**.

See Also **CView::OnPreparePrinting**

class CPtrArray : public CObject



The **CPtrArray** class supports arrays of void pointers. The member functions of **CPtrArray** are similar to the member functions of class **CObArray**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
void* CPtrArray::GetAt( int <nIndex> ) const;
```

CPtrArray incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual pointer array elements, you must set the depth of the dump context to 1 or greater. Pointer arrays may not be serialized. When a pointer array is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

#include <afxcoll.h>

Construction/Destruction — Public Members

CPtrArray	Constructs an empty array for void pointers.
~CPtrArray	Destroys a **CPtrArray** object.

Bounds — Public Members

GetSize	Gets number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations — Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; array is not allowed to grow.
ElementAt	Returns a temporary reference to the element pointer within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

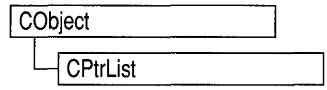
Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

class CPtrList : public CObject



The **CPtrList** class supports lists of void pointers. The member functions of **CPtrList** are similar to the member functions of class **CObList**. Because of this similarity, you can use the **CObList** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a pointer to **void**.

```
CObject*& CObList::GetHead() const;
```

for example, translates to

```
void*& CPtrList::GetHead() const;
```

CPtrList incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual pointer list elements, you must set the depth of the dump context to 1 or greater. Pointer lists may not be serialized. When a **CPtrList** object is deleted, or when its elements are removed, only the pointers are removed, not the entities they reference.

#include <afxcoll.h>

Construction/Destruction—Public Members

CPtrList	Constructs an empty list for void pointers.

Head/Tail Access—Public Members

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations—Public Members

RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.
AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.

Iteration — Public Members

GetHeadPosition	Returns the position of the head element of the list.
GetTailPosition	Returns the position of the tail element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.

Retrieval/Modification — Public Members

GetAt	Gets the element at a given position.
SetAt	Sets the element at a given position.
RemoveAt	Removes an element from this list, specified by position.

Insertion — Public Members

InsertBefore	Inserts a new element before a given position.
InsertAfter	Inserts a new element after a given position.

Searching — Public Members

Find	Gets the position of an element specified by pointer value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status — Public Members

GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

class CRect : public tagRECT

The **CRect** class is similar to a Windows **RECT** structure and also includes member functions to manipulate **CRect** objects and Windows **RECT** structures. A **CRect** object can be passed as a function parameter wherever an **LPRECT** or **RECT** structure can be passed.

A **CRect** contains member variables that define the top-left and bottom-right points of a rectangle. The width or height of the rectangle defined by **CRect** must not exceed 32,767 units.

When specifying a **CRect**, you must be careful to construct it so that the top-left point is above and to the left of the bottom-right point in the Windows coordinate system; otherwise, the **CRect** will not be recognized by some functions, such as **IntersectRect**, **UnionRect**, and **PtInRect**. For example, a top left of (10,10) and bottom right of (20,20) defines a valid rectangle; a top left of (20,20) and bottom right of (10,10), an empty rectangle.

Use caution when manipulating a **CRect** with the **CDC::DPtoLP** and **CDC::LPtoDP** member functions. If the mapping mode of a display context is such that the y-extent is negative, as in **MM_LOENGLISH**, then **CDC::DPtoLP** will transform the **CRect** so that its top is greater than the bottom. Functions such as **Height** and **Size** will then return negative values for the height of the transformed **CRect**.

When using overloaded **CRect** operators, the first operator must be a **CRect**; the second can be either a **RECT** structure or a **CRect** object.

#include <afxwin.h>

RECT Structure

The **RECT** data structure looks like this:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

Members

left Specifies the x-coordinate of the upper-left corner of a rectangle.

top Specifies the y-coordinate of the upper-left corner of a rectangle.

right Specifies the x-coordinate of the lower-right corner of a rectangle.

bottom Specifies the y-coordinate of the lower-right corner of a rectangle.

See Also

CPoint, **CSize**

Construction/Destruction—Public Members

CRect	Constructs a **CRect** object.

Operations—Public Members

Width	Calculates the width of **CRect**.
Height	Calculates the height of **CRect**.
Size	Calculates the size of **CRect**.
TopLeft	Returns a reference to the top-left point of **CRect**.
BottomRight	Returns a reference to the bottom-right point of **CRect**.
IsRectEmpty	Determines whether **CRect** is empty. **CRect** is empty if the width and/or height are 0.
IsRectNull	Determines if the **top**, **bottom**, **left**, and **right** member variables are all equal to 0.
PtInRect	Determines whether the specified point lies within **CRect**.
SetRect	Sets the dimensions of **CRect**.
SetRectEmpty	Sets **CRect** to an empty rectangle (all coordinates equal to 0).
CopyRect	Copies the dimensions of a source rectangle to **CRect**.
EqualRect	Determines whether **CRect** is equal to the given rectangle.
InflateRect	Increases or decreases the width and height of **CRect**.
OffsetRect	Moves **CRect** by the specified offsets.
SubtractRect	Subtracts one rectangle from another.
IntersectRect	Sets **CRect** equal to the intersection of two rectangles.
UnionRect	Sets **CRect** equal to the union of two rectangles.

Operators—Public Members

operator LPCRECT	Converts a **CRect** to an **LPCRECT**.
operator LPRECT	Converts a **CRect** to an **LPRECT**.
operator =	Copies the dimensions of a rectangle to **CRect**.
operator ==	Determines whether **CRect** is equal to a rectangle.
operator !=	Determines whether **CRect** is not equal to a rectangle.
operator +=	Adds the specified offsets to **CRect**.
operator –=	Subtracts the specified offsets from **CRect**.
operator &=	Sets **CRect** equal to the intersection of **CRect** and a rectangle.
operator |=	Sets **CRect** equal to the union of **CRect** and a rectangle.
operator +	Adds the given offsets to **CRect** and returns the resulting **CRect**.
operator –	Subtracts the given offsets from **CRect** and returns the resulting **CRect**.
operator &	Creates the intersection of **CRect** and a rectangle and returns the resulting **CRect**.
operator |	Creates the union of **CRect** and a rectangle and returns the resulting **CRect**.

Member Functions

CRect::BottomRight

CPoint& BottomRight( );

Remarks Returns a reference to the bottom-right point of **CRect**.

Return Value **CPOINT&**, a reference to a **CPoint** object.

CRect::CopyRect

void CopyRect(LPCRECT *lpSrcRect* **);**

lpSrcRect Points to the **RECT** structure or **CRect** object whose dimensions are to be copied.

Remarks Copies the *lpSrcRect* rectangle to the **CRect** object.

See Also **::CopyRect**, **CRect::operator =**

CRect::CRect

CRect( );

CRect(int *l*, **int** *t*, **int** *r*, **int** *b* **);**

CRect(const RECT& *srcRect* **);**

CRect(LPCRECT *lpSrcRect* **);**

CRect(POINT *point*, **SIZE** *size* **);**

l Specifies the left position of the **CRect**.

t Specifies the top of the **CRect**.

r Specifies the right position of the **CRect**.

b Specifies the bottom of the **CRect**.

srcRect Refers to the **RECT** structure with the coordinates for the **CRect** object.

lpSrcRect Points to the **RECT** structure with the coordinates for the **CRect** object.

point Specifies the origin point for the rectangle to be constructed. Corresponds to the top-left corner.

size Specifies the displacement from the top-left corner to the bottom-right corner of the rectangle to be constructed.

Remarks

Constructs a **CRect** object. The **CRect(const RECT&)** and **CRect(LPCRECT)** constructors perform a **CopyRect**. The other constructors initialize the member variables of the object directly.

See Also

CRect::SetRect, **CRect::CopyRect**, **CRect::operator =**

CRect::EqualRect

BOOL EqualRect(LPCRECT *lpRect* **) const;**

lpRect Points to a **RECT** structure or **CRect** object that contains the upper-left and lower-right corner coordinates of a rectangle.

Return Value

Nonzero if the two rectangles have the same top, left, bottom, and right values; otherwise 0.

See Also

::EqualRect

CRect::Height

int Height( ) const;

Remarks

Calculates the height of **CRect** by subtracting the top value from the bottom value. The resulting value may be negative.

Return Value

The height of **CRect**.

CRect::InflateRect

void InflateRect(int *x*, **int** *y* **);**

void InflateRect(SIZE *size* **);**

x Specifies the amount to increase or decrease the width of **CRect**. It must be negative to decrease the width.

y Specifies the amount to increase or decrease the height of **CRect**. It must be negative to decrease the height.

size Contains a **SIZE** or **CSize** that specifies the amounts to add to the **CRect**'s height and width.

Remarks

The parameters of **InflateRect** are signed values; positive values inflate the **CRect** and negative values deflate it. When inflated, the width of **CRect** is increased by two times *x* and its height is increased by two times *y*.

See Also

::InflateRect

CRect::IntersectRect

BOOL IntersectRect(LPCRECT *lpRect1*, **LPCRECT** *lpRect2* **);**

lpRect1 Points to a **RECT** structure or **CRect** object that contains a source rectangle.

lpRect2 Points to a **RECT** structure or **CRect** object that contains a source rectangle.

Remarks

Makes a **CRect** equal to the intersection of two existing rectangles. The intersection is the largest rectangle contained in both existing rectangles.

Note The value of the left coordinate must be less than the right and the top less than the bottom for both *lpRect1* and *lpRect2*.

Return Value

Nonzero if the intersection is not empty; 0 if the intersection is empty.

See Also

::IntersectRect, **CRect::operator &=**, **CRect::operator &**

CRect::IsRectEmpty

BOOL IsRectEmpty( ) const;

Remarks

Determines if **CRect** is empty. A rectangle is empty if the width and/or height are 0 or negative. Differs from **IsRectNull**, which determines if the rectangle is **NULL**.

Return Value Nonzero if **CRect** is empty; 0 if **CRect** is not empty.

See Also **::IsRectEmpty**, **CRect::IsRectNull**

CRect::IsRectNull

BOOL IsRectNull( ) const;

Remarks Determines if the top, left, bottom, and right values of the **CRect** are all equal to 0. Differs from **IsRectEmpty**, which determines if the rectangle is empty.

Return Value Nonzero if the **CRect** object's top, left, bottom, and right values are all equal to 0; otherwise 0.

See Also **CRect::IsRectEmpty**

CRect::OffsetRect

void OffsetRect(int *x*, **int** *y* **);**

void OffsetRect(POINT *point* **);**

void OffsetRect(SIZE *size* **);**

x Specifies the amount to move left or right. It must be negative to move left.

y Specifies the amount to move up or down. It must be negative to move up.

point Contains a **POINT** or **CPoint** specifying both dimensions by which to move.

size Contains a **SIZE** or **CSize** specifying both dimensions by which to move.

Remarks Moves **CRect** by the specified offsets. Moves **CRect** *x* units along the x-axis and *y* units along the y-axis. The *x* and *y* parameters are signed values, so **CRect** can be moved left or right and up or down.

CRect::PtInRect

BOOL PtInRect(POINT *point* **) const;**

point Contains a **POINT** structure or **CPoint** object.

Remarks Determines whether the specified point lies within **CRect**. A point is within **CRect** if it lies on the left or top side or is within all four sides. A point on the right or bottom side is outside **CRect**.

Note The value of the left coordinate of **CRect** must be less than the right and the top less than the bottom.

Return Value Nonzero if the point lies within **CRect**; otherwise 0.

See Also **::PtInRect**

CRect::SetRect

void SetRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

x1 Specifies the x-coordinate of the upper-left corner.

y1 Specifies the y-coordinate of the upper-left corner.

x2 Specifies the x-coordinate of the lower-right corner.

y2 Specifies the y-coordinate of the lower-right corner.

Remarks Sets the dimensions of **CRect** to the specified coordinates.

See Also **CRect::CRect**, **CRect::SetRectEmpty**, **::SetRect**

CRect::SetRectEmpty

void SetRectEmpty( );

Remarks Creates a **NULL** rectangle (all coordinates equal to 0).

See Also **::SetRectEmpty**

CRect::Size

CSize Size( ) const;

Return Value The **CRect** width and height encapsulated as the **cx** and **cy** member variables of a **CSize** object.

CRect::SubtractRect

Windows 3.1 Only **BOOL SubtractRect(LPCRECT** *lpRectSrc1*, **LPCRECT** *lpRectSrc2* **);** ♦

lpRectSrc1 Points to the **RECT** structure from which a rectangle is to be subtracted.

lpRectSrc2 Points to the **RECT** structure that is to be subtracted from the rectangle pointed to by the *lpRectSrc1* parameter.

Remarks Makes the dimensions of a **CRect** object equal to the subtraction of *lpRectSrc2* from *lpRectSrc1*. The rectangle specified by *lpRectSrc2* is subtracted from the rectangle specified by *lpRectSrc1* only when the rectangles intersect completely in either the x- or y-direction. For example, if *lpRectSrc1* were (10,10, 100,100) and *lpRectSrc2* were (50,50, 150,150), the rectangle pointed to by *lpRectSrc1* would contain the same coordinates as the original *lpRectSrc1* when the function returned. If *lpRectSrc1* were (10,10, 100,100) and *lpRectSrc2* were (50,10, 150,150), however, the rectangle pointed to by *lpRectSrc1* would contain the coordinates (10,10, 50,100) when the function returned.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CRect::IntersectRect**, **::UnionRect**, **::SubtractRect**

CRect::TopLeft

CPoint& TopLeft( );

Return Value A reference to the top-left point of **CRect**.

CRect::UnionRect

BOOL UnionRect(LPCRECT *lpRect1*, **LPCRECT** *lpRect2* **);**

lpRect1 Points to a **RECT** or **CRect** that contains a source rectangle.

lpRect2 Points to a **RECT** or **CRect** that contains a source rectangle.

Remarks Makes the dimensions of **CRect** equal to the union of the two source rectangles. The union is the smallest rectangle that contains both source rectangles. The Windows operating system ignores the dimensions of an empty rectangle; that is, a rectangle that has no height or has no width.

Note The value of the left coordinate must be less than the right and the top less than the bottom for both *lpRect1* and *lpRect2*.

Return Value Nonzero if the union is not empty; 0 if the union is empty.

See Also **::UnionRect**, **CRect::operator |=**, **CRect::operator |**

CRect::Width

int Width( ) const;

Remarks Calculates the width of **CRect** by subtracting the left value from the right value. The width may be negative.

Return Value The width of **CRect**.

Operators

CRect::operator LPCRECT

operator LPCRECT( ) const;

Remarks Converts a **CRect** to an **LPCRECT** with no need for the address-of (**&**) operator.

CRect::operator LPRECT

operator LPRECT( );

Remarks

Converts a **CRect** defined as a constant to an **LPRECT** with no need for the address-of (**&**) operator.

CRect::operator =

void operator =(const RECT& *srcRect* **);**

srcRect Refers to a source rectangle. May be a **RECT** or **CRect**.

Remarks

Copies the dimensions of *srcRect* to **CRect**.

See Also

CRect::SetRect, **::CopyRect**

CRect::operator ==

BOOL operator ==(const RECT& *rect* **) const;**

rect Refers to a source rectangle. May be a **RECT** or **CRect**.

Remarks

Determines if *rect* is equal to **CRect** by comparing the coordinates of their upper-left and lower-right corners.

Return Value

If the values of these coordinates are equal, returns nonzero; otherwise 0.

See Also

::EqualRect

CRect::operator !=

BOOL operator !=(const RECT& *rect* **) const;**

rect Refers to a source rectangle. May be a **RECT** or **CRect**.

Remarks

Determines if *rect* is not equal to **CRect** by comparing the coordinates of their upper-left and lower-right corners.

Return Value Nonzero if not equal; otherwise 0.

See Also **CRect::operator ==**

CRect::operator +=

void operator +=(POINT *point* **);**

point Contains a **POINT** or **CPoint**.

Remarks Moves **CRect** by the specified offsets. The *point* parameter's *x* and *y* parameters are added to **CRect**.

See Also **CRect::OffsetRect**

CRect::operator –=

void operator –=(POINT *point* **);**

point Contains a **POINT** or **CPoint**.

Remarks Moves **CRect** by the specified offsets. The *point* parameter's *x* and *y* parameters are subtracted from **CRect**.

See Also **CRect::OffsetRect**

CRect::operator &=

void operator &=(const RECT& *rect* **);**

rect Contains a **RECT** or **CRect**.

Remarks Sets **CRect** equal to the intersection of **CRect** and *rect*. The intersection is the largest rectangle contained in both rectangles.

Note The value of the left coordinate must be less than the right and the top less than the bottom for both **CRect** and *rect*.

See Also **CRect::IntersectRect**

CRect::operator |=

void operator |=(**const RECT&** *rect*);

rect Contains a **CRect** or **RECT**.

Remarks

Sets **CRect** equal to the union of **CRect** and *rect*. The union is the smallest rectangle that contains both source rectangles. Windows ignores the dimensions of an empty rectangle; that is, a rectangle that has no height or has no width.

Note The value of the left coordinate must be less than the right and the top less than the bottom for both **CRect** and *rect*.

See Also

CRect::UnionRect

CRect::operator +

CRect operator +(**POINT** *point*) **const;**

point Contains a **POINT** or **CPoint**.

Remarks

Returns a new **CRect** that is equal to **CRect** displaced by *point*. The *point* parameter's *x* and *y* parameters are added to **CRect**'s position.

Return Value

The **CRect** resulting from the offset by *point*.

See Also

CRect::OffsetRect

CRect::operator –

CRect operator –(**POINT** *point*) **const;**

point Contains a **POINT** or **CPoint**.

Remarks

A new **CRect** that is equal to **CRect** displaced by *–point*. The *point* parameter's *x* and *y* parameters are subtracted from **CRect**'s dimensions.

Return Value

The **CRect** resulting from the offset by *point*.

See Also

CRect::OffsetRect

CRect::operator &

CRect operator &(const RECT& *rect2* **) const;**

rect2 Contains a **RECT** or **CRect**.

Return Value

A **CRect** that is the intersection of **CRect** and *rect2*. The intersection is the largest rectangle contained in both rectangles.

Note The value of the left coordinate must be less than the right and the top less than the bottom for both **CRect** and *rect2*.

See Also

CRect::IntersectRect

CRect::operator |

CRect operator |(const RECT& *rect2* **) const;**

rect2 Contains a **RECT** or **CRect**.

Return Value

A **CRect** that is the union of **CRect** and *rect2*. A union is the smallest rectangle that contains both source rectangles. Windows ignores the dimensions of an empty rectangle; that is, a rectangle that has no height or has no width.

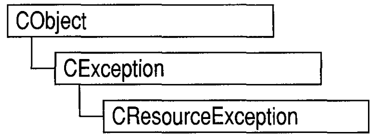
Note The value of the left coordinate must be less than the right and the top less than the bottom for both **CRect** and *rect2*.

See Also

CRect::UnionRect

class CResourceException : public CException

A **CResourceException** object is generated when Windows cannot find or allocate a requested resource. No further qualification is necessary or possible.



#include <afxwin.h>

Construction/Destruction — Public Members

CResourceException	Constructs a **CResourceException** object.

Member Functions

CResourceException::CResourceException

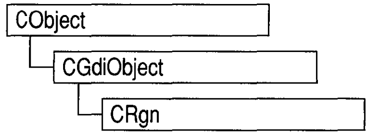
CResourceException( );

Remarks Constructs a **CResourceException** object.

Do not use this constructor directly, but rather call the global function **AfxThrowResourceException**. For more information about exceptions, see Chapter 16, "Exceptions," in the *Class Library User's Guide.*

See Also **AfxThrowResourceException**

class CRgn : public CGdiObject



The **CRgn** class encapsulates a Windows graphics device interface (GDI) region. A region is an elliptical or polygonal area within a window. To use regions, you use the member functions of class **CRgn** with the clipping functions defined as members of class **CDC**. The member functions of **CRgn** create, alter, and retrieve information about the region object for which they are called.

#include <afxwin.h>

Construction/Destruction—Public Members

CRgn	Constructs a **CRgn** object.

Initialization—Public Members

CreateRectRgn	Initializes a **CRgn** object with a rectangular region.
CreateRectRgnIndirect	Initializes a **CRgn** object with a rectangular region defined by a **RECT** structure.
CreateEllipticRgn	Initializes a **CRgn** object with an elliptical region.
CreateEllipticRgnIndirect	Initializes a **CRgn** object with an elliptical region defined by a **RECT** structure.
CreatePolygonRgn	Initializes a **CRgn** object with a polygonal region. The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first.
CreatePolyPolygonRgn	Initializes a **CRgn** object with a region consisting of a series of closed polygons. The polygons may be disjoint or they may overlap.
CreateRoundRectRgn	Initializes a **CRgn** object with a rectangular region with rounded corners.
CombineRgn	Sets a **CRgn** object so that it is equivalent to the union of two specified **CRgn** objects.
CopyRgn	Sets a **CRgn** object so that it is a copy of a specified **CRgn** object.

Operations—Public Members

EqualRgn	Checks two **CRgn** objects to determine whether they are equivalent.
FromHandle	Returns a pointer to a **CRgn** object when given a handle to a Windows region.
GetRgnBox	Retrieves the coordinates of the bounding rectangle of a **CRgn** object.
OffsetRgn	Moves a **CRgn** object by the specified offsets.
PtInRegion	Determines whether a specified point is in the region.
RectInRegion	Determines whether any part of a specified rectangle is within the boundaries of the region.
SetRectRgn	Sets the **CRgn** object to the specified rectangular region.

Member Functions

CRgn::CombineRgn

int CombineRgn(CRgn* *pRgn1*, **CRgn*** *pRgn2*, **int** *nCombineMode* **);**

pRgn1 Identifies an existing region.

pRgn2 Identifies an existing region.

nCombineMode Specifies the operation to be performed when combining the two source regions. It can be any one of the following values:

- **RGN_AND** Uses overlapping areas of both regions (intersection).
- **RGN_COPY** Creates a copy of region 1 (identified by *pRgn1*).
- **RGN_DIFF** Creates a region consisting of the areas of region 1 (identified by *pRgn1*) that are not part of region 2 (identified by *pRgn2*).
- **RGN_OR** Combines both regions in their entirety (union).
- **RGN_XOR** Combines both regions but removes overlapping areas.

Remarks

Creates a new GDI region by combining two existing regions. The regions are combined as specified by *nCombineMode*. The two specified regions are combined, and the resulting region handle is stored in the **CRgn** object. Thus, whatever region is stored in the **CRgn** object is replaced by the combined region. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller. Use **CopyRgn** to simply copy one region into another region.

Return Value

Specifies the type of the resulting region. It can be one of the following values:

- **COMPLEXREGION** New region has overlapping borders.
- **ERROR** No new region created.
- **NULLREGION** New region is empty.
- **SIMPLEREGION** New region has no overlapping borders.

See Also

CRgn::CopyRgn, **::CombineRgn**

CRgn::CopyRgn

int CopyRgn(CRgn* *pRgnSrc* **);**

pRgnSrc Identifies an existing region.

Remarks

Copies the region defined by *pRgnSrc* into the **CRgn** object. The new region replaces the region formerly stored in the **CRgn** object. This function is a special case of the **CombineRgn** member function.

Return Value

Specifies the type of the resulting region. It can be one of the following values:

- **COMPLEXREGION** New region has overlapping borders.
- **ERROR** No new region created.
- **NULLREGION** New region is empty.
- **SIMPLEREGION** New region has no overlapping borders.

See Also

CRgn::CombineRgn, **::CombineRgn**

CRgn::CreateEllipticRgn

BOOL CreateEllipticRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

x1 Specifies the logical x-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

y1 Specifies the logical y-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

x2 Specifies the logical x-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

y2 Specifies the logical y-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

Remarks

Creates an elliptical region. The region is defined by the bounding rectangle specified by *x1*, *y1*, *x2*, and *y2*. The region is stored in the **CRgn** object. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller. When it has finished using a region created with the **CreateEllipticRgn** function, an application should select the region out of the device context and use the **DeleteObject** function to remove it.

Return Value

Nonzero if the operation succeeded; otherwise 0.

See Also

CRgn::CreateEllipticRgnIndirect, **::CreateEllipticRgn**

CRgn::CreateEllipticRgnIndirect

BOOL CreateEllipticRgnIndirect(LPCRECT *lpRect* **);**

lpRect Points to a RECT structure or a CRect object that contains the logical coordinates of the upper-left and lower-right corners of the bounding rectangle of the ellipse.

Remarks

Creates an elliptical region. The region is defined by the structure or object pointed to by *lpRect* and is stored in the **CRgn** object. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller. When it has finished using a region created with the **CreateEllipticRgnIndirect** function, an application should select the region out of the device context and use the **DeleteObject** function to remove it.

Return Value

Nonzero if the operation succeeded; otherwise 0.

See Also

CRgn::CreateEllipticRgn, **::CreateEllipticRgnIndirect**

CRgn::CreatePolygonRgn

BOOL CreatePolygonRgn(LPPOINT *lpPoints*, **int** *nCount*, **int** *nMode* **);**

lpPoints Points to an array of POINT structures or an array of CPoint objects. Each structure specifies the x-coordinate and y-coordinate of one vertex of the polygon. The POINT structure has the following form:

```
typedef struct tagPOINT {
   int x;
   int y;
} POINT;
```

nCount Specifies the number of **POINT** structures or **CPoint** objects in the array pointed to by *lpPoints*.

nMode Specifies the filling mode for the region. This parameter may be either **ALTERNATE** or **WINDING**.

Remarks

Creates a polygonal region. The system closes the polygon automatically, if necessary, by drawing a line from the last vertex to the first. The resulting region is stored in the **CRgn** object. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When the polygon-filling mode is **ALTERNATE**, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on. When the polygon-filling mode is **WINDING**, the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through a counterclockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

When an application has finished using a region created with the **CreatePolygonRgn** function, it should select the region out of the device context and use the **DeleteObject** function to remove it.

Return Value

Nonzero if the operation succeeded; otherwise 0.

See Also

CRgn::CreatePolyPolygonRgn, **::CreatePolygonRgn**

CRgn::CreatePolyPolygonRgn

BOOL CreatePolyPolygonRgn(LPPOINT *lpPoints*, **LPINT** *lpPolyCounts*, **int** *nCount*, **int** *nPolyFillMode* **);**

lpPoints Points to an array of POINT structures or an array of CPoint objects that defines the vertices of the polygons. Each polygon must be explicitly closed because the system does not close them automatically. The polygons are specified consecutively. The POINT structure has the following form:

```
typedef struct tagPOINT {
   int x;
   int y;
} POINT;
```

lpPolyCounts Points to an array of integers. The first integer specifies the number of vertices in the first polygon in the *lpPoints* array, the second integer specifies the number of vertices in the second polygon, and so on.

nCount Specifies the total number of integers in the *lpPolyCounts* array.

nPolyFillMode Specifies the polygon-filling mode. This value may be either **ALTERNATE** or **WINDING**.

Remarks

Creates a region consisting of a series of closed polygons. The resulting region is stored in the **CRgn** object. The polygons may be disjoint or they may overlap. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller.

When the polygon-filling mode is **ALTERNATE**, the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on. When the polygon-filling mode is **WINDING**, the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or a counterclockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through a counterclockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

When an application has finished using a region created with the **CreatePolyPolygonRgn** function, it should select the region out of the device context and use the **DeleteObject** function to remove it.

Return Value

Nonzero if the operation succeeded; otherwise 0.

See Also

CRgn::CreatePolygonRgn, **CDC::SetPolyFillMode**, **::CreatePolyPolygonRgn**

CRgn::CreateRectRgn

BOOL CreateRectRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

x1 Specifies the logical x-coordinate of the upper-left corner of the region.

y1 Specifies the logical y-coordinate of the upper-left corner of the region.

x2 Specifies the logical x-coordinate of the lower-right corner of the region.

y2 Specifies the logical y-coordinate of the lower-right corner of the region.

Remarks Creates a rectangular region that is stored in the **CRgn** object. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller. When it has finished using a region created by **CreateRectRgn**, an application should use the **DeleteObject** function to remove the region.

Return Value Nonzero if the operation succeeded; otherwise 0.

See Also **CRgn::CreateRectRgnIndirect**, **CRgn::CreateRoundRectRgn**, **::CreateRectRgn**

CRgn::CreateRectRgnIndirect

BOOL CreateRectRgnIndirect(LPCRECT *lpRect* **);**

lpRect Points to a RECT structure or CRect object that contains the logical coordinates of the upper-left and lower-right corners of the region. The RECT structure has the following form:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Remarks Creates a rectangular region that is stored in the **CRgn** object. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller. When it has finished using a region created by **CreateRectRgnIndirect**, an application should use the **DeleteObject** function to remove the region.

Return Value Nonzero if the operation succeeded; otherwise 0.

See Also **CRgn::CreateRectRgn**, **CRgn::CreateRoundRectRgn**, **::CreateRectRgnIndirect**

CRgn::CreateRoundRectRgn

BOOL CreateRoundRectRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3* **);**

x1 Specifies the logical x-coordinate of the upper-left corner of the region.

y1 Specifies the logical y-coordinate of the upper-left corner of the region.

x2 Specifies the logical x-coordinate of the lower-right corner of the region.

y2 Specifies the logical y-coordinate of the lower-right corner of the region.

x3 Specifies the width of the ellipse used to create the rounded corners.

y3 Specifies the height of the ellipse used to create the rounded corners.

Remarks

Creates a rectangular region with rounded corners that is stored in the **CRgn** object. The size of a region is limited to 32,767 by 32,767 logical units or 64K of memory, whichever is smaller. When an application has finished using a region created with the **CreateRoundRectRgn** function, it should select the region out of the device context and use the **DeleteObject** function to remove it.

Return Value

Nonzero if the operation succeeded; otherwise 0.

See Also

CRgn::CreateRectRgn, **CRgn::CreateRectRgnIndirect**, **::CreateRoundRectRgn**

CRgn::CRgn

CRgn( );

Remarks

Constructs a **CRgn** object. The **m_hObject** data member does not contain a valid Windows GDI region until the object is initialized with one or more of the other **CRgn** member functions.

CRgn::EqualRgn

BOOL EqualRgn(CRgn* *pRgn* **) const;**

pRgn Identifies a region.

Remarks Determines whether the given region is equivalent to the region stored in the **CRgn** object.

Return Value Nonzero if the two regions are equivalent; otherwise 0.

See Also **::EqualRgn**

CRgn::FromHandle

static CRgn* PASCAL FromHandle(HRGN *hRgn* **);**

hRgn Specifies a handle to a Windows region.

Remarks Returns a pointer to a **CRgn** object when given a handle to a Windows region. If a **CRgn** object is not already attached to the handle, a temporary **CRgn** object is created and attached. This temporary **CRgn** object is valid only until the next time the application has idle time in its event loop, at which time all temporary graphic objects are deleted. Another way of saying this is that the temporary object is only valid during the processing of one window message.

Return Value A pointer to a **CRgn** object. If the function was not successful, the return value is **NULL.**

CRgn::GetRgnBox

int GetRgnBox(LPRECT *lpRect* **) const;**

lpRect Points to a RECT structure or CRect object to receive the coordinates of the bounding rectangle. The RECT structure has the following form:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Remarks Retrieves the coordinates of the bounding rectangle of the **CRgn** object.

Return Value Specifies the region's type. It can be any of the following values:

- **COMPLEXREGION** Region has overlapping borders.
- **NULLREGION** Region is empty.

- **ERROR** **CRgn** object does not specify a valid region.
- **SIMPLEREGION** Region has no overlapping borders.

See Also **::GetRgnBox**

CRgn::OffsetRgn

int OffsetRgn(int *x*, **int** *y* **);**

int OffsetRgn(POINT *point* **);**

x Specifies the number of units to move left or right.

y Specifies the number of units to move up or down.

point The x-coordinate of *point* specifies the number of units to move left or right. The y-coordinate of *point* specifies the number of units to move up or down. The *point* parameter may be either a **POINT** structure or a **CPoint** object.

Remarks Moves the region stored in the **CRgn** object by the specified offsets. The function moves the region *x* units along the x-axis and *y* units along the y-axis. The coordinate values of a region must be less than or equal to 32,767 and greater than or equal to –32,768. The *x* and *y* parameters must be carefully chosen to prevent invalid region coordinates.

Return Value The new region's type. It can be any one of the following values:

- **COMPLEXREGION** Region has overlapping borders.
- **ERROR** Region handle is not valid.
- **NULLREGION** Region is empty.
- **SIMPLEREGION** Region has no overlapping borders.

See Also **::OffsetRgn**

CRgn::PtInRegion

BOOL PtInRegion(int *x*, **int** *y* **) const;**

BOOL PtInRegion(POINT *point* **) const;**

x Specifies the logical x-coordinate of the point to test.

y Specifies the logical y-coordinate of the point to test.

point The x- and y-coordinates of *point* specify the x- and y-coordinates of the point to test the value of. The *point* parameter can either be a **POINT** structure or a **CPoint** object.

Remarks Checks whether the point given by *x* and *y* is in the region stored in the **CRgn** object.

Return Value Nonzero if the point is in the region; otherwise 0.

See Also **::PtInRegion**

CRgn::RectInRegion

BOOL RectInRegion(LPCRECT *lpRect* **) const;**

lpRect Points to a RECT structure or CRect object. The RECT structure has the following form:

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Remarks Determines whether any part of the rectangle specified by *lpRect* is within the boundaries of the region stored in the **CRgn** object.

Return Value Nonzero if any part of the specified rectangle lies within the boundaries of the region; otherwise 0.

See Also **::RectInRegion**

CRgn::SetRectRgn

void SetRectRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

void SetRectRgn(LPCRECT *lpRect* **);**

x1 Specifies the x-coordinate of the upper-left corner of the rectangular region.

y1 Specifies the y-coordinate of the upper-left corner of the rectangular region.

x2 Specifies the x-coordinate of the lower-right corner of the rectangular region.

y2 Specifies the y-coordinate of the lower-right corner of the rectangular region.

lpRect Specifies the rectangular region. Can be either a pointer to a **RECT** structure or a **CRect** object.

Remarks

Creates a rectangular region. Unlike **CreateRectRgn**, however, it does not allocate any additional memory from the local Windows application heap. Instead, it uses the space allocated for the region stored in the **CRgn** object. This means that the **CRgn** object must already have been initialized with a valid Windows region before calling **SetRectRgn**. The points given by *x1*, *y1*, *x2*, and *y2* specify the minimum size of the allocated space. Use this function instead of the **CreateRectRgn** member function to avoid calls to the local memory manager.

See Also

CRgn::CreateRectRgn, **::SetRectRgn**

struct CRuntimeClass

Each class derived from **CObject** is associated with a **CRuntimeClass** structure that you can use to obtain information about an object or its base class at run time. The ability to determine the class of an object at run time is useful when extra type checking of function arguments is needed, or when you must write special-purpose code based on the class of an object. Run-time class information is not supported directly by the C++ language.

The structure has the following members:

LPCSTR m_lpszClassName
A null-terminated string containing the ASCII class name.

int m_nObjectSize
The size of the object, in bytes. If the object has data members that point to allocated memory, the size of that memory is not included.

WORD m_wSchema
The schema number (–1 for nonserializable classes). See the **IMPLEMENT_SERIAL** macro for a description of the schema number.

void (*m_pfnConstruct)(void* p)
A pointer to the default constructor of your class (valid only if the class supports dynamic creation).

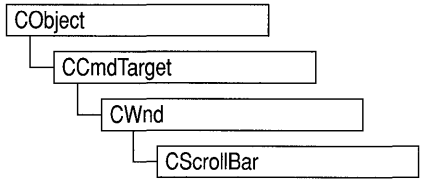
CRuntimeClass* m_pBaseClass
A pointer to the **CRuntimeClass** structure that corresponds to the base class.

CObject* CreateObject( );
Classes derived from **CObject** can support dynamic creation, which is the ability to create an object of a specified class at run time. Document, view, and frame classes, for example, should support dynamic creation. The **CreateObject** member function can be used to implement this function and create objects for these classes during run time. For more information on dynamic creation and the **CreateObject** member, see Chapter 12 of the *Class Library User's Guide*.

Note To use the **CRuntimeClass** structure, you must include the **IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE** or **IMPLEMENT_SERIAL** macro in the implementation of the class for which you want to retrieve run-time object information.

See Also **CObject::GetRuntimeClass, CObject::IsKindOf, RUNTIME_CLASS, IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE, IMPLEMENT_SERIAL**

class CScrollBar : public CWnd



The **CScrollBar** class provides the functionality of a Windows scroll-bar control. You create a scroll-bar control in two steps. First, call the constructor **CScrollBar** to construct the **CScrollBar** object, then call the **Create** member function to create the Windows scroll-bar control and attach it to the **CScrollBar** object.

If you create a **CScrollBar** object within a dialog box (through a dialog resource), the **CScrollBar** is automatically destroyed when the user closes the dialog box. If you create a **CScrollBar** object within a window, you may also need to destroy it.

If you create the **CScrollBar** object on the stack, it is destroyed automatically. If you create the **CScrollBar** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user terminates the Windows scroll bar. If you allocate any memory in the **CScrollBar** object, override the **CScrollBar** destructor to dispose of the allocations.

#include <afxwin.h>

See Also **CWnd**, **CButton**, **CComboBox**, **CEdit**, **CListBox**, **CStatic**, **CDialog**

Construction/Destruction—Public Members

CScrollBar Constructs a **CScrollBar** object.

Initialization—Public Members

Create Creates the Windows scroll bar and attaches it to the **CScrollBar** object.

Operations—Public Members

GetScrollPos	Retrieves the current position of a scroll box.
SetScrollPos	Sets the current position of a scroll box.
GetScrollRange	Retrieves the current minimum and maximum scroll-bar positions for the given scroll bar.
SetScrollRange	Sets minimum and maximum position values for the given scroll bar.
ShowScrollBar	Shows or hides a scroll bar.
EnableScrollBar	Enables or disables one or both arrows of a scroll bar.

Member Functions

CScrollBar::Create

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

dwStyle Specifies the scroll bar's style.

rect Specifies the scroll bar's size and position. Can be either a **RECT** structure or a **CRect** object.

pParentWnd Specifies the scroll bar's parent window, usually a **CDialog** object. It must not be **NULL**.

nID The scroll bar's control ID.

Remarks

You construct a **CScrollBar** object in two steps. First call the constructor, which constructs the **CScrollBar** object; then call **Create**, which creates and initializes the associated Windows scroll bar and attaches it to the **CScrollBar** object.

Apply the following window styles to a scroll bar:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely
- **WS_GROUP** To group controls

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

Return Value

Nonzero if successful; otherwise 0.

Scroll-Bar Styles

You can use any combination of the following scroll-bar styles for *dwStyle*:

- **SBS_BOTTOMALIGN** Used with the **SBS_HORZ** style. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified in the **Create** member function. The scroll bar has the default height for system scroll bars.
- **SBS_HORZ** Designates a horizontal scroll bar. If neither the **SBS_BOTTOMALIGN** nor **SBS_TOPALIGN** style is specified, the scroll bar has the height, width, and position given in the **Create** member function.
- **SBS_LEFTALIGN** Used with the **SBS_VERT** style. The left edge of the scroll bar is aligned with the left edge of the rectangle specified in the **Create** member function. The scroll bar has the default width for system scroll bars.
- **SBS_RIGHTALIGN** Used with the **SBS_VERT** style. The right edge of the scroll bar is aligned with the right edge of the rectangle specified in the **Create** member function. The scroll bar has the default width for system scroll bars.
- **SBS_SIZEBOX** Designates a size box. If neither the **SBS_SIZEBOXBOTTOMRIGHTALIGN** nor **SBS_SIZEBOXTOPLEFTALIGN** style is specified, the size box has the height, width, and position given in the **Create** member function.
- **SBS_SIZEBOXBOTTOMRIGHTALIGN** Used with the **SBS_SIZEBOX** style. The lower-right corner of the size box is aligned with the lower-right corner of the rectangle specified in the **Create** member function. The size box has the default size for system size boxes.
- **SBS_SIZEBOXTOPLEFTALIGN** Used with the **SBS_SIZEBOX** style. The upper-left corner of the size box is aligned with the upper-left corner of the rectangle specified in the **Create** member function. The size box has the default size for system size boxes.
- **SBS_TOPALIGN** Used with the **SBS_HORZ** style. The top edge of the scroll bar is aligned with the top edge of the rectangle specified in the **Create** member function. The scroll bar has the default height for system scroll bars.
- **SBS_VERT** Designates a vertical scroll bar. If neither the **SBS_RIGHTALIGN** nor **SBS_LEFTALIGN** style is specified, the scroll bar has the height, width, and position given in the **Create** member function.

See Also

CScrollBar::CScrollBar

CScrollBar::CScrollBar

CScrollBar( );

Remarks Constructs a **CScrollBar** object. After constructing the object, call the **Create** member function to create and initialize the Windows scroll bar.

See Also **CScrollBar::Create**

CScrollBar::EnableScrollBar

Windows 3.1 Only **BOOL EnableScrollBar(UINT** *nArrowFlags* **= ESB_ENABLE_BOTH);** ♦

nArrowFlags Specifies whether the scroll arrows are enabled or disabled and which arrows are enabled or disabled. This parameter can be one of the following values:

- **ESB_ENABLE_BOTH** Enables both arrows of a scroll bar.
- **ESB_DISABLE_LTUP** Disables the left arrow of a horizontal scroll bar or the up arrow of a vertical scroll bar.
- **ESB_DISABLE_RTDN** Disables the right arrow of a horizontal scroll bar or the down arrow of a vertical scroll bar.
- **ESB_DISABLE_BOTH** Disables both arrows of a scroll bar.

Remarks Enables or disables one or both arrows of a scroll bar.

Return Value Nonzero if the arrows are enabled or disabled as specified; otherwise 0, which indicates that the arrows are already in the requested state or that an error occurred.

See Also **CWnd::EnableScrollBar**, **::EnableScrollBar**

CScrollBar::GetScrollPos

int GetScrollPos( ) const;

Remarks Retrieves the current position of a scroll box. The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 100 to 200 and the scroll box is in the middle of the bar, the current position is 150.

Return Value Specifies the current position of the scroll box if successful; otherwise 0.

See Also **CScrollBar::SetScrollPos**, **CScrollBar::GetScrollRange**, **CScrollBar::SetScrollRange**, **::GetScrollPos**

CScrollBar::GetScrollRange

void GetScrollRange(LPINT *lpMinPos*, **LPINT** *lpMaxPos* **) const;**

lpMinPos Points to the integer variable that is to receive the minimum position.

lpMaxPos Points to the integer variable that is to receive the maximum position.

Remarks Copies the current minimum and maximum scroll-bar positions for the given scroll bar to the locations specified by *lpMinPos* and *lpMaxPos*. The default range for a scroll-bar control is empty (both values are 0).

See Also **::GetScrollRange**, **CScrollBar::SetScrollRange**, **CScrollBar::GetScrollPos**, **CScrollBar::SetScrollPos**

CScrollBar::SetScrollPos

int SetScrollPos(int *nPos*, **BOOL** *bRedraw* = **TRUE);**

nPos Specifies the new position for the scroll box. It must be within the scrolling range.

bRedraw Specifies whether the scroll bar should be redrawn to reflect the new position. If *bRedraw* is **TRUE**, the scroll bar is redrawn. If it is **FALSE**, it is not redrawn. The scroll bar is redrawn by default.

Remarks Sets the current position of a scroll box to that specified by *nPos* and, if specified, redraws the scroll bar to reflect the new position. Set *bRedraw* to **FALSE** whenever the scroll bar will be redrawn by a subsequent call to another function to avoid having the scroll bar redrawn twice within a short interval.

Return Value Specifies the previous position of the scroll box if successful; otherwise 0.

See Also **CScrollBar::GetScrollPos**, **CScrollBar::GetScrollRange**, **CScrollBar::SetScrollRange**, **::SetScrollPos**

CScrollBar::SetScrollRange

void SetScrollRange(int *nMinPos*, **int** *nMaxPos*, **BOOL** *bRedraw* = **TRUE);**

nMinPos Specifies the minimum scrolling position.

nMaxPos Specifies the maximum scrolling position.

bRedraw Specifies whether the scroll bar should be redrawn to reflect the change. If *bRedraw* is **TRUE**, the scroll bar is redrawn; if **FALSE**, it is not redrawn. It is redrawn by default.

Remarks

Sets minimum and maximum position values for the given scroll bar. Set *nMinPos* and *nMaxPos* to 0 to hide standard scroll bars. Do not call this function to hide a scroll bar while processing a scroll-bar notification message. If a call to **SetScrollRange** immediately follows a call to the **SetScrollPos** member function, set *bRedraw* in **SetScrollPos** to 0 to prevent the scroll bar from being redrawn twice.

The difference between the values specified by *nMinPos* and *nMaxPos* must not be greater than 32,767. The default range for a scroll-bar control is empty (both *nMinPos* and *nMaxPos* are 0).

See Also

CScrollBar::GetScrollPos, **CScrollBar::SetScrollPos**, **CScrollBar::GetScrollRange**, **::SetScrollRange**

CScrollBar::ShowScrollBar

Windows 3.1 Only

void ShowScrollBar(BOOL *bShow* = **TRUE);** ♦

bShow Specifies whether the scroll bar is shown or hidden. If this parameter is **TRUE**, the scroll bar is shown; otherwise it is hidden.

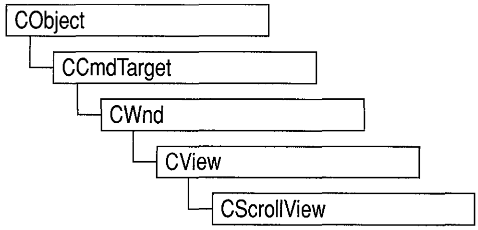
Remarks

Shows or hides a scroll bar. An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

See Also

CScrollBar::GetScrollPos, **CScrollBar::GetScrollRange**, **CWnd::ScrollWindow**, **CScrollBar::SetScrollPos**, **CScrollBar::SetScrollRange**

class CScrollView : public CView



The **CScrollView** class is a **CView** with scrolling capabilities.

You can handle scrolling yourself in any class derived from **CView** by overriding the message-mapped **OnHScroll** and **OnVScroll** member functions. But **CScrollView** adds the following features to its **CView** capabilities:

- It manages window and viewport sizes and mapping modes.
- It scrolls automatically in response to scroll-bar messages.

To take advantage of automatic scrolling, derive your view class from **CScrollView** instead of from **CView**. When the view is first created, if you want to calculate the size of the scrollable view based on the size of the document, call the **SetScrollSizes** member function from your override of either **CView::OnInitialUpdate** or **CView::OnUpdate**. (You must write your own code to query the size of the document. For an example, see Chapter 8 in the *Class Library User's Guide*.)

The call to the **SetScrollSizes** member function sets the view's mapping mode, the total dimensions of the scroll view, and the amounts to scroll horizontally and vertically. All sizes are in logical units. The logical size of the view is usually calculated from data stored in the document, but in some cases you may want to specify a fixed size. For examples of both approaches, see **CScrollView::SetScrollSizes**.

You specify the amounts to scroll horizontally and vertically in logical units. By default, if the user clicks a scroll bar shaft outside of the scroll box, **CScrollView** scrolls a "page." If the user clicks a scroll arrow at either end of a scroll bar, **CScrollView** scrolls a "line." By default, a page is 1/10 of the total size of the view; a line is 1/10 of the page size. Override these default values by passing custom sizes in the **SetScrollSizes** member function. For example, you might set the horizontal size to some fraction of the width of the total size and the vertical size to the height of a line in the current font.

Instead of scrolling, **CScrollView** can automatically scale the view to the current window size. In this mode, the view has no scroll bars and the logical view is stretched or shrunk to exactly fit the window's client area. To use this scale-to-fit capability, call **CScrollView::SetScaleToFitSize**. (Call either **SetScaleToFitSize** or **SetScrollSizes**, but not both.)

Before the `OnDraw` member function of your derived view class is called, **CScrollView** automatically adjusts the viewport origin for the **CPaintDC** device-context object that it passes to `OnDraw`.

To adjust the viewport origin for the scrolling window, **CScrollView** overrides **CView::OnPrepareDC**. This adjustment is automatic for the **CPaintDC** device context that **CScrollView** passes to `OnDraw`, but you must call **CScrollView::OnPrepareDC** yourself for any other device contexts you use, such as a **CClientDC**. You can override **CScrollView::OnPrepareDC** to set the pen, background color, and other drawing attributes, but call the base class to do scaling.

Scroll bars may appear in three places relative to a view, as shown in the following cases:

- Standard window-style scroll bars can be set for the view using the **WS_HSCROLL** and **WS_VSCROLL** styles.
- Scroll-bar controls can also be added to the frame containing the view, in which case the framework forwards **WM_HSCROLL** and **WM_VSCROLL** messages from the frame window to the currently active view.
- The framework also forwards scroll messages from a **CSplitterWnd** splitter control to the currently active splitter pane (a view). When placed in a **CSplitterWnd** with shared scroll bars, a **CScrollView** object will use the shared ones rather than creating its own.

#include <afxwin.h>

See Also **CView**, **CSplitterWnd**

Operations—Public Members

FillOutsideRect	Fills the area of a view outside the scrolling area.
GetDeviceScrollPosition	Gets the current scroll position in device units.
GetDeviceScrollSizes	Gets the current mapping mode, the total size, and the line and page sizes of the scrollable view. Sizes are in device units.
GetScrollPosition	Gets the current scroll position in logical units.
GetTotalSize	Gets the total size of the scroll view in logical units.
ResizeParentToFit	Causes the size of the view to dictate the size of its frame.
ScrollToPosition	Scrolls the view to a given point, specified in logical units.

SetScaleToFitSize	Puts the scroll view into scale-to-fit mode.
SetScrollSizes	Sets the scroll view's mapping mode, total size, and horizontal and vertical scroll amounts.

Construction/Destruction—Protected Members

CScrollView	Constructs a **CScrollView** object.

Member Functions

CScrollView::CScrollView

Protected **CScrollView( );** ♦

Remarks Constructs a **CScrollView** object. You must call either **SetScrollSizes** or **SetScaleToFitSize** before the scroll view is usable.

See Also **CScrollView::SetScrollSizes**, **CScrollView::SetScaleToFitSize**

CScrollView::FillOutsideRect

void FillOutsideRect(CDC* *pDC*, **CBrush*** *pBrush* **);**

pDC Device context in which the filling is to be done.

pBrush Brush with which the area is to be filled.

Remarks Call **FillOutsideRect** to fill the area of the view that appears outside of the scrolling area. Use **FillOutsideRect** in your scroll view's **OnEraseBkgnd** handler function to prevent excessive background repainting.

See Also **CWnd::OnEraseBkgnd**

Example

```
BOOL CScaleView::OnEraseBkgnd( CDC* pDC )
{
    CBrush br( GetSysColor( COLOR_WINDOW ) );
    FillOutsideRect( pDC, &br );
    return TRUE;                    // Erased
}
```

CScrollView::GetDeviceScrollPosition

CPoint GetDeviceScrollPosition( ) const;

Remarks

Call **GetDeviceScrollPosition** when you need the current horizontal and vertical positions of the scroll boxes in the scroll bars. This coordinate pair corresponds to the location in the document to which the upper-left corner of the view has been scrolled. This is useful for offsetting mouse-device positions to scroll-view device positions.

GetDeviceScrollPosition returns values in device units. If you want logical units, use **GetScrollPosition** instead.

See Also

CScrollView::GetScrollPosition

CScrollView::GetDeviceScrollSizes

void GetDeviceScrollSizes(int& *nMapMode*, **SIZE&** *sizeTotal*, **SIZE&** *sizePage*, **SIZE&** *sizeLine* **) const;**

nMapMode Returns the current mapping mode for this view. For a list of possible values, see **SetScrollSizes**.

sizeTotal Returns the current total size of the scroll view in device units.

sizePage Returns the current horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll-bar shaft. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

sizeLine Returns the current horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll arrow. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

Remarks

GetDeviceScrollSizes gets the current mapping mode, the total size, and the line and page sizes of the scrollable view. Sizes are in device units. This member function is rarely called.

See Also

CScrollView::SetScrollSizes, **CScrollView::GetTotalSize**

CScrollView::GetScrollPosition

CPoint GetScrollPosition( ) const;

Remarks Call **GetScrollPosition** when you need the current horizontal and vertical positions of the scroll boxes in the scroll bars. This coordinate pair corresponds to the location in the document to which the upper-left corner of the view has been scrolled.

GetScrollPosition returns values in logical units. If you want device units, use **GetDeviceScrollPosition** instead.

See Also **CScrollView::GetDeviceScrollPosition**

CScrollView::GetTotalSize

CSize GetTotalSize( ) const;

Remarks Call **GetTotalSize** to retrieve the current horizontal and vertical sizes of the scroll view.

Return Value The total size of the scroll view in logical units. The horizontal size is in the **cx** member of the **CSize** return value. The vertical size is in the **cy** member.

See Also **CScrollView::GetDeviceScrollSizes**, **CScrollView::SetScrollSizes**

CScrollView::ResizeParentToFit

void ResizeParentToFit(BOOL *bShrinkOnly* **= TRUE);**

bShrinkOnly The kind of resizing to perform. The default value, **TRUE**, shrinks the frame window if appropriate. Scroll bars will still appear for large views or small frame windows. A value of **FALSE** causes the view always to resize the frame window exactly. This can be somewhat dangerous since the frame window could get too big to fit inside the multiple document interface (MDI) frame window or the screen.

Remarks Call **ResizeParentToFit** to let the size of your view dictate the size of its frame window. This is recommended only for views in MDI child frame windows. Use **ResizeParentToFit** in the **OnInitialUpdate** handler function of your derived

CScrollView class. For an example of this member function, see **CScrollView::SetScrollSizes**.

See Also **CView::OnInitialUpdate**, **CScrollView::SetScrollSizes**

CScrollView::ScrollToPosition

void ScrollToPosition(POINT *pt* **);**

pt The point to scroll to, in logical units. The **cx** member must be a positive value (greater than or equal to 0, up to the total size of the view). The same is true for the **cy** member when the mapping mode is **MM_TEXT**. The **cy** member is negative in mapping modes other than **MM_TEXT**.

Remarks Call **ScrollToPosition** to scroll to a given point in the view. The view will be scrolled so that this point is at the upper-left corner of the window. This member function must not be called if the view is scaled to fit.

See Also **CScrollView::GetDeviceScrollPosition**, **CScrollView::SetScaleToFitSize**, **CScrollView::SetScrollSizes**

CScrollView::SetScaleToFitSize

void SetScaleToFitSize(SIZE *sizeTotal* **);**

sizeTotal The horizontal and vertical sizes to which the view is to be scaled. The scroll view's size is measured in logical units. The horizontal size is contained in the **cx** member. The vertical size is contained in the **cy** member. Both **cx** and **cy** must be greater than or equal to 0.

Remarks Call **SetScaleToFitSize** when you want to scale the viewport size to the current window size automatically. With scroll bars, only a portion of the logical view may be visible at any time. But with the scale-to-fit capability, the view has no scroll bars and the logical view is stretched or shrunk to exactly fit the window's client area. When the window is resized, the view draws its data at a new scale based on the size of the window.

You'll typically place the call to **SetScaleToFitSize** in your override of the view's **OnInitialUpdate** member function. If you don't want automatic scaling, call the **SetScrollSizes** member function instead.

SetScaleToFitSize can be used to implement a "Zoom to Fit" operation. Use **SetScrollSizes** to reinitialize scrolling.

See Also **CScrollView::SetScrollSizes**, **CView::OnInitialUpdate**

CScrollView::SetScrollSizes

void SetScrollSizes(int *nMapMode*, **SIZE** *sizeTotal*, **const SIZE&** *sizePage* **= sizeDefault, const SIZE&** *sizeLine* **= sizeDefault);**

nMapMode The mapping mode to set for this view. Possible values include:

Mapping Mode	Logical Unit	Positive y-axis Extends...
MM_TEXT	1 pixel	Downward
MM_HIMETRIC	0.01 mm	Upward
MM_TWIPS	1/1440 in	Upward
MM_HIENGLISH	0.001 in	Upward
MM_LOMETRIC	0.1 mm	Upward
MM_LOENGLISH	0.01 in	Upward

All of these modes are defined by Windows. Two standard mapping modes, **MM_ISOTROPIC** and **MM_ANISOTROPIC**, are not used for **CScrollView**. The class library provides the **SetScaleToFitSize** member function for scaling the view to window size. Column three in the table above describes the coordinate orientation.

sizeTotal The total size of the scroll view. The **cx** member contains the horizontal extent. The **cy** member contains the vertical extent. Sizes are in logical units. Both **cx** and **cy** must be greater than or equal to 0.

sizePage The horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll-bar shaft. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

sizeLine The horizontal and vertical amounts to scroll in each direction in response to a mouse click in a scroll arrow. The **cx** member contains the horizontal amount. The **cy** member contains the vertical amount.

Remarks Call **SetScrollSizes** when the view is about to be updated. Call it in your override of the **OnUpdate** member function to adjust scrolling characteristics when, for example, the document is initially displayed or when it changes size.

You will typically obtain size information from the view's associated document by calling a document member function, perhaps called `GetMyDocSize`, that you supply with your derived document class. The following code shows this approach:

```
SetScrollSizes( nMapMode, GetDocument( )->GetMyDocSize( ) );
```

Alternatively, you might sometimes need to set a fixed size, as in the following code:

```
SetScrollSizes( nMapMode, CSize(100, 100) );
```

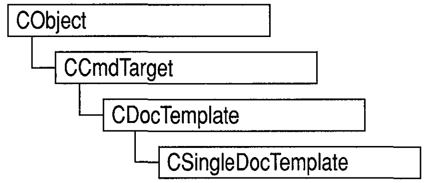
You must set the mapping mode to any of the Windows mapping modes except **MM_ISOTROPIC** or **MM_ANISOTROPIC**. If you want to use an unconstrained mapping mode, call the **SetScaleToFitSize** member function instead of **SetScrollSizes**.

See Also **CScrollView::SetScaleToFitSize**, **CScrollView::GetDeviceScrollSizes**, **CScrollView::GetTotalSize**

Example

```
void CScaleView::OnUpdate( )
{
   // ...
   // Implement a GetDocSize( ) member function in
   // your document class; it returns a CSize.
   SetScrollSizes( MM_LOENGLISH, GetDocument( )->GetDocSize( ) );
   ResizeParentToFit( );   // Default bShrinkOnly argument
   // ...
}
```

class CSingleDocTemplate : public CDocTemplate



The **CSingleDocTemplate** class defines a document template that implements the single document interface (SDI). An SDI application uses the main frame window to display a document; only one document can be open at a time. For a more detailed description of the SDI, see *The Windows Interface: An Application Design Guide.*

A document template defines the relationship between three types of classes:

- A document class, which you derive from **CDocument**.
- A view class, which displays data from the document class listed above. You can derive this class from **CView**, **CScrollView**, **CFormView**, or **CEditView**. (You can also use **CEditView** directly.)
- A frame window class, which contains the view. For an SDI document template, you can derive this class from **CFrameWnd**, or, if you don't need to customize the behavior of the main frame window, you can use **CFrameWnd** directly without deriving your own class.

An SDI application typically supports one type of document, so it has only one **CSingleDocTemplate** object. Only one document can be open at a time.

You don't need to call any member functions of **CSingleDocTemplate** except the constructor. The framework handles **CSingleDocTemplate** objects internally.

See Also **CDocTemplate**, **CDocument**, **CFrameWnd**, **CMultiDocTemplate**, **CView**, **CWinApp**

Construction/Destruction — Public Members

CSingleDocTemplate Constructs a **CSingleDocTemplate** object.

Member Functions

CSingleDocTemplate::CSingleDocTemplate

CSingleDocTemplate(UINT *nIDResource*, **CRuntimeClass*** *pDocClass*, **CRuntimeClass*** *pFrameClass*, **CRuntimeClass*** *pViewClass* **);**

nIDResource Specifies the ID of the resources used with the document type. This may include menu, icon, accelerator table, and string resources.

The string resource consists of up to seven substrings separated by the '\n' character (the '\n' character is needed as a placeholder if a substring is not included; however, trailing '\n' characters are not necessary); these substrings describe the document type. For information about the substrings, see **CDocTemplate::GetDocString**. This string resource is found in the application's resource file. For example:

```
// MYCALC.RC
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME "MyCalc Windows Application\nSheet\nWorksheet\n
Worksheets (*.myc)\n.myc\nMyCalcSheet\n MyCalc Worksheet"
END
```

You can edit this string using the String Editor in App Studio; the entire string appears as a single entry in the String Editor, not as seven separate entries.

For more information about these resource types, see the *App Studio User's Guide*.

pDocClass Points to the **CRuntimeClass** object of the document class. This class is a **CDocument**-derived class you define to represent your documents.

pFrameClass Points to the **CRuntimeClass** object of the frame window class. This class can be a **CFrameWnd**-derived class, or it can be **CFrameWnd** itself if you want default behavior for your main frame window.

pViewClass Points to the **CRuntimeClass** object of the view class. This class is a **CView**-derived class you define to display your documents.

Remarks

Constructs a **CSingleDocTemplate** object. Dynamically allocate a **CSingleDocTemplate** object and pass it to **CWinApp::AddDocTemplate** from the `InitInstance` member function of your application class.

See Also

CDocTemplate::GetDocString, **CWinApp::AddDocTemplate**, **CWinApp::InitInstance**, **CRuntimeClass**, **RUNTIME_CLASS**

Example

```
BOOL CMyApp::InitInstance()
{
        // ...
        // Establish the document type
        // supported by the application

        AddDocTemplate( new CSingleDocTemplate( IDR_MAINFRAME,
                            RUNTIME_CLASS( CSheetDoc ),
                            RUNTIME_CLASS( CFrameWnd ),
                            RUNTIME_CLASS( CSheetView ) ) );

        // ...
}
```

class CSize : public tagSIZE

The **CSize** class is similar to the Windows **SIZE** structure, which implements a relative coordinate or position. Because **CSize** derives from **tagSIZE**, **CSize** objects may be used as **SIZE** structures. The operators of this class that interact with a "size" accept either **CSize** objects or **SIZE** structures.

The **cx** and **cy** members of **SIZE** (and **CSize**) are public. In addition, **CSize** implements member functions to manipulate the **SIZE** structure.

#include <afxwin.h>

SIZE Structure

A **SIZE** structure has this form:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

Members

cx Specifies the x-extent when a function returns.

cy Specifies the y-extent when a function returns.

Some extended functions of Windows version 3.1 place viewport extents, window extents, text extents, bitmap dimensions, and the aspect-ratio filter in the **SIZE** structure.

See Also

CRect, **CPoint**

Construction/Destruction—Public Members

CSize	Constructs a **CSize** object.

Operators—Public Members

operator ==	Checks for equality between **CSize** and a size.
operator !=	Checks for inequality between **CSize** and a size.
operator +=	Adds a size to **CSize**.
operator –=	Subtracts a size from **CSize**.

Operators Returning CSize Values—Public Members

operator +	Adds two sizes.
operator –	Subtracts two sizes.

Member Functions

CSize::CSize

CSize( );

CSize(int *initCX*, **int** *initCY* **);**

CSize(SIZE *initSize* **);**

CSize(POINT *initPt* **);**

CSize(DWORD *dwSize* **);**

initCX Sets the **cx** member for the **CSize**.

initCY Sets the **cy** member for the **CSize**.

initSize **SIZE** structure or **CSize** object used to initialize **CSize**.

initPt **POINT** structure or **CPoint** object used to initialize **CSize**.

dwSize **DWORD** used to initialize **CSize**. The low-order word is the **cx** member and the high-order word is the **cy** member.

Remarks Constructs a **CSize** object. If no arguments are given, **cx** and **cy** members are not initialized.

Operators

CSize::operator ==

BOOL operator ==(SIZE *size* **) const;**

Remarks Checks for equality between two sizes.

Return Value Nonzero if the sizes are equal; otherwise 0.

CSize::operator !=

BOOL operator !=(SIZE *size*) const;

Remarks Checks for inequality between two sizes.

Return Value Nonzero if the sizes are not equal; otherwise 0.

CSize::operator +=

void operator +=(SIZE *size*);

Remarks Adds a size to a **CSize**.

CSize::operator –=

void operator –=(SIZE *size*);

Remarks Subtracts a size from a **CSize**.

CSize::operator +

CSize operator +(SIZE *size*) const;

Return Value A **CSize** that is the sum of two sizes.

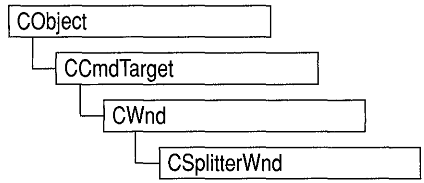
CSize::operator –

CSize operator –(SIZE *size*) const;

CSize operator –() const;

Return Value A **CSize** that is the difference between two sizes.

class CSplitterWnd : public CWnd



The **CSplitterWnd** class provides the functionality of a splitter window, which is a window that contains multiple panes. A pane is usually an application-specific object derived from **CView**, but it can be any **CWnd** object that has the appropriate child window ID.

A **CSplitterWnd** object is usually embedded in a parent **CFrameWnd** or **CMDIChildWnd** object. Create a **CSplitterWnd** object using the following steps:

1. Embed a **CSplitterWnd** member variable in the parent frame.
2. Override the parent frame's **OnCreateClient** member function.
3. From within the overridden **OnCreateClient**, call **CSplitterWnd**'s constructor, then the **Create** or **CreateStatic** member function.

Call the **Create** member function to create a dynamic splitter window. A dynamic splitter window typically is used to create and scroll a number of individual panes, or views, of the same document. The framework automatically creates an initial pane for the splitter; then the framework creates, resizes, and disposes of additional panes as the user operates the splitter window's controls.

When you call **Create**, you specify a minimum row height and column width that determine when the panes are too small to be fully displayed. After you call **Create**, you can adjust these minimums by calling the **SetColumnInfo** and **SetRowInfo** member functions.

Also use the **SetColumnInfo** and **SetRowInfo** member functions to set an "ideal" width for a column and "ideal" height for a row. When the framework displays a splitter window, it first displays the parent frame, then the splitter window. The framework then lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to lower-right corner of the splitter window's client area.

All panes in a dynamic splitter window must be of the same class. Familiar applications that support dynamic splitter windows include Microsoft Word and Microsoft Excel.

Use the **CreateStatic** member function to create a static splitter window. The user can change only the size of the panes in a static splitter window, not their number or order.

You must specifically create all the static splitter's panes when you create the static splitter. Make sure you create all the panes before the parent frame's **OnCreateClient** member function returns, or the framework will not display the window correctly.

The **CreateStatic** member function automatically initializes a static splitter with a minimum row height and column width of 0. After you call **Create**, adjust these minimums by calling the **SetColumnInfo** and **SetRowInfo** member functions. Also use **SetColumnInfo** and **SetRowInfo** after you call **CreateStatic** to indicate desired ideal pane dimensions.

The individual panes of a static splitter often belong to different classes. For examples of static splitter windows, see the App Studio graphics editor and the Windows File Manager.

A splitter window supports special scroll bars (apart from the scroll bars that panes may have). These scroll bars are children of the **CSplitterWnd** object and are shared with the panes.

You create these special scroll bars when you create the splitter window. For example, a **CSplitterWnd** that has one row, two columns, and the **WS_VSCROLL** style will display a vertical scroll bar that is shared by the two panes. When the user moves the scroll bar, **WM_VSCROLL** messages are sent to both panes. When the panes set the scroll-bar position, the shared scroll bar is set.

For further information on splitter windows, see Technical Note 29 in MSVC\HELP\MFCNOTES.HLP. For more information on how to create dynamic splitter windows, see the Scribble sample application in Chapter 8 of the *Class Library User's Guide*, and the VIEWEX example in the MFC\SAMPLES\VIEWEX subdirectory.

include <afxext.h>

See Also **CWnd**

Construction—Public Members

CSplitterWnd	Call to construct a **CSplitterWnd** object.
Create	Call to create a dynamic splitter window and attach it to the **CSplitterWnd** object.
CreateStatic	Call to create a static splitter window and attach it to the **CSplitterWnd** object.
CreateView	Call to create a pane in a splitter window.

Operations—Public Members

GetRowCount	Returns the current pane row count.
GetColumnCount	Returns the current pane column count.
GetRowInfo	Returns information on the specified row.
SetRowInfo	Call to set the specified row information.
GetColumnInfo	Returns information on the specified column.
SetColumnInfo	Call to set the specified column information.
GetPane	Returns the pane at the specified row and column.
IsChildPane	Call to determine if the window is currently a child pane of this splitter window.
IdFromRowCol	Returns the child window ID of the pane at the specified row and column.
RecalcLayout	Call to redisplay the splitter window after adjusting row or column size.

Member Functions

CSplitterWnd::Create

BOOL Create(CWnd* *pParentWnd*, **int** *nMaxRows*, **int** *nMaxCols*, **SIZE** *sizeMin*, **CCreateContext*** *pContext*, **DWORD** *dwStyle* = **WS_CHILD | WS_VISIBLE |WS_HSCROLL | WS_VSCROLL | SPLS_DYNAMIC_SPLIT, UINT** *nID* = **AFX_IDW_PANE_FIRST);**

pParentWnd The parent frame window of the splitter window.

nMaxRows The maximum number of rows in the splitter window. This value must not exceed 2.

nMaxCols The maximum number of columns in the splitter window. This value must not exceed 2.

sizeMin Specifies the minimum size at which a pane may be displayed.

pContext A pointer to a **CCreateContext** structure. In most cases, this can be the *pContext* passed to the parent frame window.

dwStyle Specifies the window style.

nID The child window ID of the window. The ID can be **AFX_IDW_PANE_FIRST** unless the splitter window is nested inside another splitter window.

Remarks

To create a dynamic splitter window, first call the constructor, then call the **Create** member function.

You can embed a **CSplitterWnd** in a parent **CFrameWnd** or **CMDIChildWnd** object by taking the following steps:

1. Embed a **CSplitterWnd** member variable in the parent frame.
2. Override the parent frame's **OnCreateClient** member function.
3. Call the **CSplitterWnd** constructor and the **Create** member function from within the overridden **OnCreateClient**.

When you create a splitter window from within a parent frame, pass the parent frame's *pContext* parameter to the splitter window. Otherwise, this parameter can be **NULL**.

The initial minimum row height and column width of a dynamic splitter window are set by the *sizeMin* parameter. These minimums, which determine if a pane is too small to be shown in its entirety, can be changed with the **SetRowInfo** and **SetColumnInfo** member functions.

For more on dynamic splitter windows, see Chapter 4 in this manual, Technical Note 29 in MFCNOTES.HLP, and the **CSplitterWnd** class overview.

Return Value

Nonzero if successful; otherwise 0.

See Also

CSplitterWnd, **CSplitterWnd::CreateStatic**, **CFrameWnd::OnCreateClient**, **CMDIChildWnd::OnCreateClient**, **CSplitterWnd::SetRowInfo**, **CSplitterWnd::SetColumnInfo**, **CSplitterWnd::CreateView**

CSplitterWnd::CreateStatic

BOOL CreateStatic(CWnd* *pParentWnd*, **int** *nRows*, **int** *nCols*, **DWORD** *dwStyle* = **WS_CHILD | WS_VISIBLE, UINT** *nID* = **AFX_IDW_PANE_FIRST);**

pParentWnd The parent frame window of the splitter window.

nRows The number of rows. This value must not exceed 16.

nCols The number of columns. This value must not exceed 16.

dwStyle Specifies the window style.

nID The child window ID of the window. The ID can be **AFX_IDW_PANE_FIRST** unless the splitter window is nested inside another splitter window.

Remarks

To create a static splitter window, first call the constructor, then call the **CreateStatic** member function.

A **CSplitterWnd** is usually embedded in a parent **CFrameWnd** or **CMDIChildWnd** object by taking the following steps:

1. Embed a **CSplitterWnd** member variable in the parent frame.
2. Override the parent frame's **OnCreateClient** member function.
3. Call the **CSplitterWnd** constructor and the **CreateStatic** member function from within the overridden **OnCreateClient**.

A static splitter window contains a fixed number of panes, often from different classes.

When you create a static splitter window, you must at the same time create all its panes. The **CreateView** member function is usually used for this purpose, but you can create other nonview classes as well.

The initial minimum row height and column width for a static splitter window is 0. These minimums, which determine when a pane is too small to be shown in its entirety, can be changed with the **SetRowInfo** and **SetColumnInfo** member functions.

To add scroll bars to a static splitter window, add the **WS_HSCROLL** and **WS_VSCROLL** styles to *dwStyle*.

See Chapter 4 in this manual, Technical Note 29 in MFCNOTES.HLP, and the **CSplitterWnd** class description for more on static splitter windows.

Return Value

Nonzero if successful; otherwise 0.

See Also

CSplitterWnd, **CSplitterWnd::Create**, **CFrameWnd::OnCreateClient**, **CMDIChildWnd::OnCreateClient**, **CSplitterWnd::SetRowInfo**, **CSplitterWnd::SetColumnInfo**, **CSplitterWnd::CreateView**

CSplitterWnd::CreateView

virtual BOOL CreateView(int *row*, **int** *col*, **CRuntimeClass*** *pViewClass*, **SIZE** *sizeInit*, **CCreateContext*** *pContext* **);**

Parameters

row Specifies the splitter window row in which to place the new view.

col Specifies the splitter window column in which to place the new view.

pViewClass Specifies the **CRuntimeClass** of the new view.

sizeInit Specifies the initial size of the new view.

pContext A pointer to a creation context used to create the view (usually the *pContext* passed into the parent frame's overridden **OnCreateClient** member function in which the splitter window is being created).

Remarks

Call this member function to create the panes for a static splitter window. All panes of a static splitter window must be created before the framework displays the splitter.

The framework also calls this member function to create new panes when the user of a dynamic splitter window splits a pane, row, or column.

Return Value

Nonzero if successful; otherwise 0.

See Also

CSplitterWnd::Create

CSplitterWnd::CSplitterWnd

CSplitterWnd( );

Remarks

Construct a **CSplitterWnd** object in two steps. First call the constructor, which creates the **CSplitterWnd** object, then call the **Create** member function, which creates the splitter window and attaches it to the **CSplitterWnd** object.

See Also

CSplitterWnd::Create

CSplitterWnd::GetColumnCount

int GetColumnCount( );

Return Value Returns the current number of columns in the splitter. For a static splitter this will also be the maximum number of columns.

See Also **CSplitterWnd::GetRowCount**

CSplitterWnd::GetColumnInfo

void GetColumnInfo(int *col*, **int&** *cxCur*, **int&** *cxMin* **);**

col Specifies a column.

cxCur A reference to an **int** to be set to the current width of the column.

cxMin A reference to an **int** to be set to the current minimum width of the column.

Remarks Call this member function to obtain information about the specified column.

See Also **CSplitterWnd::SetColumnInfo**, **CSplitterWnd::GetRowInfo**

CSplitterWnd::GetPane

CWnd* GetPane(int *row*, **int** *col* **);**

row Specifies a row.

col Specifies a column.

Return Value Returns the pane at the specified row and column. The returned pane is usually a **CView**-derived class.

See Also **CSplitterWnd::IdFromRowCol**, **CSplitterWnd::IsChildPane**

CSplitterWnd::GetRowCount

int GetRowCount( );

Return Value Returns the current number of rows in the splitter window. For a static splitter window, this will also be the maximum number of rows.

See Also **CSplitterWnd::GetColumnCount**

CSplitterWnd::GetRowInfo

void GetRowInfo(int *row*, **int&** *cyCur*, **int&** *cyMin* **);**

row Specifies a row.

cyCur Reference to **int** to be set to the current height of the row in pixels.

cyMin Reference to **int** to be set to the current minimum height of the row in pixels.

Remarks Call this member function to obtain information about the specified row.

Return Value The *cyCur* parameter is filled with the current height of the specified row, and *cyMin* is filled with the minimum height of the row.

See Also **CSplitterWnd::SetRowInfo**, **CSplitterWnd::GetColumnInfo**

CSplitterWnd::IdFromRowCol

int IdFromRowCol(int *row*, **int** *col* **);**

row Specifies the splitter window row.

col Specifies the splitter window column.

Remarks Call this member function to obtain the child window ID for the pane at the specified row and column. This member function is used for creating nonviews as panes and may be called before the pane exists.

Return Value The child window ID for the pane.

See Also **CSplitterWnd::GetPane**, **CSplitterWnd::IsChildPane**

CSplitterWnd::IsChildPane

BOOL IsChildPane(CWnd* *pWnd*, **int&** *row*, **int&** *col* **);**

pWnd A pointer to a **CWnd** object to be tested.

row Reference to an **int** in which to store row number.

col Reference to an **int** in which to store a column number.

Remarks Call this member function to determine whether *pWnd* is currently a child pane of this splitter window.

Return Value If nonzero, *pWnd* is currently a child pane of this splitter window, and *row* and *col* are filled in with the position of the pane in the splitter window. If *pWnd* is not a child pane of this splitter window, 0 is returned.

See Also **CSplitterWnd::GetPane**

CSplitterWnd::RecalcLayout

void RecalcLayout( );

Remarks Call this member function to correctly redisplay the splitter window after you have adjusted row and column sizes with the **SetRowInfo** and **SetColumnInfo** member functions. If you change row and column sizes as part of the creation process before the splitter window is visible, it is not necessary to call this member function.

The framework calls this member function whenever the user resizes the splitter window or moves a split.

See Also **CSplitterWnd::SetRowInfo**, **CSplitterWnd::SetColumnInfo**

CSplitterWnd::SetColumnInfo

void SetColumnInfo(int *col*, **int** *cxIdeal*, **int** *cxMin* **);**

col Specifies a splitter window column.

cxIdeal Specifies an ideal width for the splitter window column in pixels.

cxMin Specifies a minimum width for the splitter window column in pixels.

Remarks Call this member function to set a new minimum width and ideal width for a column. The column minimum value determines when the column will be too small to be fully displayed.

When the framework displays the splitter window, it lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to lower-right corner of the splitter window's client area.

See Also **CSplitterWnd::GetRowInfo**, **CSplitterWnd::RecalcLayout**

CSplitterWnd::SetRowInfo

void SetRowInfo(int *row*, **int** *cyIdeal*, **int** *cyMin* **);**

row Specifies a splitter window row.

cyIdeal Specifies an ideal height for the splitter window row in pixels.

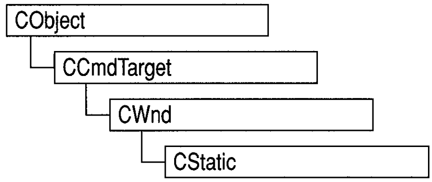
cyMin Specifies a minimum height for the splitter window row in pixels.

Remarks Call this member function to set a new minimum height and ideal height for a row. The row minimum value determines when the row will be too small to be fully displayed.

When the framework displays the splitter window, it lays out the panes in columns and rows according to their ideal dimensions, working from the upper-left to lower-right corner of the splitter window's client area.

See Also **CSplitterWnd::GetRowInfo**, **CSplitterWnd::SetColumnInfo**, **CSplitterWnd::RecalcLayout**

class CStatic : public CWnd



The **CStatic** class provides the functionality of a Windows static control. A static control is a simple text field, box, or rectangle that can be used to label, box, or separate other controls. A static control takes no input and provides no output.

Create a static control in two steps. First, call the constructor **CStatic** to construct the **CStatic** object, then call the **Create** member function to create the static control and attach it to the **CStatic** object.

If you create a **CStatic** object within a dialog box (through a dialog resource), the **CStatic** object is automatically destroyed when the user closes the dialog box. If you create a **CStatic** object within a window, you may also need to destroy it. A **CStatic** object created on the stack within a window is automatically destroyed. If you create the **CStatic** object on the heap by using the **new** function, you must call **delete** on the object to destroy it when the user terminates the Windows static control.

#include <afxwin.h>

See Also **CWnd**, **CButton**, **CComboBox**, **CEdit**, **CListBox**, **CScrollBar**, **CDialog**

Construction/Destruction—Public Members

CStatic	Constructs a **CStatic** object.

Initialization—Public Members

Create	Creates the Windows static control and attaches it to the **CStatic** object.

Operations—Public Members

SetIcon	Associates an icon with an icon resource.
GetIcon	Retrieves the handle of the icon associated with an icon resource.

Member Functions

CStatic::Create

BOOL Create(LPCSTR *lpszText*, **DWORD** *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* = **0xffff);**

lpszText Specifies the text to place in the control. If **NULL**, no text will be visible.

dwStyle Specifies the static control's window style.

rect Specifies the position and size of the static control. It can be either a **RECT** structure or a **CRect** object.

pParentWnd Specifies the **CStatic** parent window, usually a **CDialog** object. It must not be **NULL**.

nID Specifies the static control's control ID.

Remarks

Construct a **CStatic** object in two steps. First call the constructor **CStatic**, then call **Create**, which creates the Windows static control and attaches it to the **CStatic** object. Apply the following window styles to a static control:

- **WS_CHILD** Always
- **WS_VISIBLE** Usually
- **WS_DISABLED** Rarely

See **Create** in the **CWnd** base class for a full description of these window styles.

Return Value

Nonzero if successful; otherwise 0.

Static Styles

You can use any combination of the following static control styles for *dwStyle*:

- **SS_BLACKFRAME** Specifies a box with a frame drawn with the same color as window frames. The default is black.
- **SS_BLACKRECT** Specifies a rectangle filled with the color used to draw window frames. The default is black.
- **SS_CENTER** Designates a simple rectangle and displays the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next centered line.

- **SS_GRAYFRAME** Specifies a box with a frame drawn with the same color as the screen background (desktop). The default is gray.
- **SS_GRAYRECT** Specifies a rectangle filled with the color used to fill the screen background. The default is gray.
- **SS_ICON** Designates an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. The *nWidth* and *nHeight* parameters are ignored; the icon automatically sizes itself.
- **SS_LEFT** Designates a simple rectangle and displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
- **SS_LEFTNOWORDWRAP** Designates a simple rectangle and displays the given text flush-left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
- **SS_NOPREFIX** Unless this style is specified, the Windows operating system will interpret any ampersand (&) characters in the control's text to be accelerator prefix characters. In this case, the ampersand (&) is removed and the next character in the string is underlined. If a static control is to contain text where this feature is not wanted, **SS_NOPREFIX** may be added. This static-control style may be included with any of the defined static controls. You can combine **SS_NOPREFIX** with other styles by using the bitwise-OR operator. This is most often used when filenames or other strings that may contain an ampersand (&) need to be displayed in a static control in a dialog box.
- **SS_RIGHT** Designates a simple rectangle and displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-right line.
- **SS_SIMPLE** Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the **WM_CTLCOLOR** message.)
- **SS_USERITEM** Specifies a user-defined item.
- **SS_WHITEFRAME** Specifies a box with a frame drawn with the same color as the window background. The default is white.
- **SS_WHITERECT** Specifies a rectangle filled with the color used to fill the window background. The default is white.

See Also **CStatic::CStatic**

CStatic::CStatic

CStatic( );

Remarks Constructs a **CStatic** object.

See Also **CStatic::Create**

CStatic::GetIcon

Windows 3.1 Only **HICON GetIcon( ) const;** ♦

Return Value Returns the handle of the icon associated with an icon resource. This function should be called only for **CStatic** objects that represent icons created with the **SS_ICON** style.

See Also **STM_GETICON**, **CStatic::SetIcon**

CStatic::SetIcon

Windows 3.1 Only **HICON SetIcon(HICON** *hIcon* **);** ♦

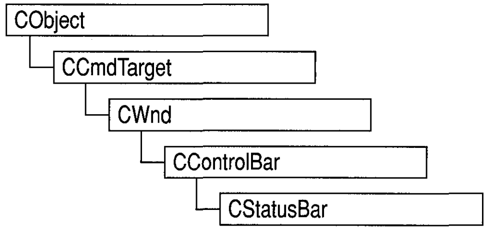
hIcon Identifies the icon to associate with an icon resource.

Remarks Associates an icon with an icon resource. This is a **CStatic** object created with the **SS_ICON** style.

Return Value The handle of the icon that was previously associated with the icon resource; 0 if an error occurred.

See Also **STM_SETICON**, **::LoadIcon**, **CStatic::GetIcon**

class CStatusBar : public CControlBar



A **CStatusBar** object is a control bar with a row of text output panes, or "indicators." The output panes commonly are used as message lines and as status indicators. Examples include the menu help-message lines that briefly explain the selected menu command and the indicators that show the status of the SCROLL LOCK, NUM LOCK, and other keys.

The framework stores indicator information in an array with the leftmost indicator at position 0. When you create a status bar, you use an array of string IDs that the framework associates with the corresponding indicators. You can then use either a string ID or an index to access an indicator.

By default, the first indicator is "stretchy": it takes up the status-bar length not used by the other indicator panes, so that the other panes are right-aligned.

To create a status bar, follow these steps:

1. Construct the **CStatusBar** object.
2. Call the **Create** function to create the status-bar window and attach it to the **CStatusBar** object.
3. Call **SetIndicators** to associate a string ID with each indicator.

There are three ways to update the text in a status-bar pane:

1. Call **SetWindowText** to update the text in pane 0 only.
2. Call **SetText** in the status bar's **ON_UPDATE_COMMAND_UI** handler.
3. Call **SetPaneText** to update the text for any pane.

#include <afxext.h>

See Also **CControlBar**, **CWnd::SetWindowText**, **CStatusBar::SetIndicators**

Construction/Destruction—Public Members

CStatusBar	Constructs a **CStatusBar** object.
Create	Creates the Windows status bar, attaches it to the **CStatusBar** object, and sets the initial font and bar height.
SetIndicators	Sets indicator IDs.

Attributes—Public Members

CommandToIndex	Gets index for a given indicator ID.
GetItemID	Gets indicator ID for a given index.
GetItemRect	Gets display rectangle for a given index.
GetPaneText	Gets indicator text for a given index.
SetPaneText	Sets indicator text for a given index.
GetPaneInfo	Gets indicator ID, style, and width for a given index.
SetPaneInfo	Sets indicator ID, style, and width for a given index.

Member Functions

CStatusBar::CommandToIndex

int CommandToIndex(UINT *nIDFind* **) const;**

nIDFind String ID of the indicator whose index is to be retrieved.

Remarks Gets the indicator index for a given ID. The index of the first indicator is 0.

Return Value The index of the indicator if successful; –1 if not successful.

See Also **CStatusBar::GetItemID**

CStatusBar::Create

BOOL Create(CWnd* *pParentWnd*,
DWORD *dwStyle* = **WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,**
UINT *nID* = **AFX_IDW_STATUS_BAR);**

pParentWnd Pointer to the **CWnd** object whose Windows window is the parent of the status bar.

dwStyle The status-bar style. In addition to the standard Windows styles, these styles are supported:

- **CBRS_TOP** Control bar is at top of frame window.
- **CBRS_BOTTOM** Control bar is at bottom of frame window.
- **CBRS_NOALIGN** Control bar is not repositioned when the parent is resized.

nID The tool bar's child-window ID.

Remarks Creates a status bar (a child window) and associates it with the **CStatusBar** object. Also sets the initial font and sets the status bar's height to a default value.

Return Value Nonzero if successful; otherwise 0.

See Also **CStatusBar::SetIndicators**

CStatusBar::CStatusBar

CStatusBar( );

Remarks Constructs a **CStatusBar** object, creates a default status-bar font if necessary, and sets the font characteristics to default values.

See Also **CStatusBar::Create**

CStatusBar::GetItemID

UINT GetItemID(int *nIndex* **) const;**

nIndex Index of the indicator whose ID is to be retrieved.

Remarks Returns the ID of the indicator specified by *nIndex*.

Return Value The ID of the indicator specified by *nIndex*.

See Also **CStatusBar::CommandToIndex**

CStatusBar::GetItemRect

void GetItemRect(int *nIndex*, **LPRECT** *lpRect* **) const;**

nIndex Index of the indicator whose rectangle coordinates are to be retrieved.

lpRect Points to a **RECT** structure or a **CRect** object that will receive the coordinates of the indicator specified by *nIndex*.

Remarks

Copies the coordinates of the indicator specified by *nIndex* into the structure pointed to by *lpRect*. Coordinates are in pixels relative to the upper-left corner of the status bar.

See Also

CStatusBar::CommandToIndex, **CStatusBar::GetPaneInfo**

CStatusBar::GetPaneInfo

void GetPaneInfo(int *nIndex*, **UINT&** *nID*, **UINT&** *nStyle*, **int&** *cxWidth* **) const;**

nIndex Index of the pane whose information is to be retrieved.

nID Reference to a **UINT** that is set to the ID of the pane.

nStyle Reference to a **UINT** that is set to the style of the pane.

cxWidth Reference to an integer that is set to the width of the pane.

Remarks

Sets *nID*, *nStyle*, and *cxWidth* to the ID, style, and width of the indicator pane at the location specified by *nIndex*.

See Also

CStatusBar::SetPaneInfo, **CStatusBar::GetItemID**, **CStatusBar::GetItemRect**

CStatusBar::GetPaneText

void GetPaneText(int *nIndex*, **CString&** *s* **) const;**

nIndex Index of the pane whose text is to be retrieved.

s Reference to a **CString** object to which the pane's text is copied.

Remarks Copies the pane's text to the **CString** object.

See Also **CStatusBar::SetPaneText**

CStatusBar::SetIndicators

BOOL SetIndicators(const UINT FAR* *lpIDArray*, **int** *nIDCount* **);**

lpIDArray Pointer to an array of IDs.

nIDCount Number of elements in the array pointed to by *lpIDArray*.

Remarks Sets each indicator's ID to the value specified by the corresponding element of the array *lpIDArray*, loads the string resource specified by each ID, and sets the indicator's text to the string.

Return Value Nonzero if successful; otherwise 0.

See Also **CStatusBar::CStatusBar**, **CStatusBar::Create**, **CStatusBar::SetPaneInfo**, **CStatusBar::SetPaneText**

CStatusBar::SetPaneInfo

void SetPaneInfo(int *nIndex*, **UINT** *nID*, **UINT** *nStyle*, **int** *cxWidth* **);**

nIndex Index of the indicator pane whose style is to be set.

nID New ID for the indicator pane.

nStyle New style for the indicator pane.

cxWidth New width for the indicator pane.

Remarks Sets the specified indicator pane to a new ID, style, and width.

The following indicator styles are supported:

- **SBPS_NOBORDERS** No 3-D border around the pane.
- **SBPS_POPOUT** Reverse border so that text "pops out."
- **SBPS_DISABLED** Do not draw text.

- **SBPS_STRETCH** Stretch pane to fill unused space. Only one pane per status bar can have this style.
- **SBPS_NORMAL** No stretch, borders, or pop-out.

See Also **CStatusBar::GetPaneInfo**

CStatusBar::SetPaneText

BOOL SetPaneText(int *nIndex*, **LPCSTR** *lpszNewText*,
BOOL *bUpdate* = **TRUE);**

nIndex Index of the pane whose text is to be set.

lpszNewText Pointer to the new pane text.

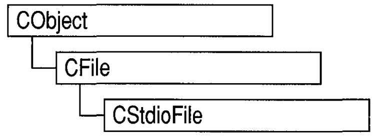
bUpdate If **TRUE**, the pane is invalidated after the text is set.

Remarks Sets the pane text to the string pointed to by *lpszNewText*.

Return Value Nonzero if successful; otherwise 0.

See Also **CStatusBar::GetPaneText**

class CStdioFile : public CFile



A **CStdioFile** object represents a C run-time stream file as opened by the **fopen** function. Stream files are buffered and can be opened in either text mode (the default) or binary mode. Text mode provides special processing for carriage return–linefeed pairs. When you write a newline character (0x0A) to a text-mode **CStdioFile** object, the byte pair (0x0A, 0x0D) is sent to the file. When you read, the byte pair (0x0A, 0x0D) is translated to a single 0x0A byte.

The **CFile** functions **Duplicate**, **LockRange**, and **UnlockRange** are not implemented for **CStdioFile**. If you call these functions on a **CStdioFile**, you will get a **CNotSupportedException**.

#include <afx.h>

Data Members — Public Members

m_pStream	Contains a pointer to an open file.

Construction/Destruction — Public Members

CStdioFile	Constructs a **CStdioFile** object from a path or file pointer.

Text Read/Write — Public Members

ReadString	Reads a single line of text.
WriteString	Writes a single line of text.

Member Functions

CStdioFile::CStdioFile

CStdioFile( );

CStdioFile(FILE* *pOpenStream* **);**

CStdioFile(const char* *pszFileName*, **UINT** *nOpenFlags* **)**
throw(CFileException);

pOpenStream Specifies the file pointer returned by a call to the C run-time function **fopen**.

pszFileName Specifies a string that is the path to the desired file. The path can be relative or absolute.

nOpenFlags Sharing and access mode. Specifies the action to take when the file is opened. You can combine options by using the bitwise-OR (|) operator. One access permission and a text-binary specifier are required; the **create** and **noInherit** modes are optional. See **CFile::CFile** for a list of mode options. The share flags do not apply.

Remarks

The default version of the constructor works in conjunction with the **CFile::Open** member function to test errors. The one-parameter version constructs a **CStdioFile** object from a pointer to a file that is already open. Allowed pointer values include the predefined input/output file pointers **stdin**, **stdout**, or **stderr**. The two-parameter version constructs a **CStdioFile** object and opens the corresponding operating-system file with the given path. **CFileException** is thrown if the file cannot be opened or created.

Example

```
char* pFileName = "test.dat";
CStdioFile f1;
if( !f1.Open( pFileName,
        CFile::modeCreate | CFile::modeWrite | CFile::typeText ) ) {
    #ifdef _DEBUG
        afxDump << "Unable to open file" << "\n";
    #endif
    exit( 1 );
}
CStdioFile f2( stdout );
TRY
{
    CStdioFile f3( pFileName,
        CFile::modeCreate | CFile::modeWrite | CFile::typeText );
}
CATCH( CFileException, e )
{
    #ifdef _DEBUG
        afxDump << "File could not be opened " << e->m_cause << "\n";
    #endif
}
END_CATCH
```

CStdioFile::ReadString

virtual char FAR* ReadString(char FAR* *lpsz*, **UINT** *nMax* **)**
throw(CFileException);

lpsz Specifies a pointer to a user-supplied buffer that will receive a null-terminated text string.

nMax Specifies the maximum number of characters to read. Should be one less than the size of the *lpsz* buffer.

Remarks

Reads text data into a buffer, up to a limit of *nMax*–1 characters, from the file associated with the **CStdioFile** object. Reading is stopped by a carriage return–linefeed pair. If, in that case, fewer than *nMax*–1 characters have been read, a newline character is stored in the buffer. A null character ('\0') is appended in either case. **CFile::Read** is also available for text-mode input, but it does not terminate on a carriage return–linefeed pair.

Return Value

A pointer to the buffer containing the text data; **NULL** if end-of-file was reached.

Example

```
extern CStdioFile f;
char buf[100];

f.ReadString( buf, 100 );
```

CStdioFile::WriteString

virtual void WriteString(const char FAR* *lpsz* **)**
throw(CFileException);

lpsz Specifies a pointer to a buffer containing a null-terminated text string.

Remarks

Writes data from a buffer to the file associated with the **CStdioFile** object. The terminating null character ('\0') is not written to the file. A newline character is written as a carriage return–linefeed pair. **WriteString** throws an exception in response to several conditions, including the disk-full condition.

This is a text-oriented write function available only to **CStdioFile** and its descendents. **CFile::Write** is also available, but rather than terminating on a null character, it writes the requested number of bytes to the file.

Example

```
extern CStdioFile f;
char buf[] = "test string";

f.WriteString( buf );
```

Data Members

CStdioFile::m_pStream

Remarks

The **m_pStream** data member is the pointer to an open file as returned by the C run-time function **fopen**. It is **NULL** if the file has never been opened or has been closed.

class CString

A **CString** object consists of a variable-length sequence of characters. The **CString** class provides a variety of functions and operators that manipulate **CString** objects using a syntax similar to that of Basic. Concatenation and comparison operators, together with simplified memory management, make **CString** objects easier to use than ordinary character arrays. The increased processing overhead is not significant. The **CString** "Application Notes" section offers useful information on:

- **CString** Exception Cleanup
- **CString** Argument Passing

The maximum size of a **CString** object is **MAXINT** (32,767) characters. The **const char*** operator gives direct access to the characters in a **CString** object, which makes it look like a C-language character array. Unlike a character array, however, the **CString** class has a built-in memory-allocation capability. This allows string objects to grow as a result of concatenation operations. No attempt is made to fold **CString** objects. If you make two **CString** objects containing `Chicago`, for example, the characters in `Chicago` are stored in two places. The **CString** class is not implemented as a Microsoft Foundation Class Library collection class, although **CString** objects can certainly be stored as elements in collections.

The overloaded **const char*** conversion operator allows **CString** objects to be freely substituted for character pointers in function calls. The **CString(const char*** *psz*) constructor allows character pointers to be substituted for **CString** objects. Use the **GetBuffer** and **ReleaseBuffer** member functions when you need to directly access a **CString** as a nonconstant pointer to **char** (**char*** instead of a **const char***).

CString objects follow "value semantics." A **CString** object represents a unique value. Think of a **CString** as an actual string, not as a pointer to a string. Where possible, allocate **CString** objects on the frame rather than on the heap. This saves memory and simplifies parameter passing.

#include <afx.h>

Construction/Destruction — Public Members

CString	Constructs **CString** objects in various ways.
~CString	Destroys a **CString** object.

The String as an Array—Public Members

GetLength	Returns the number of characters in a **CString** object.
IsEmpty	Tests whether the length of a **CString** object is 0.
Empty	Forces a string to have 0 length.
GetAt	Returns the character at a given position.
operator [ ]	Returns the character at a given position—operator substitution for **GetAt**.
SetAt	Sets a character at a given position.
operator const char* ( )	Directly accesses characters stored in a **CString** object.

Assignment/Concatenation—Public Members

operator =	Assigns a new value to a **CString** object.
operator +	Concatenates two strings and returns a new string.
operator +=	Concatenates a new string to the end of an existing string.

Comparison—Public Members

operator ==, <, etc.	Comparison operators (ASCII, case sensitive).
Compare	Compares two strings (ASCII, case sensitive).
CompareNoCase	Compares two strings (ASCII, case insensitive).
Collate	Compares two strings with proper language-dependent ordering.

Extraction—Public Members

Mid	Extracts the middle part of a string (like the Basic MID$ command).
Left	Extracts the left part of a string (like the Basic LEFT$ command).
Right	Extracts the right part of a string (like the Basic RIGHT$ command).
SpanIncluding	Extracts a substring that contains only the characters in a set.
SpanExcluding	Extracts a substring that contains only the characters not in a set.

Other Conversions—Public Members

MakeUpper	Converts all the characters in this string to uppercase characters.
MakeLower	Converts all the characters in this string to lowercase characters.
MakeReverse	Reverses the characters in this string.

Searching—Public Members

Find	Finds a character or substring inside a larger string.
ReverseFind	Finds a character inside a larger string; starts from the end.
FindOneOf	Finds the first matching character from a set.

Archive/Dump—Public Members

operator <<	Inserts a **CString** object to an archive or dump context.
operator >>	Extracts a **CString** object from an archive.

Buffer Access—Public Members

GetBuffer	Returns a pointer to the characters in the **CString**.
GetBufferSetLength	Returns a pointer to the characters in the **CString**, truncating to the specified length.
ReleaseBuffer	Yields control of the buffer returned by **GetBuffer**.

Windows-Specific—Public Members

LoadString	Loads an existing **CString** object from a Windows resource.
AnsiToOem	Makes an in-place conversion from the ANSI character set to the OEM character set.
OemToAnsi	Makes an in-place conversion from the OEM character set to the ANSI character set.

Member Functions

CString::AnsiToOem

void AnsiToOem();

Remarks

Converts all the characters in this **CString** object from the ANSI character set to the OEM character set. See the IBM PC Extended Character Set table and the ANSI table in the *Microsoft Windows Programmer's Reference*. This function is available only in the Windows compiled version of the Microsoft Foundation Class Library, and it is declared in AFX.H only if **_WINDOWS** is defined.

See Also

CString::OemToAnsi

Example

```
CString s( '\265' );  // Octal ANSI code for '1/2'
s.AnsiToOem();
ASSERT( s == "\253" ); // Octal OEM code for '1/2'
```

CString::Collate

int Collate(const char* *psz* **) const;**

psz The other string used for comparison.

Remarks

Performs a locale-specific comparison of two strings; uses the run-time function **strcoll**. **Compare** performs a faster, ASCII-only comparison. A **CString** object can be used as the argument because the class provides the appropriate conversion operator.

Return Value

The function returns 0 if the strings are identical, –1 if this **CString** object is less than *psz*, or 1 if this **CString** object is greater than *psz*.

See Also

CString::Compare, **CString::CompareNoCase**

Example

```
CString s1( "abc" );
CString s2( "abd" );
ASSERT( s1.Collate( s2 ) == -1 );
```

CString::Compare

int Compare(const char* *psz* **) const;**

psz The other string used for comparison.

Remarks

Compares this **CString** object with another string, character by character; uses the run-time function **strcmp**. If you need a language-specific comparison, use the **Collate** member function.

Return Value

The function returns 0 if the strings are identical, –1 if this **CString** object is less than *psz*, or 1 if this **CString** object is greater than *psz*.

See Also

CString::CompareNoCase, **CString::Collate**

Example

```
CString s1( "abc" );
CString s2( "abd" );
ASSERT( s1.Compare( s2 ) == -1 ); // Compare with another CString.
ASSERT( s1.Compare( "abe" ) == -1 ); // Compare with a char * string.
```

CString::CompareNoCase

int CompareNoCase(const char* *psz* **) const;**

psz The other string used for comparison.

Remarks

Compares this **CString** object with another string, character by character; uses the run-time function **stricmp**. The algorithm for deciding case applies only to ASCII characters: 'A' == 'a' -> 'Z' == 'z'. If you need a language-specific comparison, use the **Collate** member function.

Return Value

The function returns 0 if the strings are identical (ignoring case), –1 if this **CString** object is less than *psz* (ignoring case), or 1 if this **CString** object is greater than *psz* (ignoring case).

See Also

CString::Compare, **CString::Collate**

Example

```
CString s1( "abc" );
CString s2( "ABD" );
ASSERT( s1.CompareNoCase( s2 ) == -1 ); // Compare with a CString.
ASSERT( s1.Compare( "ABE" ) == -1 ); // Compare with a char * string.
```

CString::CString

CString( );

CString(const CString& *stringSrc* **)**
throw(CMemoryException);

CString(const char* *psz* **)**
throw(CMemoryException);

CString(char *ch*, **int** *nRepeat* **= 1)**
throw(CMemoryException);

CString(const char* *pch*, **int** *nLength* **)**
throw(CMemoryException);

CString(const char FAR* *lpsz* **)**
throw(CMemoryException);

CString(const char FAR* *lpch*, **int** *nLength* **)**
throw(CMemoryException);

stringSrc An existing **CString** object to be copied into this **CString** object.

psz A null-terminated string to be copied into this **CString** object.

ch A single character to be repeated *nRepeat* times.

nRepeat The repeat count for *ch*.

pch A pointer to an array of characters of length *nLength*, not null-terminated.

nLength A count of the number of characters in *pch*.

lpsz A far pointer to a null-terminated ASCII string.

lpch A far pointer to an array of characters of length *nLength*.

Remarks Each of these constructors initializes a new **CString** object with the specified data. Because the constructors copy the input data into new allocated storage, you should be aware that memory exceptions may result. Note that some of these constructors act as conversion functions. This allows you to substitute, for example, a **char*** where a **CString** object is expected.

See Also **CString::operator =**, "CString Exception Cleanup," page 791

Example

```
CString s1;                       // Empty string
CString s2( "cat" );              // From a C string literal
CString s3 = s2;                  // Copy constructor
CString s4( s2 + " " + s3 );     // From a string expression

CString s5( 'x' );                // s5 = "x"
CString s6( 'x', 6 );             // s6 = "xxxxxx"

CString city = "Philadelphia"; // NOT the assignment operator
```

CString::~CString

~CString( );

Remarks Releases allocated memory used to store the string's character data.

CString::Empty

void Empty( );

Remarks Makes this **CString** object an empty string and frees memory as appropriate.

See Also **CString::IsEmpty**, "CString Exception Cleanup," page 791

Example

```
CString s1( "abc" );
CString s2;
s1.Empty();
ASSERT( s1 == s2 );
```

CString::Find

int Find(char *ch* **) const;**

int Find(const char* *pszSub* **) const;**

ch A single character to search for.

pszSub A substring to search for.

Remarks

Searches this string for the first match of a substring. The function is overloaded to accept both single characters (similar to the run-time function **strchr**) and strings (similar to **strstr**).

Return Value

The zero-based index of the first character in this **CString** object that matches the requested substring or characters; –1 if the substring or character is not found.

See Also

CString::ReverseFind, **CString::FindOneOf**

Example

```
CString s( "abcdef" );
ASSERT( s.Find( 'c' ) == 2 );
ASSERT( s.Find( "de" ) == 3 );
```

CString::FindOneOf

int FindOneOf(const char* *pszCharSet* **) const;**

pszCharSet String containing characters for matching.

Remarks

Searches this string for the first character that matches any character contained in *pszCharSet*.

Return Value

The zero-based index of the first character in this string that is also in *pszCharSet*; –1 if there is no match.

See Also

CString::Find

Example

```
CString s( "abcdef" );
ASSERT( s.FindOneOf( "xd" ) == 3 ); // 'd' is first match
```

CString::GetAt

char GetAt(int *nIndex* **) const;**

nIndex Zero-based index of the character in the CString object. The *nIndex* parameter must be greater than or equal to 0 and less than the value returned by GetLength. The Debug version of the Microsoft Foundation Class Library validates the bounds of *nIndex*; the Release version will not.

Remarks

You can think of a **CString** object as an array of characters. The **GetAt** member function returns a single character specified by an index number. The overloaded subscript ([ ]) operator is a convenient alias for **GetAt**.

Return Value A **char** containing the character at the specified position in the string.

See Also **CString::SetAt**, **CString::GetLength**, **CString::operator [ ]**

Example

```
CString s( "abcdef" );
ASSERT( s.GetAt(2) == 'c' );
```

CString::GetBuffer

char* GetBuffer(int *nMinBufLength* **)**
throw(CMemoryException);

nMinBufLength The minimum size of the CString character buffer in bytes. You do not need to allow space for a null terminator.

Remarks Returns a pointer to the internal character buffer for the **CString** object. The returned pointer to **char** is not **const** and thus allows direct modification of **CString** contents.

If you use the pointer returned by **GetBuffer** to change the string contents, you must call **ReleaseBuffer** before using any other **CString** member functions. The address returned by **GetBuffer** is invalid after the call to **ReleaseBuffer** or any other **CString** operation. The buffer memory will be freed automatically when the **CString** object is destroyed. Note that if you keep track of the string length yourself, you need not append the terminating null byte. You must, however, specify the final string length when you release the buffer with **ReleaseBuffer**, or you can pass –1 for the length and **ReleaseBuffer** will perform a **strlen** on the buffer to determine its length.

Return Value A **char** pointer to the object's (usually null-terminated) ASCII character buffer.

See Also **CString::GetBufferSetLength**, **CString::ReleaseBuffer**

Example

```
CString s;
    char* p = s.GetBuffer(10); // Allocate space for 10 characters.
    s = "abcdefg"; // p is still valid because length of s is 7
                   // characters.
    p[1] = 'B'; // Change 'b' to 'B'.
#ifdef _DEBUG
    afxDump << "char* p " << (void*) p << ":" << p << "\n";
#endif
    char* q = s.GetBuffer(12);  // Get a new, larger buffer.
    // q is a different address from p, but the string is the same.
```

```
#ifdef _DEBUG
    afxDump << "char* q " << (void*) q << ":" << q << "\n";
#endif
    s += "hij";  // String length is still smaller than 12.
#ifdef _DEBUG
    afxDump << "char* q " << (void*) q << ":" << q << "\n";
#endif
    s += "klmnop";  // Now it is larger than 12, so the characters
                    // are moved, and q is no longer valid.
#ifdef _DEBUG
    afxDump << "char* q " << (void*) q << ":" << q << "\n";
    afxDump << "CString s " << s << "\n"; // s contains
                               // "aBcdefghijklmnop".
#endif
    s.ReleaseBuffer();
```

CString::GetBufferSetLength

char* GetBufferSetLength(int *nNewLength* **)**
throw(CMemoryException);

nNewLength The exact size of the CString character buffer in bytes.

Remarks

Returns a pointer to the internal character buffer for the **CString** object, truncating or growing its length if necessary to exactly match the length specified in *nNewLength*. The returned pointer to **char** is not **const** and thus allows direct modification of **CString** contents.

If you use the pointer returned by **GetBuffer** to change the string contents, you must call **ReleaseBuffer** before using any other **CString** member functions. The address returned by **GetBuffer** is invalid after the call to **ReleaseBuffer** or any other **CString** operation. The buffer memory will be freed automatically when the **CString** object is destroyed.

Note that if you keep track of the string length yourself, you need not append the terminating null byte. You must, however, specify the final string length when you release the buffer with **ReleaseBuffer**, or you can pass –1 for the length and **ReleaseBuffer** will perform a **strlen** on the buffer to determine its length.

Return Value

A **char** pointer to the object's (usually null-terminated) ASCII character buffer.

See Also

CString::GetBuffer, **CString::ReleaseBuffer**

CString::GetLength

int GetLength( ) const;

Remarks

Returns a count of the characters in this **CString** object. The count does not include a null terminator.

See Also

CString::IsEmpty

Example

```
CString s( "abcdef" );
ASSERT( s.GetLength() == 6 );
```

CString::IsEmpty

BOOL IsEmpty( ) const;

Remarks

Tests a **CString** object for the empty condition.

Return Value

TRUE if the **CString** object has 0 length; otherwise **FALSE**.

See Also

CString::GetLength

Example

```
CString s;
ASSERT( s.IsEmpty() );
```

CString::Left

CString Left(int *nCount* **) const**
  throw(CMemoryException);

nCount The number of characters to extract from this CString object.

Remarks

Extracts the first (that is, leftmost) *nCount* characters from this **CString** object and returns a copy of the extracted substring. If *nCount* exceeds the string length, then the entire string is extracted. **Left** is similar to the Basic LEFT$ command (except that indexes are zero-based).

Return Value A **CString** object containing a copy of the specified range of characters. Note that the returned **CString** object may be empty.

See Also **CString::Mid**, **CString::Right**

Example

```
CString s( "abcdef" );
ASSERT( s.Left(3) == "abc" );
```

CString::LoadString

BOOL LoadString(UINT *nID* **)**
throw(CMemoryException);

nID A Windows string resource ID.

Remarks Reads a Windows string resource, identified by *nID*, into an existing **CString** object. The maximum string size is 255 characters. This function is declared in AFX.H only if **_WINDOWS** is defined. Its use requires the Windows-compiled version of the Microsoft Foundation classes, and it is normally used with AFXWIN.H.

Return Value **TRUE** if resource load was successful; otherwise **FALSE**.

Example

```
#define IDS_FILENOTFOUND 1
CString s;
s.LoadString( IDS_FILENOTFOUND );
```

CString::MakeLower

void MakeLower( );

Remarks Converts this **CString** object to a lowercase string.

See Also **CString::MakeUpper**

Example

```
CString s( "ABC" );
s.MakeLower();
ASSERT( s == "abc" );
```

CString::MakeReverse

void MakeReverse( );

Remarks Reverses the order of the characters in this **CString** object.

Example

```
CString s( "abc" );
s.MakeReverse();
ASSERT( s == "cba" );
```

CString::MakeUpper

void MakeUpper( );

Remarks Converts this **CString** object to an uppercase string.

See Also **CString::MakeLower**

Example

```
CString s( "abc" );
s.MakeUpper();
ASSERT( s == "ABC" );
```

CString::Mid

CString Mid(int *nFirst* **) const**
throw(CMemoryException);

CString Mid(int *nFirst*, **int** *nCount* **) const**
throw(CMemoryException);

nFirst The zero-based index of the first character in this **CString** object that is to be included in the extracted substring.

nCount The number of characters to extract from this **CString** object. If this parameter is not supplied, then the remainder of the string is extracted.

Remarks Extracts a substring of length *nCount* characters from this **CString** object, starting at position *nFirst* (zero-based). The function returns a copy of the extracted substring. **Mid** is similar to the Basic MID$ command (except that indexes are zero-based).

Return Value A **CString** object that contains a copy of the specified range of characters. Note that the returned **CString** object may be empty.

See Also **CString::Left**, **CString::Right**

Example

```
CString s( "abcdef" );
ASSERT( s.Mid( 2, 3 ) == "cde" );
```

CString::OemToAnsi

void OemToAnsi( );

Remarks Converts all the characters in this **CString** object from the OEM character set to the ANSI character set. See the IBM PC Extended Character Set table and the ANSI table in the *Microsoft Windows Programmer's Reference*. This function is available only in the Windows-compiled library of the Microsoft Foundation classes and is declared in AFX.H only if **_WINDOWS** is defined.

See Also **CString::AnsiToOem**

Example

```
CString s( '\253' );  // Octal OEM code for '1/2'
s.OemToAnsi();
ASSERT( s == "\265" ); // Octal ANSI code for '1/2'
```

CString::ReleaseBuffer

void ReleaseBuffer(int *nNewLength* = **–1);**

nNewLength The new length of the string in characters, not counting a null terminator. If the string is null-terminated, the –1 default value sets the CString size to the current length of the string.

Remarks Use **ReleaseBuffer** to end use of a buffer allocated by **GetBuffer**. If you know that the string in the buffer is null-terminated, you can omit the *nNewLength* argument. If your string is not null-terminated, then use *nNewLength* to specify its length. The address returned by **GetBuffer** is invalid after the call to **ReleaseBuffer** or any other **CString** operation.

See Also **CString::GetBuffer**

Example

```
CString s;
char* p = s.GetBuffer( 1024 );
s = "abc";
ASSERT( s.GetLength() == 3 ); // String length = 3
s.ReleaseBuffer(); // Surplus memory released, p is now invalid.
ASSERT( s.GetLength() == 3 ); // Length still 3
```

CString::ReverseFind

int ReverseFind(char *ch* **) const;**

ch The character to search for.

Remarks Searches this **CString** object for the last match of a substring. The function is similar to the run-time function **strrchr**.

Return Value The index of the last character in this **CString** object that matches the requested character; –1 if the character is not found.

See Also **CString::Find**, **CString::FindOneOf**

Example

```
CString s( "abcabc" );
ASSERT( s.ReverseFind( 'b' ) == 4 );
```

CString::Right

CString Right(int *nCount* **) const**
throw(CMemoryException);

nCount The number of characters to extract from this CString object.

Remarks Extracts the last (that is, rightmost) *nCount* characters from this **CString** object and returns a copy of the extracted substring. If *nCount* exceeds the string length, then the entire string is extracted. **Right** is similar to the Basic RIGHT$ command (except that indexes are zero-based).

Return Value A **CString** object that contains a copy of the specified range of characters. Note that the returned **CString** object may be empty.

See Also **CString::Mid**, **CString::Left**

Example

```
CString s( "abcdef" );
ASSERT( s.Right(3) == "def" );
```

CString::SetAt

void SetAt(int *nIndex*, **char** *ch* **);**

nIndex Zero-based index of the character in the CString object. The *nIndex* parameter must be greater than or equal to 0 and less than the value returned by GetLength. The Debug version of the Microsoft Foundation Class Library will validate the bounds of *nIndex*; the Release version will not.

ch The character to insert. Must not be '\0'.

Remarks You can think of a **CString** object as an array of characters. The **SetAt** member function overwrites a single character specified by an index number. **SetAt** will not enlarge the string if the index exceeds the bounds of the existing string.

See Also **CString::GetAt**, **CString::operator [ ]**

CString::SpanExcluding

CString SpanExcluding(const char* *pszCharSet* **) const throw(CMemoryException);**

pszCharSet A string interpreted as a set of characters.

Remarks Extracts the largest substring that excludes only the characters in the specified set *pszCharSet*; starts from the first character in this **CString** object. If the first character of the string is included in the character set, then **SpanExcluding** returns an empty string.

Return Value A copy of the substring that contains only characters not in *pszCharSet*.

See Also **CString::SpanIncluding**

CString::SpanIncluding

CString SpanIncluding(const char* *pszCharSet* **) const**
throw(CMemoryException);

pszCharSet A string interpreted as a set of characters.

Remarks

Extracts the largest substring that contains only the characters in the specified set *pszCharSet*; starts from the first character in this **CString** object. If the first character of the string is not in the character set, then **SpanIncluding** returns an empty string.

Return Value

A copy of the substring that contains only characters in *pszCharSet*.

See Also

CString::SpanExcluding

Operators

CString::operator =

const CString& operator =(const CString& *stringSrc* **)**
throw(CMemoryException);

const CString& operator =(const char* *psz* **)**
throw(CMemoryException);

const CString& operator =(char *ch* **)**
throw(CMemoryException);

Remarks

The **CString** assignment (=) operator reinitializes an existing **CString** object with new data. If the destination string (that is, the left side) is already large enough to store the new data, no new memory allocation is performed. You should be aware that memory exceptions may occur whenever you use the assignment operator because new storage is often allocated to hold the resulting **CString** object.

See Also **CString::CString**

Example

```
CString s1, s2;            // Empty CString objects

s1 = "cat";                // s1 = "cat"
s2 = s1;                   // s1 and s2 each = "cat"
s1 = "the " + s1;          // Or expressions
s1 = 'x';                  // Or just individual characters
```

CString::operator const char* ()

operator const char* () const;

Remarks This useful casting operator provides an efficient method to access the null-terminated C string contained in a **CString** object. No characters are copied; only a pointer is returned. Be careful with this operator. If you change a **CString** object after you have obtained the character pointer, you may cause a reallocation of memory that invalidates the pointer.

Return Value A character pointer if the cast was successful; otherwise a null pointer.

CString::operator <<, >>

friend CArchive& operator <<(CArchive& *ar*, **const CString&** *string* **)**
throw(CArchiveException);

friend CArchive& operator >>(CArchive& *ar*, **CString&** *string* **)**
throw(CArchiveException);

friend CDumpContext& operator <<(CDumpContext& *dc*,
const CString& *string* **);**

Remarks The **CString** insertion (<<) operator supports diagnostic dumping and storing to an archive. The extraction (>>) operator supports loading from an archive.

The **CDumpContext** operators are valid only in the Debug version of the Microsoft Foundation Class Library.

Example

```
// Operator <<, >> example
    extern CArchive ar;
    CString s( "abc" );
#ifdef _DEBUG
    afxDump << s;  // Prints the value (abc)
    afxDump << &s;  // Prints the address
#endif

    if( ar.IsLoading() )
        ar >> s;
    else
        ar << s;
```

CString::operator +

friend CString operator +(const CString& *string1*, **const CString&** *string2* **)**
throw(CMemoryException);

friend CString operator +(const CString& *string*, **char** *ch* **)**
throw(CMemoryException);

friend CString operator +(char *ch*, **const CString&** *string* **)**
throw(CMemoryException);

friend CString operator +(const CString& *string*, **const char*** *psz* **)**
throw(CMemoryException);

friend CString operator +(const char* *psz*, **const CString&** *string* **)**
throw(CMemoryException);

Remarks

The + concatenation operator joins two strings and returns a **CString** object. One of the two argument strings must be a **CString** object. The other can be a character pointer or a character. You should be aware that memory exceptions may occur whenever you use the concatenation operator since new storage may be allocated to hold temporary data. You must ensure that the maximum length limit is not exceeded. The Debug version of the Microsoft Foundation Class Library asserts when it detects strings that are too long.

Return Value

A **CString** object that is the temporary result of the concatenation. This return value makes it possible to combine several concatenations in the same expression.

See Also

CString::operator +=

Example

```
CString s1( "abc" );
    CString s2( "def" );
    ASSERT( (s1 + s2 ) == "abcdef" );
    CString s3;
    s3 = CString( "abc" ) + "def" ; // Correct
// s3 = "abc" + "def"; // Wrong! One of the arguments must be a CString.
```

CString::operator +=

const CString& operator +=(const CString& *string* **)**
throw(CMemoryException);

const CString& operator +=(char *ch* **)**
throw(CMemoryException);

const CString& operator +=(const char* *psz* **)**
throw(CMemoryException);

Remarks

The += concatenation operator joins characters to the end of this string. The operator accepts another **CString** object, a character pointer, or a single character. You should be aware that memory exceptions may occur whenever you use this concatenation operator because new storage may be allocated for characters added to this **CString** object. You must ensure that the maximum length limit is not exceeded. The Debug version of the Microsoft Foundation Class Library asserts when it detects strings that are too long.

See Also

CString::operator +

Example

```
CString s( "abc" );
ASSERT( ( s += "def" ) == "abcdef" );
```

CString Comparison Operators

BOOL operator ==(const CString& *s1*, **const CString&** *s2* **);**

BOOL operator ==(const CString& *s1*, **const char*** *s2* **);**

BOOL operator ==(const char* *s1*, **const CString&** *s2* **);**

BOOL operator !=(const CString& *s1*, **const CString&** *s2* **);**

BOOL operator !=(const CString& *s1*, **const char*** *s2* **);**

BOOL operator !=(const char* *s1*, **const CString&** *s2* **);**

BOOL operator <(const CString& *s1*, **const CString&** *s2* **);**

BOOL operator <(const CString& *s1*, **const char*** *s2* **);**

BOOL operator <(const char* *s1*, **const CString&** *s2* **);**

BOOL operator >(const CString& *s1*, **const CString&** *s2* **);**

BOOL operator >(const CString& *s1*, **const char*** *s2* **);**

BOOL operator >(const char* *s1*, **const CString&** *s2* **);**

BOOL operator <=(const CString& *s1*, **const CString&** *s2* **);**

BOOL operator <=(const CString& *s1*, **const char*** *s2* **);**

BOOL operator <=(const char* *s1*, **const CString&** *s2* **);**

BOOL operator >=(const CString& *s1*, **const CString&** *s2* **);**

BOOL operator >=(const CString& *s1*, **const char*** *s2* **);**

BOOL operator >=(const char* *s1*, **const CString&** *s2* **);**

Remarks

These comparison operators compare two **CString** objects, and they compare a **CString** object with an ordinary null-terminated C string. The operators are a convenient substitute for the case-sensitive **Compare** member function.

Return Value

TRUE if the strings meet the comparison condition; otherwise **FALSE**.

Example

```
CString s1( "abc" );
CString s2( "abd" );
ASSERT( s1 < s2 ); // Operator is overloaded for both.
ASSERT( "ABC" < s1 ); // CString and char*
ASSERT( s2 > "abe" );
```

CString::operator [ ]

char operator [ ](int *nIndex* **) const;**

Remarks

You can think of a **CString** object as an array of characters. The overload subscript ([ ]) operator returns a single character specified by the zero-based index in *nIndex*. This operator is a convenient substitute for the **GetAt** member function. You can use the subscript ([ ]) operator on the right side of an expression (r-value semantics), but you cannot use it on the left side of an expression (l-value semantics). That is, you can use this operator to get characters in a **CString**, but you cannot use it to set characters in the **CString**.

See Also **CString::GetAt**, **CString::SetAt**

Example

```
CString s( "abc" );
ASSERT( s[1] == 'b' );
```

Application Notes

CString Exception Cleanup

Memory Leaks

If you notice that the Microsoft Foundation Class Library diagnostic memory allocator is reporting leaks for non-**CObject** memory blocks, check your exception-processing logic to see if **CString** objects are being cleaned up properly. The **CString** class is typical in that its constructor and member functions allocate memory that must be freed by the destructor. **CString** is unique, however, in that instances are often allocated on the frame rather than on the heap. When a frame-allocated **CString** object goes out of scope, its destructor is called invisibly without need for a **delete** statement. Whether you explicitly destroy an object or not, you must be sure that the destructor call is not bypassed by uncaught exceptions. For frame-allocated (and heap-allocated) **CString** objects, use a **CATCH** statement to channel execution through the end of the function that contains the **CString** allocation.

Example This is an example of incorrect programming.

```
void TestFunction1()
{
    CString s1 = "test";
    OtherFunction();   // OtherFunction may raise an exception.
       // This point not passed if an exception occurred.
       // s1's destructor called here (frees character storage for
       // "test")
}
```

You must add **TRY/CATCH** code to free the string character data in response to memory exceptions.

Now the program has been improved to properly handle exceptions.

```
void TestFunction2()
{
    CString s1;
    TRY
    {
        s1 = "test";
        OtherFunction(); // OtherFunction may raise an exception.
    }
    CATCH( CException, e )
    {
        s1.Empty();                // Frees up associated data
        THROW_LAST()
    }
    END_CATCH
}
```

CString Argument Passing

Argument-Passing Conventions

When you define a class interface, you must determine the argument-passing convention for your member functions. There are some standard rules for passing and returning **CString** objects. If you follow these rules, you will have efficient, correct code.

Strings as Function Inputs

If a string is an input to a function, in most cases it is best to declare the string function parameter as **const char***. Convert to a **CString** object as necessary within the function using constructors and assignment operators. If the string contents are to be changed by a function, declare the parameter as a nonconstant **CString** reference (**CString&**).

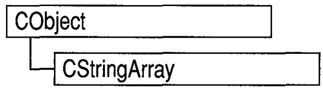
Strings as Function Outputs

Normally you can return **CString** objects from functions since **CString** objects follow value semantics like primitive types. To return a read-only string, use a constant **CString** reference (**const CString&**).

Example

```
class CName : public CObject
{
private:
    CString m_firstName;
    char m_middleInit;
    CString m_lastName;
public:
    CName() {}
    void SetData( const char* fn, const char mi, const char* ln )
    {
        m_firstName = fn;
        m_middleInit = mi;
        m_lastName = ln;
    }
    void GetData( CString& cfn, char mi, CString& cln )
    {
        cfn = m_firstName;
        mi = m_middleInit;
        cln = m_lastName;
    }
    CString GetLastName()
    {
        return m_lastName;
    }
};
    CName name;
    CString last, first;
    char middle;
    name.SetData( "John", 'Q', "Public" );
    ASSERT( name.GetLastName() == "Public" );
    name.GetData( first, middle, last );
    ASSERT( ( first == "John" ) && ( last == "Public" ) );
}
return 0;
}
```

class CStringArray : public CObject



The **CStringArray** class supports arrays of **CString** objects. The member functions of **CStringArray** are similar to the member functions of class **CObArray**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value, substitute a **CString**. Wherever you see a **CObject** pointer as a function parameter, substitute a **const** pointer to **char**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
CString CStringArray::GetAt( int <nIndex> ) const;
```

and

```
void SetAt( int <nIndex>, CObject* <newElement> )
```

translates to

```
void SetAt( int <nIndex>, const char* <newElement> )
```

CStringArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of **CString** objects is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each element is, in turn, serialized. If you need a dump of individual string elements in the array, you must set the depth of the dump context to 1 or greater. When a **CString** array is deleted, or when its elements are removed, string memory is freed as appropriate.

#include <afxcoll.h>

Construction/Destruction—Public Members

CStringArray	Constructs an empty array for **CString** objects.
~CStringArray	Destroys a **CStringArray** object.

Bounds—Public Members

GetSize	Gets number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations—Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; array not allowed to grow.
ElementAt	Returns a temporary reference to the element pointer within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

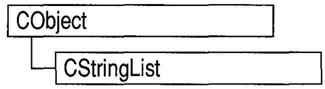
Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

class CStringList : public CObject



The **CStringList** class supports lists of **CString** objects. All comparisons are done by value, meaning that the characters in the string are compared instead of the addresses of the strings. The member functions of **CStringList** are similar to the member functions of class **CObList**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a return value, substitute a **CString**. Wherever you see a **CObject** pointer as a function parameter, substitute a **const** pointer to **char**.

```
CObject*& CObList::GetHead() const;
```

for example, translates to

```
CString& CStringList::GetHead() const;
```

and

```
POSITION AddHead( CObject* <newElement> );
```

translates to

```
POSITION AddHead( const char* <newElement> );
```

CStringList incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If a list of **CString** objects is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each **CString** element is, in turn, serialized.

If you need a dump of individual **CString** elements, you must set the depth of the dump context to 1 or greater. When a **CStringList** object is deleted, or when its elements are removed, the **CString** objects are deleted as appropriate.

#include <afxcoll.h>

Construction/Destruction—Public Members

CStringList	Constructs an empty list for **CString** objects.

Head/Tail Access—Public Members

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations — Public Members

RemoveHead	Removes the element from the head of the list.
RemoveTail	Removes the element from the tail of the list.
AddHead	Adds an element (or all the elements in another list) to the head of the list (makes a new head).
AddTail	Adds an element (or all the elements in another list) to the tail of the list (makes a new tail).
RemoveAll	Removes all the elements from this list.

Iteration — Public Members

GetHeadPosition	Returns the position of the head element of the list.
GetTailPosition	Returns the position of the tail element of the list.
GetNext	Gets the next element for iterating.
GetPrev	Gets the previous element for iterating.

Retrieval/Modification — Public Members

GetAt	Gets the element at a given position.
SetAt	Sets the element at a given position.
RemoveAt	Removes an element from this list as specified by position.

Insertion — Public Members

InsertBefore	Inserts a new element before a given position.
InsertAfter	Inserts a new element after a given position.

Searching — Public Members

Find	Gets the position of an element specified by string value.
FindIndex	Gets the position of an element specified by a zero-based index.

Status — Public Members

GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

class CTime

A **CTime** object represents an absolute time and date. The **CTime** class incorporates the ANSI **time_t** data type and its associated run-time functions, including the ability to convert to and from a Gregorian date and 24-hour time. **CTime** values are based on universal coordinated time (UCT), which is equivalent to Greenwich mean time (GMT). The local time zone is controlled by the **TZ** environment variable. For more information on the **time_t** data type and the run-time functions that are used by **CTime**, see the *Run-Time Library Reference*. Note that **CTime** was the **strftime** function, which is not supported for Windows dynamic-link libraries (DLL). Therefore, **CTime** cannot be used in Windows DLLs. A companion class, **CTimeSpan**, represents a time interval—the difference between two **CTime** objects.

The **CTime** and **CTimeSpan** classes are not designed for derivation. Because there are no virtual functions, the size of **CTime** and **CTimeSpan** objects is exactly 4 bytes. Most member functions are inline.

#include <afx.h>

See Also Run-time functions: **asctime**, **_ftime**, **gmtime**, **localtime**, **strftime**, **time**

Construction/Destruction—Public Members

CTime	Constructs **CTime** objects in various ways.
GetCurrentTime	Creates a **CTime** object that represents the current time (static member function).

Extraction—Public Members

GetTime	Returns a **time_t** that corresponds to this **CTime** object.
GetYear	Returns the year that this **CTime** object represents.
GetMonth	Returns the month that this **CTime** object represents (1 through 12).
GetDay	Returns the day that this **CTime** object represents (1 through 31).
GetHour	Returns the hour that this **CTime** object represents (0 through 23).
GetMinute	Returns the minute that this **CTime** object represents (0 through 59).

GetSecond	Returns the second that this **CTime** object represents (0 through 59).
GetDayOfWeek	Returns the day of the week (1 for Sunday, 2 for Monday, and so forth).

Conversion—Public Members

GetGmtTm	Breaks down a **CTime** object into components—based on UCT.
GetLocalTm	Breaks down a **CTime** object into components—based on the local time zone.
Format	Converts a **CTime** object into a formatted string—based on the local time zone.
FormatGmt	Converts a **CTime** object into a formatted string—based on UCT.

Operators—Public Members

operator =	Assigns new time values.
operator +, –	Add and subtract **CTimeSpan** and **CTime** objects.
operator +=, –=	Add and subtract a **CTimeSpan** object to and from this **CTime** object.
operator ==, < , etc.	Compare two absolute times.

Archive/Dump—Public Members

operator <<	Outputs a **CTime** object to **CArchive** or **CDumpContext**.
operator >>	Inputs a **CTime** object from **CArchive**.

Member Functions

CTime::CTime

CTime( );

CTime(const CTime& *timeSrc* **);**

CTime(time_t *time* **);**

CTime(int *nYear*, **int** *nMonth*, **int** *nDay*, **int** *nHour*, **int** *nMin*, **int** *nSec* **);**

CTime(WORD *wDosDate*, **WORD** *wDosTime* **);**

timeSrc Indicates a **CTime** object that already exists.

time Indicates a time value.

nYear, *nMonth*, *nDay*, *nHour*, *nMin*, *nSec* Indicate year, month, day, hour, minute, and second.

wDosDate, *wDosTime* Indicate the date and time obtained through the MS-DOS functions **_dos_getftime** and **_dos_getdate**.

Remarks

All these constructors create a new **CTime** object initialized with the specified absolute time, based on the current time zone. Each constructor is described below:

- **CTime( );** Constructs a **CTime** object with a 0 (illegal) value. Note that 0 is an invalid time. This constructor allows you to define **CTime** object arrays. You should initialize such arrays with valid times prior to use.
- **CTime(const CTime&);** Constructs a **CTime** object from another **CTime** value.
- **CTime(time_t);** Constructs a **CTime** object from a **time_t** type.
- **CTime(int, int,** etc.**);** Constructs a **CTime** object from local time components with each component constrained to the following ranges:

Component	Range
nYear	1970–2038
nMonth	1–12
nDay	1–31
nHour	0–23
nMin	0–59
nSec	0–59

This constructor makes the appropriate conversion to UCT. The Debug version of the Microsoft Foundation Class Library asserts if one or more of the time-day components is out of range. It is your responsibility to validate the arguments prior to calling.

Example

```
time_t osBinaryTime;  // C run-time time (defined in <time.h>)
time( &osBinaryTime ) ;  // Get the current time from the
                         // operating system.
CTime time1; // Empty CTime. (0 is illegal time value.)
CTime time2 = time1; // Copy constructor.
CTime time3( osBinaryTime );  // CTime from C run-time time
CTime time4( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
```

CTime::Format

CString Format(const char* *pFormat* **);**

pFormat Specifies a formatting string similar to the **printf** formatting string. See the run-time function **strftime** for details.

Remarks Generates a formatted string that corresponds to this **CTime** object. The time value is converted to local time.

Return Value A **CString** that contains the formatted time.

See Also **CTime::FormatGmt**

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
CString s = t.Format( "%A, %B %d, %Y" );
ASSERT( s == "Friday, March 19, 1999" );
```

CTime::FormatGmt

CString FormatGmt(const char* *pFormat* **);**

pFormat Specifies a formatting string similar to the **printf** formatting string. See the run-time function **strftime** for details.

Remarks Generates a formatted string that corresponds to this **CTime** object. The time value is not converted and thus reflects UCT.

Return Value A **CString** that contains the formatted time.

See Also **CTime::Format**

Example See the example for **Format**.

CTime::GetCurrentTime

static CTime PASCAL GetCurrentTime( );

Remarks Returns a **CTime** object that represents the current time.

Example

```
CTime t = CTime::GetCurrentTime();
```

CTime::GetDay

int GetDay( ) const;

Remarks Returns the day of the month, based on local time, in the range 1 through 31.

See Also **CTime::GetDayOfWeek**

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
ASSERT( t.GetDay() == 19 );
ASSERT( t.GetMonth() == 3 );
ASSERT( t.GetYear() == 1999 );
```

CTime::GetDayOfWeek

int GetDayOfWeek( ) const;

Remarks Returns the day of the week based on local time; 1 = Sunday, 2 = Monday, ..., 7 = Saturday.

CTime::GetGmtTm

struct tm* GetGmtTm(struct tm* *ptm* = **NULL) const;**

ptm Points to a buffer that will receive the time data. If this pointer is **NULL**, an internal, statically allocated buffer is used. The data in this default buffer is overwritten as a result of calls to other **CTime** member functions.

Remarks Gets a **struct tm** that contains a decomposition of the time contained in this **CTime** object. **GetGmtTm** returns UCT.

Return Value A pointer to a filled-in **struct tm** as defined in the include file TIME.H. The members and the values they store are as follows:

- **tm_sec** Seconds
- **tm_min** Minutes
- **tm_hour** Hours (0–23)
- **tm_mday** Day of month (1–31)
- **tm_mon** Month (0–11; January = 0)

- **tm_year** Year (actual year minus 1900)
- **tm_wday** Day of week (1–7; Sunday = 1)
- **tm_yday** Day of year (0–365; January 1 = 0)
- **tm_isdst** Always 0

Note The year in **struct tm** is in the range 70 to 138; the year in the **CTime** interface is in the range 1970 to 2038 (inclusive).

Example See the example for **GetLocalTm**.

CTime::GetHour

int GetHour( ) const;

Remarks Returns the hour, based on local time, in the range 0 through 23.

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
ASSERT( t.GetSecond() == 0 );
ASSERT( t.GetMinute() == 15 );
ASSERT( t.GetHour() == 22 );
```

CTime::GetLocalTm

struct tm* GetLocalTm(struct tm* *ptm* **= NULL) const;**

ptm Points to a buffer that will receive the time data. If this pointer is **NULL**, an internal, statically allocated buffer is used. The data in this default buffer is overwritten as a result of calls to other **CTime** member functions.

Remarks Gets a **struct tm** containing a decomposition of the time contained in this **CTime** object. **GetLocalTm** returns local time.

Return Value A pointer to a filled-in **struct tm** as defined in the include file TIME.H. See **GetGmtTm** for the structure layout.

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
struct tm* osTime;  // A pointer to a structure containing time
                    // elements.
osTime = t.GetLocalTm( NULL );
ASSERT( osTime->tm_mon == 2 ); // Note zero-based month!
```

CTime::GetMinute

int GetMinute( ) const;

Remarks Returns the minute, based on local time, in the range 0 through 59.

Example See the example for **GetHour**.

CTime::GetMonth

int GetMonth( ) const;

Remarks Returns the month, based on local time, in the range 1 through 12 (1 = January).

Example See the example for **GetDay**.

CTime::GetSecond

int GetSecond( ) const;

Remarks Returns the second, based on local time, in the range 0 through 59.

Example See the example for **GetHour**.

CTime::GetTime

time_t GetTime( ) const;

Remarks Returns a **time_t** value for the given **CTime** object.

See Also **CTime::CTime**

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
time_t osBinaryTime = t.GetTime(); // time_t defined in <time.h>
printf( "time_t = %ld\n", osBinaryTime );
```

CTime::GetYear

int GetYear( ) const;

Remarks
Returns the year, based on local time, in the range 1970 to 2038.

Example
See the example for **GetDay**.

Operators

CTime::operator =

const CTime& operator =(const CTime& *timeSrc* **);**

const CTime& operator =(time_t *t* **);**

Remarks
These overloaded assignment operators copy the source time into this **CTime** object. The internal time storage in a **CTime** object is independent of time zone. Time-zone conversion is not necessary during assignment.

See Also
CTime::CTime

Example

```
time_t osBinaryTime;  // C run-time time (defined in <time.h>)
CTime t1 = osBinaryTime; // Assignment from time_t
CTime t2 = t1; // Assignment from CTime
```

CTime::operator +, –

CTime operator +(CTimeSpan *timeSpan* **) const;**

CTime operator –(CTimeSpan *timeSpan* **) const;**

CTimeSpan operator –(CTime *time* **) const;**

Remarks
CTime objects represent absolute time. **CTimeSpan** objects represent relative time. The first two operators allow you to add and subtract **CTimeSpan** objects to and from **CTime** objects. The third allows you to subtract one **CTime** object from another to yield a **CTimeSpan** object.

Example

```
CTime t1( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
CTime t2( 1999, 3, 20, 22, 15, 0 ); // 10:15PM March 20, 1999
CTimeSpan ts = t2 - t1;  // Subtract 2 CTimes
ASSERT( ts.GetTotalSeconds() == 86400L );
ASSERT( ( t1 + ts ) == t2 );  // Add a CTimeSpan to a CTime.
ASSERT( ( t2 - ts ) == t1 );  // Subtract a CTimeSpan from a CTime.
```

CTime::operator +=, –=

const CTime& operator +=(CTimeSpan *timeSpan* **);**

const CTime& operator –=(CTimeSpan *timeSpan* **);**

Remarks

These operators allow you to add and subtract a **CTimeSpan** object to and from this **CTime** object.

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
t += CTimeSpan( 0, 1, 0, 0 ); // 1 hour exactly
ASSERT( t.GetHour() == 23 );
```

CTime Comparison Operators

BOOL operator ==(CTime *time* **) const;**

BOOL operator !=(CTime *time* **) const;**

BOOL operator <(CTime *time* **) const;**

BOOL operator >(CTime *time* **) const;**

BOOL operator <=(CTime *time* **) const;**

BOOL operator >=(CTime *time* **) const;**

Remarks

These operators compare two absolute times and return **TRUE** if the condition is true; otherwise **FALSE**.

Example

```
CTime t1 = CTime::GetCurrentTime();
CTime t2 = t1 + CTimeSpan( 0, 1, 0, 0 );    // 1 hour later
ASSERT( t1 != t2 );
ASSERT( t1 < t2 );
ASSERT( t1 <= t2 );
```

CTime::operators <<, >>

friend CDumpContext& operator <<(CDumpContext& *dc*, **CTime** *time* **);**

friend CArchive& operator <<(CArchive& *ar*, **CTime** *time* **);**

friend CArchive& operator >>(CArchive& *ar*, **CTime&** *rtime* **);**

Remarks

The **CTime** insertion (<<) operator supports diagnostic dumping and storing to an archive. The extraction (>>) operator supports loading from an archive.

When you send a **CTime** object to the dump context, the local time is displayed in readable date-time format.

See Also

CArchive, **CDumpContext**

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
afxDump << t << "\n"; // Prints 'CTime("Fri Mar 19 22:15:00 1999")'.

extern CArchive ar;
if( ar.IsLoading() )
  ar >> t;
else
  ar << t;
```

class CTimeSpan

A **CTimeSpan** object represents a relative time span. The **CTimeSpan** class incorporates the ANSI **time_t** data type and its associated run-time functions. These functions convert seconds to various combinations of days, hours, minutes, and seconds. A **CTimeSpan** object keeps time in seconds. Because the **CTimeSpan** object is stored as a signed number in 4 bytes, the maximum allowed span is approximately ± 68 years.

A companion class, **CTime**, represents an absolute time. A **CTimeSpan** is the difference between two **CTime** values. The **CTime** and **CTimeSpan** classes are not designed for derivation. Because there are no virtual functions, the size of both **CTime** and **CTimeSpan** objects is exactly 4 bytes. Most member functions are inline.

#include <afx.h>

See Also Run-time functions: **asctime**, **_ftime**, **gmtime**, **localtime**, **strftime**, **time**

Construction/Destruction—Public Members

CTimeSpan	Constructs **CTimeSpan** objects in various ways.

Extraction—Public Members

GetDays	Returns the number of complete days in this **CTimeSpan**.
GetHours	Returns the number of hours in the current day (–23 through 23).
GetTotalHours	Returns the total number of complete hours in this **CTimeSpan**.
GetMinutes	Returns the number of minutes in the current hour (–59 through 59).
GetTotalMinutes	Returns the total number of complete minutes in this **CTimeSpan**.
GetSeconds	Returns the number of seconds in the current minute (–59 through 59).
GetTotalSeconds	Returns the total number of complete seconds in this **CTimeSpan**.

Conversion—Public Members

Format	Converts a **CTimeSpan** into a formatted string.

Operators—Public Members

operator =	Assigns new time-span values.
operator +, –	Add and subtract **CTimeSpan** objects.
operator +=, –=	Add and subtract a **CTimeSpan** object to and from this **CTimeSpan**.
operator ==, <, etc.	Compare two relative time values.

Archive/Dump—Public Members

operator <<	Outputs a **CTimeSpan** object to **CArchive** or **CDumpContext**.
operator >>	Inputs a **CTimeSpan** object from **CArchive**.

Member Functions

CTimeSpan::CTimeSpan

CTimeSpan( );

CTimeSpan(const CTimeSpan& *timeSpanSrc* **);**

CTimeSpan(time_t *time* **);**

CTimeSpan(LONG *lDays*, **int** *nHours*, **int** *nMins*, **int** *nSecs* **);**

timeSpanSrc A **CTimeSpan** object that already exists.

time A **time_t** time value.

lDays, *nHours*, *nMins*, *nSecs* Days, hours, minutes, and seconds, respectively.

Remarks

All these constructors create a new **CTimeSpan** object initialized with the specified relative time. Each constructor is described below:

- **CTimeSpan( );** Constructs an uninitialized **CTimeSpan** object.
- **CTimeSpan(const CTimeSpan&);** Constructs a **CTimeSpan** object from another **CTimeSpan** value.

- **CTimeSpan(time_t);** Constructs a **CTimeSpan** object from a **time_t** type. This value should be the difference between two absolute **time_t** values.
- **CTimeSpan(LONG, int, int, int);** Constructs a **CTimeSpan** object from components with each component constrained to the following ranges:

Component	Range
lDays	0–25,000 (approximately)
nHours	0–23
nMins	0–59
nSecs	0–59

Note that the Debug version of the Microsoft Foundation Class Library asserts if one or more of the time-day components is out of range. It is your responsibility to validate the arguments prior to calling.

Example

```
CTimeSpan ts1;  // Uninitialized time value
CTimeSpan ts2a( ts1 ); // Copy constructor
CTimeSpan ts2b = ts1; // Copy constructor again
CTimeSpan ts3( 100 ); // 100 seconds
CTimeSpan ts4( 0, 1, 5, 12 );    // 1 hour, 5 minutes, and 12 seconds
```

CTimeSpan::Format

CString Format(const char* *pFormat* **);**

pFormat A formatting string similar to the **printf** formatting string. Formatting codes, preceded by a percent (%) sign, are replaced by the corresponding **CTimeSpan** component. Other characters in the formatting string are copied unchanged to the returned string. The value and meaning of the formatting codes for **Format** are listed below:

- **%D** Total days in this CTimeSpan
- **%H** Hours in the current day
- **%M** Minutes in the current hour
- **%S** Seconds in the current minute
- **%%** Percent sign

Remarks

Generates a formatted string that corresponds to this **CTimeSpan**. The Debug version of the library checks the formatting codes and asserts if the code is not in the table above.

Return Value A **CString** object that contains the formatted time.

Example

```
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
CString s = ts.Format( "Total days: %D, hours: %H, mins: %M, secs: %S"
    );
ASSERT( s == "Total days: 3, hours: 01, mins: 05, secs: 12" );
```

CTimeSpan::GetDays

LONG GetDays( ) const;

Remarks Returns the number of complete days. This value may be negative if the time span is negative.

Example

```
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetDays() == 3 );
```

CTimeSpan::GetHours

int GetHours( ) const;

Remarks Returns the number of hours in the current day. The range is –23 through 23.

Example

```
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetHours() == 1 );
ASSERT( ts.GetMinutes() == 5 );
ASSERT( ts.GetSeconds() == 12 );
```

CTimeSpan::GetMinutes

int GetMinutes( ) const;

Remarks Returns the number of minutes in the current hour. The range is –59 through 59.

Example See the example for **GetHours**.

CTimeSpan::GetSeconds

int GetSeconds( ) const;

Remarks Returns the number of seconds in the current minute. The range is –59 through 59.

Example See the example for **GetHours**.

CTimeSpan::GetTotalHours

LONG GetTotalHours( ) const;

Remarks Returns the total number of complete hours in this **CTimeSpan**.

Example

```
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ASSERT( ts.GetTotalHours() == 73 );
ASSERT( ts.GetTotalMinutes() == 4385 );
ASSERT( ts.GetTotalSeconds() == 263112 );
```

CTimeSpan::GetTotalMinutes

LONG GetTotalMinutes( ) const;

Remarks Returns the total number of complete minutes in this **CTimeSpan**.

Example See the example for **GetTotalHours**.

CTimeSpan::GetTotalSeconds

LONG GetTotalSeconds( ) const;

Remarks Returns the total number of complete seconds in this **CTimeSpan**.

Example See the example for **GetTotalHours**.

Operators

CTimeSpan::operator =

const CTimeSpan& operator =(const CTimeSpan& *timeSpanSrc* **);**

Remarks The overloaded assignment operator copies the source **CTimeSpan** *timeSpanSrc* object into this **CTimeSpan** object.

See Also **CTimeSpan::CTimeSpan**

Example

```
CTimeSpan ts1;
CTimeSpan ts2( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ts1 = ts2;
ASSERT( ts1 == ts2 );
```

CTimeSpan::operator +, –

CTimeSpan operator +(CTimeSpan *timeSpan* **) const;**

CTimeSpan operator –(CTimeSpan *timeSpan* **) const;**

Remarks These two operators allow you to add and subtract **CTimeSpan** objects to and from each other.

Example

```
CTimeSpan ts1( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
CTimeSpan ts2( 100 ); // 100 seconds
CTimeSpan ts3 = ts1 + ts2;
ASSERT( ts3.GetSeconds() == 52 ); // 6 mins, 52 secs
```

CTimeSpan::operator +=, –=

const CTimeSpan& operator +=(CTimeSpan *timeSpan* **);**

const CTimeSpan& operator –=(CTimeSpan *timeSpan* **);**

Remarks These operators allow you to add and subtract a **CTimeSpan** object to and from this **CTimeSpan**.

Example

```
CTimeSpan ts1( 10 ); // 10 seconds
CTimeSpan ts2( 100 ); // 100 seconds
ts2 -= ts1;
ASSERT( ts2.GetTotalSeconds() == 90 );
```

CTimeSpan Comparison Operators

BOOL operator ==(CTimeSpan *timeSpan* **) const;**

BOOL operator !=(CTimeSpan *timeSpan* **) const;**

BOOL operator <(CTimeSpan *timeSpan* **) const;**

BOOL operator >(CTimeSpan *timeSpan* **) const;**

BOOL operator <=(CTimeSpan *timeSpan* **) const;**

BOOL operator >=(CTimeSpan *timeSpan* **) const;**

Remarks

These operators compare two relative time values. They return **TRUE** if the condition is true; otherwise **FALSE**.

Example

```
CTimeSpan ts1( 100 );
CTimeSpan ts2( 110 );
ASSERT( ( ts1 != ts2 ) && ( ts1 < ts2 ) && ( ts1 <= ts2 ) );
```

CTimeSpan::operators <<, >>

friend CDumpContext& operator <<(CDumpContext& *dc*, **CTimeSpan** *timeSpan* **);**

friend CArchive& operator <<(CArchive& *ar*, **CTimeSpan** *timeSpan* **);**

friend CArchive& operator >>(CArchive& *ar*, **CTimeSpan&** *timeSpan* **);**

Remarks

The **CTimeSpan** insertion (<<) operator supports diagnostic dumping and storing to an archive. The extraction (>>) operator supports loading from an archive.

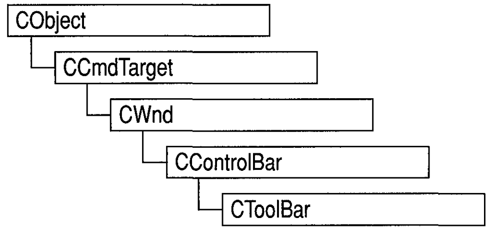
When you send a **CTimeSpan** object to the dump context, the value is displayed in a human-readable format that shows days, hours, minutes, and seconds.

Example

```
CTimeSpan ts( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
#ifdef _DEBUG
afxDump << ts << "\n";
#endif
// Prints 'CTimeSpan(3 days, 1 hours, 5 minutes and 12 seconds)'

extern CArchive ar;
if( ar.IsLoading( ))
  ar >> ts;
else
  ar << ts;
```

class CToolBar : public CControlBar

Objects of the class **CToolBar** are control bars that have a row of bitmapped buttons and optional separators. The buttons can act like pushbuttons, check-box buttons, or radio buttons. **CToolBar** objects are usually embedded members of frame-window objects derived from the class **CFrameWnd** or **CMDIFrameWnd**.



To create a toolbar from within a frame-window object, follow these steps:

1. Construct the **CToolBar** object.
2. Call the **Create** function to create the Windows toolbar and attach it to the **CToolBar** object.
3. Call **LoadBitmap** to load the bitmap that contains the toolbar button images.
4. Call **SetButtons** to set the button style and associate each button with an image in the bitmap.

All the button images in the toolbar are taken from one bitmap, which must contain one image for each button. All images must be the same size; the default is 16 pixels wide and 15 pixels high. Images must be side by side in the bitmap.

The **SetButtons** function takes a pointer to an array of control IDs and an integer that specifies the number of elements in the array. The function sets each button's ID to the value of the corresponding element of the array and assigns each button an image index, which specifies the position of the button's image in the bitmap. If an array element has the value **ID_SEPARATOR**, no image index is assigned.

The order of the images in the bitmap is typically the order in which they are drawn on the screen, but you can use the **SetButtonInfo** function to change the relationship between image order and drawing order.

All buttons in a toolbar are the same size. The default is 24 x 22 pixels, in accordance with *The Windows Interface: An Application Design Guide*. Any additional space between the image and button dimensions is used to form a border around the image.

Each button has one image. The various button states and styles (pressed, up, down, disabled, disabled down, and indeterminate) are generated from that one image. Although bitmaps can be any color, you can achieve the best results with images in black and shades of gray.

Toolbar buttons imitate pushbuttons by default. However, toolbar buttons can also imitate check-box buttons or radio buttons. Check-box buttons have three states: checked, cleared, and indeterminate. Radio buttons have only two states: checked and cleared.

To create a check-box button, assign it the style **TBBS_CHECKBOX** or use a **CCmdUI** object's **SetCheck** member function in an **ON_UPDATE_COMMAND_UI** handler. Calling **SetCheck** turns a pushbutton into a check-box button. Pass **SetCheck** an argument of 0 for unchecked, 1 for checked, or 2 for indeterminate.

To create a radio button, call a **CCmdUI** object's **SetRadio** member function from an **ON_UPDATE_COMMAND_UI** handler. Pass **SetRadio** an argument of 0 for unchecked or nonzero for checked. In order to provide a radio group's mutually exclusive behavior, you must have **ON_UPDATE_COMMAND_UI** handlers for all of the buttons in the group.

See Also **CControlBar**, **CToolBar::Create**, **CToolBar::LoadBitmap**, **CToolBar::SetButtons**, **CCmdUI::SetCheck**, **CCmdUI::SetRadio**

Construction/Destruction — Public Members

CToolBar	Constructs a **CToolBar** object.
Create	Creates the Windows toolbar and attaches it to the **CToolBar** object.
SetSizes	Sets the sizes of buttons and their bitmaps.
SetHeight	Sets the height of the toolbar.
LoadBitmap	Loads the bitmap containing bitmap-button images.
SetButtons	Sets button styles and an index of button images within the bitmap.

Attributes — Public Members

CommandToIndex	Returns the index of a button with the given command ID.
GetItemID	Returns the command ID of a button or separator at the given index.
GetItemRect	Gets the display rectangle for the item at the given index.
GetButtonInfo	Gets a button's ID, style, and image number.
SetButtonInfo	Sets a button's ID, style, and image number.

Member Functions

CToolBar::CommandToIndex

int CommandToIndex(UINT *nIDFind* **);**

nIDFind Command ID of a toolbar button.

Remarks Returns the index of the first toolbar button, starting at position 0, whose command ID matches *nIDFind*.

Return Value The index of the button, or –1 if no button has the given command ID.

See Also **CToolBar::GetItemId**

CToolBar::Create

BOOL Create(CWnd* *pParentWnd*, **DWORD** *dwStyle* = **WS_CHILD | WS_VISIBLE | CBRS_TOP, UINT** *nID* = **AFX_IDW_TOOLBAR);**

pParentWnd Pointer to the window that is the toolbar's parent.

dwStyle The toolbar style. Additional toolbar styles supported are:

- **CBRS_TOP** Control bar is at top of the frame window.
- **CBRS_BOTTOM** Control bar is at bottom of the frame window.
- **CBRS_NOALIGN** Control bar is not repositioned when the parent is resized.

nID The toolbar's child-window ID.

Remarks Creates a Windows toolbar (a child window) and associates it with the **CToolBar** object. Also sets the toolbar height to a default value.

Return Value Nonzero if successful; otherwise 0.

See Also **CToolBar::CToolBar**, **CToolBar::LoadBitmap**, **CToolBar::SetButtons**

CToolBar::CToolBar

CToolBar( );

Remarks

Constructs a **CToolBar** object and sets the default sizes.

Call **Create** to create the toolbar window.

See Also

CToolBar::Create

CToolBar::GetButtonInfo

void GetButtonInfo(int *nIndex*, **UINT&** *nID*, **UINT&** *nStyle*, **int&** *iImage* **)
const;**

nIndex Index of the toolbar button or separator whose information is to be retrieved.

nID Reference to a **UINT** that is set to the command ID of the button.

nStyle Reference to a **UINT** that is set to the style of the button.

iImage Reference to an integer that is set to the index of the button's image within the bitmap.

Remarks

Gets the control ID, style, and image index of the toolbar button or separator at the location specified by *nIndex*. Those values are assigned to the variables referenced by *nID*, *nStyle*, and *iImage*. The image index is the position of the image within the bitmap that contains images for all the toolbar buttons. The first image is at position 0.

If *nIndex* specifies a separator, *iImage* is set to the separator width in pixels.

See Also

CToolBar::SetButtonInfo, **CToolBar::GetItemID**

CToolBar::GetItemID

UINT GetItemID(int *nIndex* **) const;**

nIndex Index of the item (button or separator) whose ID is to be retrieved.

Remarks Returns the command ID of the button or separator specified by *nIndex*. Separators return **ID_SEPARATOR**.

Return Value The command ID of the button or separator specified by *nIndex*.

See Also **CToolBar::CommandToIndex**, **CControlBar::GetCount**

CToolBar::GetItemRect

void GetItemRect(int *nIndex*, **LPRECT** *lpRect* **);**

nIndex Index of the item (button or separator) whose rectangle coordinates are to be retrieved.

lpRect Address of the **RECT** structure that will contain the item's coordinates.

Remarks Fills the **RECT** structure whose address is contained in *lpRect* with the coordinates of the button or separator specified by *nIndex*. Coordinates are in pixels relative to the upper-left corner of the toolbar.

Use **GetItemRect** to get the coordinates of a separator you want to replace with a combo box or other control.

See Also **CToolBar::CommandToIndex**

CToolBar::LoadBitmap

BOOL LoadBitmap(LPCSTR *lpszResourceName* **);**

BOOL LoadBitmap(UINT *nIDResource* **);**

lpszResourceName Pointer to the resource name of the bitmap to be loaded.

nIDResource Resource ID of the bitmap to be loaded.

Remarks Loads the bitmap specified by *lpszResourceName* or *nIDResource*. The bitmap should contain one image for each toolbar button. If the images are not of the standard size (16 pixels wide and 15 pixels high), call **SetSizes** to set the button sizes and their images.

Return Value Nonzero if successful; otherwise 0.

See Also **CToolBar::Create**, **CToolBar::SetButtons**, **CToolBar::SetSizes**

CToolBar::SetButtonInfo

void SetButtonInfo(int *nIndex*, **UINT** *nID*, **UINT** *nStyle*, **int** *iImage* **);**

nIndex Index of the button or separator whose information is to be set.

nID The value to which the button's command ID is set.

nStyle The new button style. The following button styles are supported:

- **TBBS_BUTTON** Standard pushbutton (default)
- **TBBS_SEPARATOR** Separator
- **TBBS_CHECKBOX** Auto check-box button

iImage New index for the button's image within the bitmap.

Remarks

Sets the button's command ID, style, and image number. For separators, which have the style **TBBS_SEPARATOR**, this function sets the separator's width in pixels to the value stored in *iImage*.

For information on bitmap images and buttons, see the class overview and **CToolBar::LoadBitmap**.

See Also

CToolBar::GetButtonInfo, **CToolBar::LoadBitmap**

CToolBar::SetButtons

BOOL SetButtons(const UINT FAR* *lpIDArray*, **int** *nIDCount* **);**

lpIDArray Pointer to an array of command IDs.

nIDCount Number of elements in the array pointed to by *lpIDArray*.

Remarks

Sets each toolbar button's command ID to the value specified by the corresponding element of the array *lpIDArray*. If an element of the array has the value **ID_SEPARATOR**, a separator is created in the corresponding position of the toolbar. This function also sets each button's style to **TBBS_BUTTON** and each separator's style to **TBBS_SEPARATOR**, and assigns an image index to each button. The image index specifies the position of the button's image within the bitmap.

You do not need to account for separators in the bitmap because this function does not assign image indexes for separators. If your toolbar has buttons at positions 0, 1, and 3 and a separator at position 2, the images at positions 0, 1, and 2 in your bitmap are assigned to the buttons at positions 0, 1, and 3, respectively.

If *lpIDArray* is **NULL**, this function allocates space for the number of items specified by *nIDCount*. Use **SetButtonInfo** to set each item's attributes.

Return Value Nonzero if successful; otherwise 0.

See Also **CToolBar::Create**, **CToolBar::SetButtonInfo**

CToolBar::SetHeight

void SetHeight(int *cyHeight* **);**

cyHeight The height in pixels of the toolbar.

Remarks Sets the toolbar's height to the value, in pixels, specified in *cyHeight*.

After calling **SetSizes**, use this function to override the standard toolbar height. If the height is too small, the buttons will be clipped at the bottom.

If this function is not called, the framework uses the size of the button to determine the toolbar height.

See Also **CToolBar::SetSizes**, **CToolBar::SetButtonInfo**, **CToolBar::SetButtons**

CToolBar::SetSizes

void SetSizes(Size *sizeButton*, **Size** *sizeImage* **);**

sizeButton The size in pixels of each button.

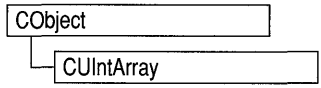
sizeImage The size in pixels of each image.

Remarks Sets the toolbar's buttons to the size, in pixels, specified in *sizeButton*. The *sizeImage* parameter must contain the size, in pixels, of the images in the toolbar's bitmap. The dimensions in *sizeButton* must be sufficient to hold the image plus 3 pixels on each side for the button outline. This function also sets the toolbar height to fit the buttons.

Call this function only for toolbars that do not follow *The Windows Interface: An Application Design Guide* recommendations for button and image sizes.

See Also **CToolBar::LoadBitmap**, **CToolBar::SetButtonInfo**, **CToolBar::SetButtons**, **CToolBar::SetHeight**

class CUIntArray : public CObject



The **CUIntArray** class supports arrays of unsigned integers. An unsigned integer, or **UINT**, differs from words and doublewords in that the physical size of a **UINT** can change depending on the target operating environment. Under Windows version 3.1, a **UINT** is the same size as a **WORD**. Under Windows NT, a **UINT** is the same size as a doubleword. The member functions of **CUIntArray** are similar to the member functions of class **CObArray**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **UINT**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
UINT CUIntArray::GetAt( int <nIndex> ) const;
```

CUIntArray incorporates the **IMPLEMENT_DYNAMIC** macro to support run-time type access and dumping to a **CDumpContext** object. If you need a dump of individual unsigned integer elements, you must set the depth of the dump context to 1 or greater. Unsigned integer arrays may not be serialized.

#include <afxcoll.h>

Construction/Destruction—Public Members

CUIntArray	Constructs an empty array for unsigned integers.
~CUIntArray	Destroys a **CUIntArray** object.

Bounds—Public Members

GetSize	Gets the number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations—Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; the array is not allowed to grow.
ElementAt	Returns a temporary reference to the element pointer within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

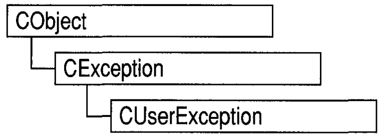
Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

class CUserException : public CException



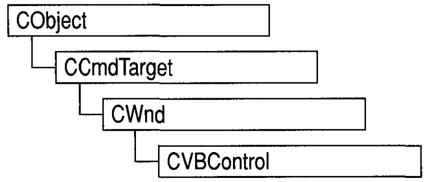
A **CUserException** is thrown to stop an end-user operation. Use **CUserException** when you want to use the throw/catch exception mechanism for application-specific exceptions. "User" in the class name can be interpreted as "my user did something exceptional that I need to handle." A **CUserException** is usually thrown after calling the global function **AfxMessageBox** to notify the user that an operation has failed. When you write an exception handler, handle the exception specially since the user usually has already been notified of the failure. The framework throws this exception in some cases. To throw a **CUserException** yourself, alert the user and then call the global function **AfxThrowUserException**. In this example, a function with operations that may fail alerts the user and throws a **CUserException**. The calling function catches the exception and handles it specially:

```
void DoSomeOperation( )
{
    // Processing
    // If something goes wrong...
    AfxMessageBox( "The x operation failed" );
    AfxThrowUserException( );
}

BOOL TrySomething( )
{
    TRY
    {
        // Could throw a CUserException or other exception.
        DoSomeOperation( );
    }
    CATCH( CUserException, e )
    {
        return FALSE;    // User already notified.
    }
    AND_CATCH( CException, e )
    {
        // For other exception types, notify user here.
        AfxMessageBox( "Some operation failed" );
        return FALSE;
    }
    END_CATCH
    return TRUE;   // No exception thrown.
}
```

#include <afxwin.h>

See Also **CException**, **AfxMessageBox**, **AfxThrowUserException**

class CVBControl : public CWnd



Class **CVBControl** allows you to take advantage of the large number of custom controls available for the Visual Basic programming system and Visual C++. You can load controls, get their properties, set their properties, change their screen location, and perform many other operations. Within your application, every VBX control, such as a dialog box or scroll bar, becomes an object of class **CVBControl**.

You can use VBX controls either in dialog boxes or application windows. For more information on programming with VBX controls using class **CVBControl**, see Chapter 17 of the *Class Library User's Guide* and Technical Note 27 in MSVC\HELP\MFCNOTES.HLP.

#include <afxwin.h>

#include <afxext.h>

Data Members—Public Members

m_nError	Contains a VBX or control-defined error value when a **CVBControl** "get" or "set" member function (such as **GetNumProperty**) generates an error.

Construction/Destruction—Public Members

CVBControl	Constructs a **CVBControl** object.

Initialization—Public Members

Create	Creates the control after it has been constructed.

Property Access —Public Members

GetFloatProperty	Gets the floating-point value assigned to a floating-point property.
GetNumProperty	Gets the integer value assigned to an integer-valued control property.
GetPictureProperty	Gets a handle to a picture that is assigned to a picture property.
GetStrProperty	Gets the string assigned to a string property of a control.
SetFloatProperty	Sets a floating-point property to the specified value.

SetNumProperty	Sets an integer-valued property to the specified value.
SetPictureProperty	Sets a picture property to a specified picture.
SetStrProperty	Sets a string property to the specified string.

Attributes—Public Members

GetEventIndex	Returns the index number associated with the specified event.
GetEventName	Returns the name of the event associated with the specified index number.
GetNumEvents	Returns the number of events associated with the control.
GetNumProps	Returns the number of properties associated with the control.
GetPropFlags	Returns a 32-bit value that specifies the property flags for the control.
GetPropIndex	Returns the index number assigned to a control property.
GetPropName	Returns the name of the property associated with the specified index number.
GetPropType	Returns the type of the property.
GetVBXClass	Returns the name of the control class.
IsPropArray	Checks whether the specified property is an array.

Methods—Public Members

AddItem	Adds items to a list managed by a list-box control or combo-box control.
Move	Moves a control to a specified location and resizes the control at the same time.
Refresh	Updates a control to reflect changes that have been made to the control or to the environment.
RemoveItem	Removes an item from a list managed by the control.

Operations—Public Members

BeginNewVBHeap	Causes the creation of a new VBX-control heap when the next VBX control is created.
CloseChannel	Disassociates the file associated with the specified channel number.
GetChannel	Retrieves a pointer to a **CFile** object currently associated with the specified file channel number.
OpenChannel	Associates a file with a file channel number.

Member Functions

CVBControl::AddItem

void AddItem(LPCSTR *lpszItem*, **LONG** *lIndex* **);**

lpszItem The string associated with the item in the list.

lIndex The index number of the item in the list.

Remarks

Call this function to add items to a list in a list box or combo box in a VBX control. This function mimics Visual Basic's **AddItem** method. For additional information, see the *Visual Basic Programmer's Guide.*

See Also

CVBControl::RemoveItem, **CVBControl::Refresh**

CVBControl::BeginNewVBHeap

static void BeginNewVBHeap( );

Remarks

Starts a new VBX-control heap space when the next VBX control is created. All VBX controls that are created after this function is called will be placed in a new heap space. Use this function only if you suspect that a VBX control is running out of memory.

The **CVBControl** object itself is not allocated in the VBX heap space. Only the extra data needed by the control, such as its properties, are allocated. For complex VBX controls, such as graphs or grids, or for large numbers of VBX controls, there may be insufficient heap space to store all of the property data. **BeginNewVBHeap** allows you to allocate extra heaps as needed. For additional information, see Chapter 17 of the *Class Library User's Guide* and Technical Note 27 in MSVC\HELP\MFCNOTES.HLP.

CVBControl::CloseChannel

static BOOL PASCAL CloseChannel(WORD *wChannel* **);**

wChannel The number of the channel that is to be closed.

Remarks

Call this function to disassociate a file from the specified channel number. Typically, you use **CloseChannel** to close a channel that has been opened using the **CVBControl::OpenChannel** member function. **CloseChannel** does not physically close a file—it only disassociates a file from its channel number.

Return Value

Nonzero if the function was successful; otherwise 0.

See Also

CVBControl::OpenChannel, **CVBControl::GetChannel**

CVBControl::Create

BOOL Create(LPCSTR *lpszWindowName*, **DWORD** *dwStyle*,
const RECT& *rect*, **CWnd*** *pParentWnd*, **UINT** *nID*,
CFile* *pFile* = **NULL, BOOL** *bAutoDelete* = **FALSE);**

lpszWindowName A string containing the VBX filename, the control name, and the window text for the control. This string must have the following format: "*VBX file*;*control name*;*window text*". For example, "THREED.VBX;Check 3D;Check this Box".

dwStyle The window style of the control.

rect The rectangle that is to contain the control. This can either be a standard **RECT** structure or a **CRect** object.

pParentWnd A pointer to the parent window of the control.

nID The control's ID. This is usually specified with a **#define** statement in a header file.

pFile A pointer to the file containing saved information for the attributes of the control. This will usually be **NULL** for manually created controls.

bAutoDelete Indicates whether the control should be automatically deleted on exit. Set this parameter to **TRUE** if you want the control to be automatically deleted. Otherwise set this parameter to **FALSE** and delete the control explicitly.

Remarks

Call this member function to create the VBX control. Before using **Create**, you must use the **CVBControl** constructor to construct the object. In most cases, the *dwStyle* parameter should be **NULL** to allow the use of the window styles specified by the control. For advanced usage, you can override the control's specification and use one of the many window styles defined in the Windows environment or a bitwise combination of more than one style. The **WS_CHILD** style is automatically included with any styles specified, so using **WS_CHILD** disables the default styles and adds no other styles. For a complete list of window styles and their meanings, see **CWnd::Create**.

The file pointed to by *pFile* contains a binary representation of the initial values of a control's properties. The format of this file varies for each control. App Studio generates this binary information as part of a **DLGINIT** resource for controls loaded into a dialog box or form view. Since the framework automatically creates controls loaded in this manner, it is not necessary to call **Create**. The framework does not provide a means of generating these binary files, so this parameter will most often be **NULL**. If *pFile* is not **NULL**, *dwStyle* must be **NULL** for the control to operate properly. For more information on the format of this file, see Chapter 17 of the *Class Library User's Guide* and Technical Note 27 in MSVC\HELP\MFCNOTES.HLP.

Return Value Nonzero if the control was successfully created; otherwise 0.

See Also **CVBControl::CVBControl**

CVBControl::CVBControl

CVBControl( );

Remarks Call this function to construct a VBX control. Constructing a VBX-control object does not display the object. You must call the **Create** member function after calling the constructor to create the control. Use **CWnd::ShowWindow** to display the control if it is not displayed by default.

See Also **CVBControl::Create**

CVBControl::GetChannel

static CFile* PASCAL GetChannel(WORD *wChannel* **);**

wChannel The channel number associated with the desired file.

Remarks Call this function to determine which file is currently associated with a channel number. For more information on channel numbers, see **CVBControl::OpenChannel**.

Return Value A pointer to the **CFile** object currently associated with the file number *wChannel*.

See Also **CVBControl::CloseChannel**, **CVBControl::OpenChannel**

CVBControl::GetEventIndex

int GetEventIndex(LPCSTR *lpszEventName* **) const;**

lpszEventName The name associated with the event whose index you want returned.

Return Value The index number associated with the event specified by *lpszEventName*.

See Also **CVBControl::GetEventName**, **CVBControl::GetPropIndex**

CVBControl::GetEventName

LPCSTR GetEventName(int *nIndex* **) const;**

nIndex The index number associated with the event whose name you want returned.

Return Value The name of the event associated with the index number *nIndex*.

See Also **CVBControl::GetEventIndex**, **CVBControl::GetPropName**

CVBControl::GetFloatProperty

float GetFloatProperty(int *nPropIndex*, **int** *index* = **0);**

float GetFloatProperty(LPCSTR *lpszPropName*, **int** *index* = **0);**

nPropIndex The index of the floating-point property whose value you want returned.

index Specifies the index of the array element whose value you want returned if the property is an array of floating-point numbers. The default index is 0.

lpszPropName The name of the floating-point property whose value you want returned.

Remarks Call this function to retrieve the floating-point value assigned to a floating-point control property. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value The floating-point value of the property, or the floating-point value of a specified array element if the property is an array.

See Also **CVBControl::GetNumProperty**, **CVBControl::GetStrProperty**, **CVBControl::GetPictureProperty**

CVBControl::GetNumEvents

int GetNumEvents( ) const;

Return Value The number of events associated with the control.

See Also **CVBControl::GetNumProps**

CVBControl::GetNumProperty

LONG GetNumProperty(int *nPropIndex*, **int** *index* = **0);**

LONG GetNumProperty(LPCSTR *lpszPropName*, **int** *index* = **0);**

nPropIndex The index of the integer property whose value you want returned.

index Specifies the index of the array element whose value you want returned if the property is an array of integers. The default index is 0.

lpszPropName The name of the integer property whose value you want returned.

Remarks Call this function to retrieve the value assigned to an integer-valued or Boolean control property. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value The integer value of the property, or the integer value of a specified array element if the property is an array.

See Also **CVBControl::GetFloatProperty**, **CVBControl::GetStrProperty**, **CVBControl::GetPictureProperty**

CVBControl::GetNumProps

int GetNumProps( ) const;

Return Value The number of properties the control has.

See Also **CVBControl::GetNumEvents**

CVBControl::GetPictureProperty

HPIC GetPictureProperty(int *nPropIndex*, **int** *index* = **0);**

HPIC GetPictureProperty(LPCSTR *lpszPropName*, **int** *index* = **0);**

nPropIndex The index number of the property whose value you want returned.

index Specifies the index of the array element whose pointer you want returned if the property is an array of picture pointers. The default index is 0.

lpszPropName The name of the property whose value you want returned.

Remarks Call this function to retrieve a handle to a picture that is assigned to a picture property. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value A handle to the picture associated with the property, or the handle value of a specified array element if the property is an array.

See Also **CVBControl::GetFloatProperty**, **CVBControl::GetStrProperty**, **CVBControl::GetNumProperty**

CVBControl::GetPropFlags

DWORD GetPropFlags(int *nIndex* **) const;**

nIndex The index number of the property whose flags you want returned.

Remarks Returns a 32-bit value specifying the property flags for the property.

See Also **CVBControl::GetNumProps**

CVBControl::GetPropIndex

int GetPropIndex(LPCSTR *lpszPropName* **) const;**

lpszPropName The name of the property whose index you want returned.

Remarks Allows you to use an index number instead of a string containing the name of the property to refer to a particular property of any instance of a single type of control.

Return Value The integer index assigned to the control property.

See Also **CVBControl::GetPropName**, **CVBControl::GetEventIndex**

CVBControl::GetPropName

LPCSTR GetPropName(int *nIndex* **) const;**

nIndex The index number of the property whose name you want returned.

Return Value The name of the property associated with the specified index.

See Also **CVBControl::GetPropIndex**, **CVBControl::GetEventName**

CVBControl::GetPropType

UINT GetPropType(int *nIndex* **) const;**

nIndex The index number of the property whose type you want returned.

Return Value The type of the property associated with *nIndex*. The property type can have one of the following values, as defined in AFXEXT.H:

Type	Value	Get/Set Function to Use
DT_HSZ	0x01	Get/SetStrProperty
DT_SHORT	0x02	Get/SetNumProperty
DT_LONG	0x03	Get/SetNumProperty
DT_BOOL	0x04	Get/SetNumProperty
DT_COLOR	0x05	Get/SetNumProperty
DT_ENUM	0x06	Get/SetNumProperty

Type	Value	Get/Set Function to Use
DT_REAL	0x07	Get/SetFloatProperty
DT_XPOS	0x08	Get/SetNumProperty
DT_XSIZE	0x09	Get/SetNumProperty
DT_YPOS	0x0A	Get/SetNumProperty
DT_YSIZE	0x0B	Get/SetNumProperty
DT_PICTURE	0x0C	Get/SetPictureProperty

See Also **CVBControl::GetFloatProperty**, **CVBControl::GetStrProperty**, **CVBControl::GetPictureProperty**, **CVBControl::GetNumProperty**

CVBControl::GetStrProperty

CString GetStrProperty(int *nPropIndex*, **int** *index* = **0);**

CString GetStrProperty(LPCSTR *lpszPropName*, **int** *index* = **0);**

nPropIndex The index number of the property whose value you want returned.

index Specifies the index of the array element whose value you want returned if the property is an array of strings. The default index is 0.

lpszPropName The name of the property whose value you want returned.

Remarks Call this function to retrieve a string property of a VBX control. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value The string assigned to the specified property. If the property is an array of strings, the string assigned to the specified array element is returned.

See Also **CVBControl::GetFloatProperty**, **CVBControl::GetPictureProperty**, **CVBControl::GetNumProperty**

CVBControl::GetVBXClass

LPCSTR GetVBXClass( ) const;

Remarks Returns the class name that is used during the **Create** call. When a control is created, the window class used will have a "Thunder" prefix added to the class name.

Return Value The name of the control class.

See Also **CVBControl::Create**

CVBControl::IsPropArray

BOOL IsPropArray(int *nIndex* **) const;**

nIndex The index number of the property.

Remarks Checks whether the property associated with *nIndex* is a property array. A property array is a property that consists of an array of values.

Return Value Nonzero if the property associated with *nIndex* is an array; otherwise 0.

See Also **CVBControl::GetPropType**

CVBControl::Move

void Move(RECT& *rect* **);**

rect A rectangle specifying the new location and size of the control.

Remarks Call this function to move a VBX control to the location specified by *rect*. The upper-left corner of the control is moved to the coordinates **rect.left** and **rect.top**, and the control is resized to fit within the rectangle.

CVBControl::OpenChannel

static void PASCAL OpenChannel(CFile* *pFile*, **WORD** *wChannel* **);**

pFile A pointer to the file that is to be associated with the specified channel number.

wChannel The channel number you want associated with the specified file.

Remarks Call this function to associate the file pointed to by *pFile* with the *wChannel* file number. The three member functions **OpenChannel**, **CloseChannel**, and **GetChannel** provide a mechanism through which controls can access files as they normally do in Visual Basic—through file numbers. Use these functions to handle control properties that access files. For example, if a control is able to send the contents of a list box to disk, these three member functions are typically used to support the necessary file I/O.

See Also **CVBControl::CloseChannel**, **CVBControl::GetChannel**

CVBControl::Refresh

void Refresh( );

Remarks Call this function to update a VBX control to reflect changes that have been made to the control or to the environment. For example, if a list box contains a list of files in the current directory, and a new file was created in that directory, **Refresh** will regenerate the list of files in the list box to show the new file. This function mimics Visual Basic's **Refresh** method. For additional information, see the *Visual Basic Programmer's Guide*.

See Also **CVBControl::AddItem**, **CVBControl::RemoveItem**

CVBControl::RemoveItem

void RemoveItem(LONG *lIndex* **);**

lIndex The index number of the item you want removed from the list.

Remarks

Call this function to remove an item from a list box or combo box in a VBX control. This function mimics Visual Basic's **RemoveItem** method. For additional information, see the *Visual Basic Programmer's Guide*.

See Also

CVBControl::AddItem, **CVBControl::Refresh**

CVBControl::SetFloatProperty

BOOL SetFloatProperty(int *nPropIndex*, **float** *value*, **int** *index* = **0);**

BOOL SetFloatProperty(LPCSTR *lpszPropName*, **float** *value*, **int** *index* = **0);**

nPropIndex The index number of the property whose value you want to set.

value The new floating-point value for the property.

index Specifies the index of the array element whose value you want to set if the property is an array of floating-point numbers. The default index is 0.

lpszPropName The name of the property whose value you want to set.

Remarks

Sets a floating-point property to the value specified by *value*. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value

Nonzero if the function was successful; otherwise 0.

See Also

CVBControl::SetStrProperty, **CVBControl::SetPictureProperty**, **CVBControl::SetNumProperty**

CVBControl::SetNumProperty

BOOL SetNumProperty(int *nPropIndex*, **LONG** *lValue*, **int** *index* = **0);**

BOOL SetNumProperty(LPCSTR *lpszPropName*, **LONG** *lValue*, **int** *index* = **0);**

nPropIndex The index number of the property whose value you want to set.

lValue The new value for the property.

index Specifies the index of the array element whose value you want to set if the property is an array of integers. The default index is 0.

lpszPropName The name of the property whose value you want to set.

Remarks Sets an integer-valued property to the value specified by *lValue*. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value Nonzero if the function was successful; otherwise 0.

See Also **CVBControl::SetStrProperty**, **CVBControl::SetPictureProperty**, **CVBControl::SetFloatProperty**

CVBControl::SetPictureProperty

BOOL SetPictureProperty(int *nPropIndex*, **HPIC** *hPic*, **int** *index* = **0);**

BOOL SetPictureProperty(LPCSTR *lpszPropName*, **HPIC** *hPic*, **int** *index* = **0);**

nPropIndex The index of the property whose value you want to set.

hPic A handle to a picture you want to assign to the specified property.

index Specifies the index of the array element whose value you want to set if the property is an array of picture pointers. The default index is 0.

lpszPropName The name of the property whose value you want to set.

Remarks Sets a picture property to a specified picture identified by *hPic*. The property can be referenced either through its index, *nPropIndex*, or through its name, *lpszPropName*.

Return Value Nonzero if the function was successful; otherwise 0.

See Also **CVBControl::SetStrProperty**, **CVBControl::SetNumProperty**, **CVBControl::SetFloatProperty**

CVBControl::SetStrProperty

BOOL SetStrProperty(int *nPropIndex*, **LPCSTR** *lpszValue*, **int** *index* = **0);**

BOOL SetStrProperty(LPCSTR *lpszPropName*, **LPCSTR** *lpszValue*, **int** *index* = **0);**

nPropIndex The index number of the property whose value you want to set.

lpszValue The new string value for the property.

index Specifies the index of the array element whose value you want to set if the property is an array of strings. The default index is 0.

lpszPropName The name of the property whose value you want to set.

Remarks

Sets a string property to the string specified by *lpszValue*. The property can be referenced either through its index *nPropIndex*, or through its name, *lpszPropName*.

Return Value

Nonzero if the function was successful; otherwise 0.

See Also

CVBControl::SetNumProperty, **CVBControl::SetPictureProperty**, **CVBControl::SetFloatProperty**

Data Members

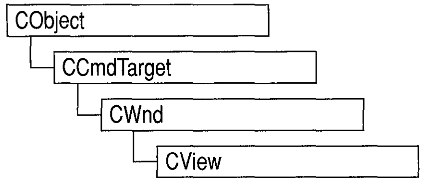
CVBControl::m_nError

Remarks

m_nError is a public variable of type **int**. This data member contains a VBX or control-defined error value when a **CVBControl** "get" or "set" member function (such as **GetPropType**) generates an error. This data member can be used to identify and take action on a wide range of errors, such as "insufficient memory." Normally, however, it is not necessary to check for errors on these operations.

The value of **m_nError** is set to the Visual Basic error code associated with the error. For a list of these error codes, see the *Visual Basic Programmer's Guide*.

class CView : public CWnd



The **CView** class provides the basic functionality for user-defined view classes. A view is attached to a document and acts as an intermediary between the document and the user: the view renders an image of the document on the screen or printer and interprets user input as operations upon the document.

A view is a child of a frame window. More than one view can share a frame window, as in the case of a splitter window. The relationship between a view class, a frame window class, and a document class is established by a **CDocTemplate** object. When the user opens a new window or splits an existing one, the framework constructs a new view and attaches it to the document.

A view can be attached to only one document, but a document can have multiple views attached to it at once—for example, if the document is displayed in a splitter window or in multiple child windows in a multiple document interface (MDI) application. Your application can support different types of views for a given document type; for example, a word-processing program might provide both a complete text view of a document and an outline view that shows only the section headings. These different types of views can be placed in separate frame windows or in separate panes of a single frame window if you use a splitter window.

A view may be responsible for handling several different types of input, such as keyboard input or mouse input, as well as commands from menus, toolbars, or scroll bars. A view receives commands forwarded by its frame window. If the view does not handle a given command, it forwards the command to its associated document. Like all command targets, a view handles messages via a message map.

The view is responsible for displaying and modifying the document's data but not for storing it. The document provides the view with the necessary details about its data. You can let the view access the document's data members directly, or you can provide member functions in the document class for the view class to call.

When a document's data changes, the view responsible for the changes typically calls the **CDocument::UpdateAllViews** function for the document, which notifies all the other views by calling the **OnUpdate** member function for each. The default implementation of **OnUpdate** invalidates the view's entire client area. You can override it to invalidate only those regions of the client area that map to the modified portions of the document.

To use **CView**, derive a class from it and implement the **OnDraw** member function to perform screen display. You can also use **OnDraw** to perform printing and print preview. The framework handles the print loop for printing and previewing your document.

A view handles scroll-bar messages in its **OnHScroll** and **OnVScroll** member functions. You can implement scroll-bar message handling in these functions, or you can use the derived class **CScrollView** to handle scrolling for you.

Besides **CScrollView**, the Microsoft Foundation Class Library provides two other classes derived from **CView**:

- **CFormView**, a scrollable view that contains dialog-box controls and is based on a dialog template resource.
- **CEditView**, a view that provides a simple multiline text editor. You can use a **CEditView** object as a control in a dialog box as well as a view on a document.

The **CView** class also has a derived class named **CPreviewView**, which is used by the framework to perform print previewing. This class provides support for the features unique to the print-preview window, such as a toolbar, single- or double-page preview, and zooming, that is, enlarging the previewed image. You don't need to call or override any of **CPreviewView**'s member functions unless you want to implement your own interface for print preview (for example, if you want to support editing in print preview mode). See Technical Note 30 in MSVC\HELP\MFCNOTES.HLP for more details on customizing print preview.

include <afxwin.h>

See Also

CWnd, **CFrameWnd**, **CSplitterWnd**, **CDC**, **CDocTemplate**, **CDocument**, **CFormView**, **CEditView**, **CScrollView**

Operations—Public Members

DoPreparePrinting	Displays Print dialog box and creates printer device context; call when overriding the **OnPreparePrinting** member function.
GetDocument	Returns the document associated with the view.

Overridables—Public Members

IsSelected	Tests whether a document item is selected. Required for Object Linking and Embedding (OLE) support.

Constructors—Protected Members

CView	Constructs a **CView** object.

Overridables—Protected Members

OnActivateView	Called when a view is activated.
OnBeginPrinting	Called when a print job begins; override to allocate graphics device interface (GDI) resources.
OnDraw	Called to render an image of the document for screen display, printing, or print preview. Implementation required.
OnEndPrinting	Called when a print job ends; override to deallocate GDI resources.
OnEndPrintPreview	Called when preview mode is exited.
OnInitialUpdate	Called after a view is first attached to a document.
OnPrepareDC	Called before the **OnDraw** member function is called for screen display or the **OnPrint** member function is called for printing or print preview.
OnPreparePrinting	Called before a document is printed or previewed; override to initialize Print dialog box.
OnPrint	Called to print or preview a page of the document.
OnUpdate	Called to notify a view that its document has been modified.

Member Functions

CView::CView

Protected **CView();**♦

Remarks Constructs a **CView** object. The framework calls the constructor when a new frame window is created or a window is split. Override the **OnInitialUpdate** member function to initialize the view after the document is attached.

See Also **CView::OnInitialUpdate**

CView::DoPreparePrinting

BOOL DoPreparePrinting(CPrintInfo* *pInfo* **);**

pInfo Points to a **CPrintInfo** structure that describes the current print job.

Remarks Call this function from your override of **OnPreparePrinting** to invoke the Print dialog box and create a printer device context.

This function's behavior depends on whether it is being called for printing or print preview (specified by the **m_bPreview** member of the *pInfo* parameter). If a file is being printed, this function invokes the Print dialog box, using the values in the **CPrintInfo** structure that *pInfo* points to; after the user has closed the dialog box, the function creates a printer device context based on settings the user specified in the dialog box and returns this device context through the *pInfo* parameter. This device context is used to print the document.

If a file is being previewed, this function creates a printer device context using the current printer settings; this device context is used for simulating the printer during preview.

Return Value Nonzero if printing or print preview can begin; 0 if the operation has been cancelled.

See Also **CPrintInfo, CView::OnPreparePrinting**

CView::GetDocument

CDocument* GetDocument( ) const;

Remarks Call this function to get a pointer to the view's document. This allows you to call the document's member functions.

Return Value A pointer to the **CDocument** object associated with the view. **NULL** if the view is not attached to a document.

See Also **CDocument**

CView::IsSelected

virtual BOOL IsSelected(const CObject* *pDocItem* **) const;**

pDocItem Points to the document item being tested.

Remarks

Called by the framework to check whether the specified document item is selected. The default implementation of this function returns **FALSE**. Override this function if you're implementing selection using **CDocItem** objects. You must override this function if your view contains Object Linking and Embedding (OLE) items. See Chapter 18 in the *Class Library User's Guide* for more information on OLE.

Return Value

Nonzero if the specified document item is selected; otherwise 0.

See Also

CDocItem, **COleClientItem**

CView::OnActivateView

Protected

virtual void OnActivateView(BOOL *bActivate*, **CView*** *pActivateView*, **CView*** *pDeactiveView* **);** ♦

bActivate Indicates whether the view is being activated or deactivated.

pActivateView Points to the view object that is being activated.

pDeactiveView Points to the view object that is being deactivated.

Remarks

Called by the framework when a view is activated or deactivated. The default implementation of this function sets the focus to the view being activated. Override this function if you want to perform special processing when a view is activated or deactivated. For example, if you want to provide special visual cues that distinguish the active view from the inactive views, you would examine the *bActivate* parameter and update the view's appearance accordingly.

The *pActivateView* and *pDeactiveView* parameters point to the same view if the application's main frame window is activated with no change in the active view—for example, if the focus is being transferred from another application to this one, rather than from one view to another within the application. This allows a view to rerealize its palette, if needed.

See Also

CWnd::OnActivate

CView::OnBeginPrinting

Protected **virtual void OnBeginPrinting(CDC*** *pDC*, **CPrintInfo*** *pInfo* **);** ♦

pDC Points to the printer device context.

pInfo Points to a **CPrintInfo** structure that describes the current print job.

Remarks Called by the framework at the beginning of a print or print preview job, after **OnPreparePrinting** has been called. The default implementation of this function does nothing. Override this function to allocate any GDI resources, such as pens or fonts, needed specifically for printing. Select the GDI objects into the device context from within the **OnPrint** member function for each page that uses them. If you are using the same view object to perform both screen display and printing, use separate variables for the GDI resources needed for each display; this allows you to update the screen during printing.

You can also use this function to perform initializations that depend on properties of the printer device context. For example, the number of pages needed to print the document may depend on settings that the user specified from the Print dialog box (such as page length). In such a situation, you cannot specify the document length in the **OnPreparePrinting** member function, where you would normally do so; you must wait until the printer device context has been created based on the dialog box settings. **OnBeginPrinting** is the first overridable function that gives you access to the **CDC** object representing the printer device context, so you can set the document length from this function. Note that if the document length is not specified by this time, a scroll bar is not displayed during print preview.

See Also **CView::OnEndPrinting**, **CView::OnPreparePrinting**, **CView::OnPrint**

CView::OnDraw

Protected **virtual void OnDraw(CDC*** *pDC* **) = 0;** ♦

pDC Points to the device context to be used for rendering an image of the document.

Remarks Called by the framework to render an image of the document. The framework calls this function to perform screen display, printing, and print preview, passing a different device context in each case. There is no default implementation.

You must override this function to display your view on the document. You can make graphic device interface (GDI) calls using the **CDC** object that the *pDC* parameter points to. You can select GDI resources, such as pens or fonts, into the device context before drawing and then deselect them afterwards. Often your drawing code can be device-independent; that is, it doesn't require information about what type of device is displaying the image.

To optimize drawing, you can find out if a given rectangle will be drawn or not by calling the **RectVisible** member function of the device context. If you need to distinguish between normal screen display and printing, call the **IsPrinting** member function of the device context.

See Also **CDC::IsPrinting**, **CDC::RectVisible**, **CView::OnPrint**, **CWnd::OnCreate**, **CWnd::OnDestroy**, **CWnd::PostNcDestroy**

CView::OnEndPrinting

Protected **virtual void OnEndPrinting(CDC*** *pDC*, **CPrintInfo*** *pInfo* **);** ♦

pDC Points to the printer device context.

pInfo Points to a **CPrintInfo** structure that describes the current print job.

Remarks Called by the framework after a document has been printed or previewed. The default implementation of this function does nothing. Override this function to free any GDI resources you allocated in the **OnBeginPrinting** member function.

See Also **CView::OnBeginPrinting**

CView::OnEndPrintPreview

Protected **virtual void OnEndPrintPreview(CDC*** *pDC*, **CPrintInfo*** *pInfo*, **POINT** *point*, **CPreviewView*** *pView* **);** ♦

pDC Points to the printer device context.

pInfo Points to a **CPrintInfo** structure that describes the current print job.

point Specifies the point on the page that was last displayed in preview mode.

pView Points to the view object used for previewing.

Remarks Called by the framework when the user exits print preview mode. The default implementation of this function calls the **OnEndPrinting** member function and restores the main frame window to the state it was in before print preview began.

Override this function to perform special processing when preview mode is terminated. For example, if you want to maintain the user's position in the document when switching from preview mode to normal display mode, you can scroll to the position described by the *point* parameter and the **m_nCurPage** member of the **CPrintInfo** structure that the *pInfo* parameter points to.

Always call the base class version of **OnEndPrinting** from your override, typically at the end of the function.

See Also **CPrintInfo**, **CView::OnEndPrinting**

CView::OnInitialUpdate

Protected **virtual void OnInitialUpdate( );** ♦

Remarks Called by the framework after the view is first attached to the document, but before the view is initially displayed. The default implementation of this function calls the **OnUpdate** member function with no hint information (that is, using the default values of 0 for the *lHint* parameter and **NULL** for the *pHint* parameter). Override this function to perform any one-time initialization that requires information about the document. For example, if your application has fixed-sized documents, you can use this function to initialize a view's scrolling limits based on the document size. If your application supports variable-sized documents, use **OnUpdate** to update the scrolling limits every time the document changes.

See Also **CView::OnUpdate**

CView::OnPrepareDC

Protected **virtual void OnPrepareDC(CDC*** *pDC*, **CPrintInfo*** *pInfo* **= NULL);** ♦

pDC Points to the device context to be used for rendering an image of the document.

pInfo Points to a **CPrintInfo** structure that describes the current print job if **OnPrepareDC** is being called for printing or print preview; the **m_nCurPage** member specifies the page about to be printed. This parameter is **NULL** if **OnPrepareDC** is being called for screen display.

Remarks

Called by the framework before the **OnDraw** member function is called for screen display and before the **OnPrint** member function is called for each page during printing or print preview. The default implementation of this function does nothing if the function is called for screen display. However, this function is overridden in derived classes, such as **CScrollView**, to adjust attributes of the device context; consequently, you should always call the base class implementation at the beginning of your override.

If the function is called for printing, the default implementation examines the page information stored in the *pInfo* parameter. If the length of the document has not been specified, **OnPrepareDC** assumes the document to be one page long and stops the print loop after one page has been printed. The function stops the print loop by setting the **m_bContinuePrinting** member of the structure to **FALSE**.

Override **OnPrepareDC** for any of the following reasons:

- To adjust attributes of the device context as needed for the specified page. For example, if you need to set the mapping mode or other characteristics of the device context, do so in this function.
- To perform print-time pagination. Normally you specify the length of the document when printing begins, using the **OnPreparePrinting** member function. However, if you don't know in advance how long the document is (for example, when printing an undetermined number of records from a database), override **OnPrepareDC** to test for the end of the document while it is being printed. When there is no more of the document to be printed, set the **m_bContinuePrinting** member of the **CPrintInfo** structure to **FALSE**.
- To send escape codes to the printer on a page-by-page basis. To send escape codes from **OnPrepareDC**, call the **Escape** member function of the *pDC* parameter.

Call the base class version of **OnPrepareDC** at the beginning of your override.

See Also

CDC::Escape, **CPrintInfo**, **CView::OnBeginPrinting**, **CView::OnDraw**, **CView::OnPreparePrinting**, **CView::OnPrint**

CView::OnPreparePrinting

Protected

virtual BOOL OnPreparePrinting(CPrintInfo* *pInfo* **);** ♦

pInfo Points to a **CPrintInfo** structure that describes the current print job.

Remarks

Called by the framework before a document is printed or previewed. The default implementation does nothing.

You must override this function to enable printing and print preview. Call the **DoPreparePrinting** member function, passing it the *pInfo* parameter, and then return its return value; **DoPreparePrinting** displays the Print dialog box and creates a printer device context. If you want to initialize the Print dialog box with values other than the defaults, assign values to the members of *pInfo*. For example, if you know the length of the document, pass the value to the **SetMaxPages** member function of *pInfo* before calling **DoPreparePrinting**. This value is displayed in the To: box in the Range portion of the Print dialog box.

DoPreparePrinting does not display the Print dialog box for a preview job. If you want to bypass the Print dialog box for a print job, check that the **m_bPreview** member of *pInfo* is **FALSE** and then set it to **TRUE** before passing it to **DoPreparePrinting**; reset it to **FALSE** afterwards.

If you need to perform initializations that require access to the **CDC** object representing the printer device context (for example, if you need to know the page size before specifying the length of the document), override the **OnBeginPrinting** member function.

Return Value

Nonzero to begin printing; 0 if the print job has been cancelled.

See Also

CPrintInfo, **CView::DoPreparePrinting**, **CView::OnBeginPrinting**, **CView::OnPrepareDC**, **CView::OnPrint**

Example

The following is an override of **OnPreparePrinting** provided by AppWizard if you select the printing option when you create a set of starter files. This override is sufficient unless you want to initialize the Print dialog box.

```
void CMyView::OnPreparePrinting( CPrintInfo *pInfo )
{
    return DoPreparePrinting( pInfo );
}
```

CView::OnPrint

Protected

virtual void OnPrint(CDC* *pDC*, **CPrintInfo*** *pInfo* **);** ♦

pDC Points to the printer device context.

pInfo Points to a **CPrintInfo** structure that describes the current print job.

Remarks

Called by the framework to print or preview a page of the document. For each page being printed, the framework calls this function immediately after calling the **OnPrepareDC** member function. The page being printed is specified by the **m_nCurPage** member of the **CPrintInfo** structure that *pInfo* points to. The default implementation calls the **OnDraw** member function and passes it the printer device context.

Override this function for any of the following reasons:

- To allow printing of multipage documents. Render only the portion of the document that corresponds to the page currently being printed. If you're using **OnDraw** to perform the rendering, you can adjust the viewport origin so that only the appropriate portion of the document is printed.
- To make the printed image look different from the screen image (that is, if your application is not WYSIWYG). Instead of passing the printer device context to **OnDraw**, use the device context to render an image using attributes not shown on the screen.

  If you need GDI resources for printing that you don't use for screen display, select them into the device context before drawing and deselect them afterwards. These GDI resources should be allocated in **OnBeginPrinting** and released in **OnEndPrinting**.
- To implement headers or footers. You can still use **OnDraw** to do the rendering by restricting the area that it can print on.

Note that the **m_rectDraw** member of the *pInfo* parameter describes the printable area of the page in logical units.

Do not call **OnPrepareDC** in your override of **OnPrint**; the framework calls **OnPrepareDC** automatically before calling **OnPrint**.

See Also

CView::OnBeginPrinting, **CView::OnEndPrinting**, **CView::OnPrepareDC**, **CView::OnDraw**

Example

The following is a skeleton for an overridden **OnPrint** function:

```
void CMyView::OnPrint( CDC *pDC, CPrintInfo *pInfo )
{
    // Print headers and/or footers, if desired.
    // Find portion of document corresponding to pInfo->m_nCurPage.
    OnDraw( pDC );
}
```

CView::OnUpdate

Protected

virtual void OnUpdate(CView* *pSender*, **LPARAM** *lHint*, **CObject*** *pHint* **);** ♦

pSender Points to the view that modified the document, or **NULL** if all views are to be updated.

lHint Contains information about the modifications.

pHint Points to an object storing information about the modifications.

Remarks

Called by the framework after the view's document has been modified; this function is called by **CDocument::UpdateAllViews** and allows the view to update its display to reflect those modifications. It is also called by the default implementation of **OnInitialUpdate**. The default implementation invalidates the entire client area, marking it for painting when the next **WM_PAINT** message is received. Override this function if you want to update only those regions that map to the modified portions of the document. To do this you must pass information about the modifications using the hint parameters.

To use *lHint*, define special hint values, typically a bitmask or an enumerated type, and have the document pass one of these values. To use *pHint*, derive a hint class from **CObject** and have the document pass a pointer to a hint object; when overriding **OnUpdate**, use the **CObject::IsKindOf** member function to determine the run-time type of the hint object.

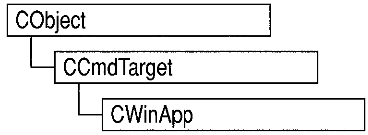
Typically you should not perform any drawing directly from **OnUpdate**. Instead, determine the rectangle describing, in device coordinates, the area that requires updating; pass this rectangle to **CWnd::InvalidateRect**. This causes painting to occur the next time a **WM_PAINT** message is received.

If *lHint* is 0 and *pHint* is **NULL**, the document has sent a generic update notification. If a view receives a generic update notification, or if it cannot decode the hints, it should invalidate its entire client area.

See Also

CDocument::UpdateAllViews, **CView::OnInitialUpdate**, **CWnd::Invalidate**, **CWnd::InvalidateRect**

class CWinApp : public CCmdTarget



The **CWinApp** class is the base class from which you derive a Windows application object. An application object provides member functions for initializing your application (and each instance of it) and for running the application.

Each application that uses the Microsoft Foundation classes can only contain one object derived from **CWinApp**. This object is constructed when other C++ global objects are constructed and is already available when Windows calls the **WinMain** function, which is supplied by the Microsoft Foundation Class Library. Declare your derived **CWinApp** object at the global level.

When you derive an application class from **CWinApp**, override the **InitInstance** member function to create your application's main window object. In addition to the **CWinApp** member functions, the Microsoft Foundation Class Library provides the following global functions to access your **CWinApp** object and other global information:

- **AfxGetApp**   Obtains a pointer to the **CWinApp** object.
- **AfxGetInstanceHandle**   Obtains a handle to the current application instance.
- **AfxGetResourceHandle**   Obtains a handle to the application's resources.
- **AfxGetAppName**   Obtains a pointer to a string containing the application's name. Alternately, if you have a pointer to the **CWinApp** object, use **m_pszExename** to get the application's name.

For more information about these global functions, see "Macros and Globals" in this manual.

See Chapter 2 of this manual for more on the **CWinApp** class, including an overview of:

- **CWinApp**-derived code written by AppWizard.
- **CWinApp**'s role in the execution sequence of your application.
- **CWinApp**'s default member function implementations.
- **CWinApp**'s key overridables.

#include <afxwin.h>

Data Members—Public Members

m_pszAppName	Specifies the name of the application.
m_hInstance	Identifies the current instance of the application.
m_hPrevInstance	Identifies the previous instance of the application.
m_lpCmdLine	Points to a null-terminated string that specifies the command line for the application.
m_nCmdShow	Specifies how the window is to be shown initially.
m_pMainWnd	Holds a pointer to the application's main window. For an example of how to initialize **m_pMainWnd**, see **InitInstance**.
m_bHelpMode	Indicates if the user is in Help context mode (typically invoked with SHIFT+F1).
m_pszExeName	The module name of the application.
m_pszHelpFilePath	The path to the application's Help file.
m_pszProfileName	The application's .INI filename.

Construction/Destruction—Public Members

CWinApp	Constructs a **CWinApp** object.

Operations—Public Members

LoadCursor	Loads a cursor resource.
LoadStandardCursor	Loads a Windows predefined cursor that the **IDC_** constants specify in WINDOWS.H.
LoadOEMCursor	Loads a Windows OEM predefined cursor that the **OCR_** constants specify in WINDOWS.H.
LoadIcon	Loads an icon resource.
LoadStandardIcon	Loads a Windows predefined icon that the **IDI_** constants specify in WINDOWS.H.
LoadOEMIcon	Loads a Windows OEM predefined icon that the **OIC_** constants specify in WINDOWS.H.
LoadVBXFile	Loads a VBX control file.
UnloadVBXFile	Unloads a VBX control file.
GetProfileInt	Retrieves an integer from an entry in the application's .INI file.
WriteProfileInt	Writes an integer to an entry in the application's .INI file.

GetProfileString	Retrieves a string from an entry in the application's .INI file.
WriteProfileString	Writes a string to an entry in the application's .INI file.
AddDocTemplate	Adds a document template to the application's list of available document templates.
OpenDocumentFile	Called by the framework to open a document from a file.
AddToRecentFileList	Adds a filename to the most recently used (MRU) file list.
GetPrinterDeviceDefaults	Retrieves the printer device defaults.

Overridables—Public Members

InitApplication	Override to perform any application-level initialization.
InitInstance	Override to perform Windows instance initialization, such as creating your window objects.
Run	Runs the default message loop. Override to customize the message loop.
OnIdle	Override to perform application-specific idle-time processing.
ExitInstance	Override to clean up when your application terminates.
PreTranslateMessage	Filters messages before they are dispatched to the Windows functions **TranslateMessage** and **DispatchMessage**.
SaveAllModified	Prompts the user to save all modified documents.
DoMessageBox	Implements **AfxMessageBox** for the application.
ProcessMessageFilter	Intercepts certain messages before they reach the application.
ProcessWndProcException	Intercepts all unhandled exceptions thrown by the application's message and command handlers.
DoWaitCursor	Turns the wait cursor on and off.
OnDDECommand	Called by the framework in response to a dynamic data exchange (DDE) execute command.
WinHelp	Calls the **WinHelp** Windows function.

Initialization—Protected Members

LoadStdProfileSettings	Loads standard .INI file settings and enables the MRU file list feature.
SetDialogBkColor	Sets the default background color for dialog boxes and message boxes.
EnableVBX	Enables the use of VBX custom controls in the application.
EnableShellOpen	Allows the user to open data files from the Windows File Manager.
RegisterShellFileTypes	Registers all the application's document types with the Windows File Manager.

Command Handlers—Protected Members

OnFileNew	Implements the **ID_FILE_NEW** command.
OnFileOpen	Implements the **ID_FILE_OPEN** command.
OnFilePrintSetup	Implements the **ID_FILE_PRINT_SETUP** command.
OnContextHelp	Handles Shift+F1 Help within the application.
OnHelp	Handles F1 Help within the application (using the current context).
OnHelpIndex	Handles the **ID_HELP_INDEX** command and provides a default Help topic.
OnHelpUsing	Handles the **ID_HELP_USING** command.

Member Functions

CWinApp::AddDocTemplate

void AddDocTemplate(CDocTemplate* *pTemplate* **);**

pTemplate A pointer to the **CDocTemplate** to be added.

Remarks Call this member function to add a document template to the list of available document templates that the application maintains. You should add all document templates to an application before you call **RegisterShellFileTypes**.

See Also **CWinApp::RegisterShellFileTypes**, **CMultiDocTemplate**, **CSingleDocTemplate**

CWinApp::AddToRecentFileList

virtual void AddToRecentFileList(const char* *pszPathName* **);**

pszPathName The path of the file.

Remarks Call this member function to add *pszPathName* to the MRU file list. You should call the **LoadStdProfileSettings** member function to load the current MRU file list before you use this member function.

The framework calls this member function when it opens a file or executes the Save As command to save a file with a new name.

See Also **CWinApp::LoadStdProfileSettings**

CWinApp::CWinApp

CWinApp(const char* *pszAppName* **= NULL);**

pszAppName A null-terminated string that contains the application name that Windows uses. If this argument is not supplied or is **NULL**, **CWinApp** uses the resource string **AFX_IDS_APP_TITLE** or the filename of the executable file.

Remarks Constructs a **CWinApp** object and passes *pszAppName* to be stored as the application name. You should construct one global object of your **CWinApp**-derived class. You can have only one **CWinApp** object in your application. The constructor stores a pointer to the **CWinApp** object so that **WinMain** can call the object's member functions to initialize and run the application.

CWinApp::DoMessageBox

virtual int DoMessageBox(LPCSTR *lpszPrompt*, **UINT** *nType*,
UINT *nIDPrompt* **);**

lpszPrompt Address of text in the message box.

nType The message box style.

nIDPrompt An index to a Help context string.

Remarks

The framework calls this member function to implement a message box for the global function **AfxMessageBox**. Do not call this member function to open a message box; use **AfxMessageBox** instead.

Override this member function to customize your application-wide processing of **AfxMessageBox** calls.

Return Value

Returns the same values as **AfxMessageBox**.

See Also

AfxMessageBox, **::MessageBox**

CWinApp::DoWaitCursor

virtual void DoWaitCursor(int *nCode* **);**

nCode If this parameter is 0, the original cursor is restored. If 1, a wait cursor appears. If –1, the wait cursor ends.

Remarks

Called by the framework to implement **CCmdTarget::BeginWaitCursor**, **CCmdTarget::EndWaitCursor**, and **CCmdTarget::RestoreWaitCursor**. Implements an hourglass cursor. **DoWaitCursor** maintains a reference count. When positive, the hourglass cursor is displayed.

If your code changes the cursor, call `DoWaitCursor(0)` to restore the cursor to the state the framework is maintaining.

Override this member function to change the wait cursor or to do additional processing while the wait cursor is displayed.

See Also

CCmdTarget::BeginWaitCursor, **CCmdTarget::EndWaitCursor**, **CCmdTarget::RestoreWaitCursor**

CWinApp::EnableShellOpen

Protected **void EnableShellOpen( );** ♦

Remarks Call this function, typically from your **InitInstance** override, to enable your application's users to open data files when they double-click the files from within the Windows File Manager. Call the **RegisterShellFileTypes** member function in conjunction with this function, or provide a .REG file with your application for manual registration of document types.

See Also **CWinApp::OnDDECommand**, **CWinApp::RegisterShellFileTypes**

CWinApp::EnableVBX

Protected **void EnableVBX( );** ♦

Remarks Call this member function from within the **InitInstance** member function to enable the use of VBX controls within your application.

See Also **CWinApp::LoadVBXFile**, **CWinApp::UnloadVBXFile**

CWinApp::ExitInstance

virtual int ExitInstance( );

Remarks Called by the framework from within the **Run** member function to exit this instance of the application. Do not call this member function from anywhere but within the **Run** member function.

The default implementation of this function writes framework options to the application's .INI file. Override this function to clean up when your application terminates.

Return Value The application's exit code; 0 indicates no errors, and values greater than 0 indicate an error. This value is used as the return value from **WinMain**.

See Also **CWinApp::Run**, **CWinApp::InitInstance**

CWinApp::GetPrinterDeviceDefaults

BOOL GetPrinterDeviceDefaults(PRINTDLG FAR* *pPrintDlg* **);**

pPrintDlg A far pointer to a **PRINTDLG** structure.

Remarks Call this member function to prepare a printer device context for printing. Retrieves the current printer defaults from the Windows .INI file as necessary, or uses the last printer configuration set by the user in Print Setup.

Return Value Nonzero if successful; otherwise 0.

See Also **PRINTDLG**, **CPrintDialog**

CWinApp::GetProfileInt

UINT GetProfileInt(LPCSTR *lpszSection*, **LPCSTR** *lpszEntry*, **int** *nDefault* **);**

lpszSection Points to a null-terminated string that specifies the section containing the entry.

lpszEntry Points to a null-terminated string that contains the entry whose value is to be retrieved.

nDefault Specifies the default value to return if the framework cannot find the entry. This value can be an unsigned value in the range 0 through 65,535 or a signed value in the range –32,768 through 32,767.

Remarks Call this member function to retrieve the value of an integer from an entry within a specified section of the application's .INI file.

This member function is not case sensitive, so the strings in the *lpszSection* and *lpszEntry* parameters may differ in case.

Return Value The integer value of the string that follows the specified entry if the function is successful. The return value is the value of the *nDefault* parameter if the function does not find the entry. The return value is 0 if the value that corresponds to the specified entry is not an integer.

Windows 3.1 Only This member function supports hexadecimal notation for the value in the .INI file. When you retrieve a signed integer, you should cast the value into an **int**. ♦

See Also **CWinApp::GetProfileString**, **CWinApp::WriteProfileInt**, **::GetPrivateProfileInt**

CWinApp::GetProfileString

CString GetProfileString(LPCSTR *lpszSection*, **LPCSTR** *lpszEntry*, **LPCSTR** *lpszDefault* = **NULL);**

lpszSection Points to a null-terminated string that specifies the section containing the entry.

lpszEntry Points to a null-terminated string that contains the entry whose string is to be retrieved. This value must not be **NULL**.

lpszDefault Points to the default string value for the given entry if the entry cannot be found in the initialization file.

Remarks

Call this member function to retrieve the string associated with an entry within the specified section in the application's .INI file.

Return Value

The return value is the string from the application's .INI file or *lpszDefault* if the string cannot be found. The maximum string length supported by the framework is **_MAX_PATH**. If *lpszDefault* is **NULL**, the return value is an empty string.

See Also

CWinApp::GetProfileInt, **CWinApp::WriteProfileString**

CWinApp::InitApplication

virtual BOOL InitApplication( );

Remarks

Windows allows several copies of the same program to run at the same time. There are two types of application initialization:

1. One-time application initialization that is done the first time the program runs.
2. Instance initialization that runs each time a copy of the program runs, including the first time.

This function is called by the version of **WinMain** that the framework provides. Override **InitApplication** to implement one-time initialization such as Windows class registration. Override **InitInstance** to implement per-instance initialization.

Return Value

Nonzero if initialization is successful; otherwise 0.

See Also

CWinApp::InitInstance

CWinApp::InitInstance

virtual BOOL InitInstance( );

Remarks

Windows allows several copies of the same program to run at the same time. Application initialization is conceptually divided into two sections: one-time application initialization that is done the first time the program runs, and instance initialization that runs each time a copy of the program runs, including the first time. The framework's implementation of **WinMain** calls this function.

Override **InitInstance** to initialize each new instance of your application running under Windows. Typically, you override **InitInstance** to construct your main window object and set the **m_pMainWnd** data member to point to that window. For more information on overriding this member function, see Chapter 2, "Using the Classes to Write Applications for Windows."

Return Value

Nonzero if initialization is successful; otherwise 0.

See Also

CWinApp::InitApplication

CWinApp::LoadCursor

HCURSOR LoadCursor(LPCSTR *lpszResourceName* **) const;**

HCURSOR LoadCursor(UINT *nIDResource* **) const;**

lpszResourceName Points to a null-terminated string that contains the name of the cursor resource. You can use a **CString** for this argument.

nIDResource ID number of the cursor resource.

Remarks

Loads the cursor resource named by *lpszResourceName* or specified by *nIDResource* from the current executable file. **LoadCursor** loads the cursor into memory only if it has not been previously loaded; otherwise, it retrieves a handle of the existing resource. Use the **LoadStandardCursor** or **LoadOEMCursor** member function to access the predefined Windows cursors.

Return Value

A handle to a cursor. If unsuccessful, returns **NULL**.

See Also

CWinApp::LoadStandardCursor, **CWinApp::LoadOEMCursor**, **::LoadCursor**

CWinApp::LoadIcon

HICON LoadIcon(LPCSTR *lpszResourceName* **) const;**

HICON LoadIcon(UINT *nIDResource* **) const;**

lpszResourceName Points to a null-terminated string that contains the name of the icon resource. You can also use a **CString** for this argument.

nIDResource ID number of the icon resource.

Remarks Loads the icon resource named by *lpszResourceName* or specified by *nIDResource* from the executable file. **LoadIcon** loads the icon only if it has not been previously loaded; otherwise, it retrieves a handle of the existing resource. You can use the **LoadStandardIcon** or **LoadOEMIcon** member function to access the predefined Windows icons.

Return Value A handle to an icon. If unsuccessful, returns **NULL**.

See Also **CWinApp::LoadStandardIcon**, **CWinApp::LoadOEMIcon**, **::LoadIcon**

CWinApp::LoadOEMCursor

HCURSOR LoadOEMCursor(UINT *nIDCursor* **) const;**

nIDCursor An **OCR_** manifest constant identifier that specifies a predefined Windows cursor. You must have **#define OEMRESOURCE** before **#include <afxwin.h>** to gain access to the **OCR_** constants in WINDOWS.H.

Remarks Loads the Windows predefined cursor resource specified by *nIDCursor*. Use the **LoadOEMCursor** or **LoadStandardCursor** member function to access the predefined Windows cursors.

Return Value A handle to a cursor. If unsuccessful, returns **NULL**.

See Also **CWinApp::LoadCursor**, **CWinApp::LoadStandardCursor**, **::LoadCursor**

CWinApp::LoadOEMIcon

HICON LoadOEMIcon(UINT *nIDIcon* **) const;**

nIDIcon An **OIC_** manifest constant identifier that specifies a predefined Windows icon. You must have **#define OEMRESOURCE** before **#include afxwin.h** to access the **OIC_** constants in WINDOWS.H.

Remarks Loads the Windows predefined icon resource specified by *nIDIcon*. Use the **LoadOEMIcon** or **LoadStandardIcon** member function to access the predefined Windows icons.

Return Value A handle to an icon. If unsuccessful, returns **NULL**.

See Also **CWinApp::LoadStandardIcon**, **CWinApp::LoadIcon**, **::LoadIcon**

CWinApp::LoadStandardCursor

HCURSOR LoadStandardCursor(LPCSTR *lpszCursorName* **) const;**

lpszCursorName An **IDC_** manifest constant identifier that specifies a predefined Windows cursor. These identifiers are defined in WINDOWS.H. The following list shows the possible predefined values and meanings for *lpszCursorName*:

- **IDC_ARROW** Standard arrow cursor
- **IDC_IBEAM** Standard text-insertion cursor
- **IDC_WAIT** Hourglass cursor used when Windows performs a time-consuming task
- **IDC_CROSS** Cross-hair cursor for selection
- **IDC_UPARROW** Arrow that points straight up
- **IDC_SIZE** Cursor to use to resize a window
- **IDC_ICON** Cursor to use to drag a file
- **IDC_SIZENWSE** Two-headed arrow with ends at upper left and lower right
- **IDC_SIZENESW** Two-headed arrow with ends at upper right and lower left
- **IDC_SIZEWE** Horizontal two-headed arrow
- **IDC_SIZENS** Vertical two-headed arrow

Remarks Loads the Windows predefined cursor resource that *lpszCursorName* specifies. Use the **LoadStandardCursor** or **LoadOEMCursor** member function to access the predefined Windows cursors.

Return Value A handle to a cursor. If unsuccessful, returns **NULL**.

See Also **CWinApp::LoadOEMCursor**, **CWinApp::LoadCursor**, **::LoadCursor**

CWinApp::LoadStandardIcon

HICON LoadStandardIcon(LPCSTR *lpszIconName* **) const;**

lpszIconName A manifest constant identifier that specifies a predefined Windows icon. These identifiers are defined in WINDOWS.H. The following list shows the possible predefined values and meanings for *lpszIconName*:

- **IDI_APPLICATION** Default application icon
- **IDI_HAND** Hand-shaped icon used in serious warning messages
- **IDI_QUESTION** Question-mark shape used in prompting messages
- **IDI_EXCLAMATION** Exclamation point shape used in warning messages
- **IDI_ASTERISK** Asterisk shape used in informative messages

Remarks Loads the Windows predefined icon resource that *lpszIconName* specifies. Use the **LoadStandardIcon** or **LoadOEMIcon** member function to access the predefined Windows icons.

Return Value A handle to an icon. If unsuccessful, returns **NULL**.

See Also **CWinApp::LoadOEMIcon**, **CWinApp::LoadIcon**, **::LoadIcon**

CWinApp::LoadStdProfileSettings

Protected **void LoadStdProfileSettings( );** ♦

Remarks Call this member function from within the **InitInstance** member function to enable and load the current MRU file list and the last preview state.

See Also **CWinApp::AddToRecentFileList**

CWinApp::LoadVBXFile

HMODULE LoadVBXFile(LPCSTR *lpszFileName* **);**

lpszFileName Points to a null-terminated string that specifies the name of the VBX custom-control dynamic-link library (DLL).

Remarks

Call this member function to load the specified VBX custom-control DLL. Typically, the framework automatically calls this member function to load the proper DLL when a VBX control is created. When the control is destroyed, the framework discards the DLL.

The framework will first attempt to load a VBX file when the corresponding control is created in a dialog box. If the VBX file is not available, the control will not appear in the dialog box, and your application may fail if your code tries to access the missing control.

To verify the existence of a VBX file, call **LoadVBXFile** in your **InitInstance** member function and take appropriate action if the file is missing. If the VBX file exists, call **UnloadVBXFile** to return to the framework's automatic loading and unloading of VBX files.

You may also use **LoadVBXFile** and **UnloadVBXFile** to optimize the performance of frequently used controls. If you call **LoadVBXFile** before a control is created, the framework will no longer load and discard the VBX file each time the control is created and destroyed.

If you call **LoadVBXFile**, it is then your responsibility to call **UnloadVBXFile**, either after the control is destroyed or in the **ExitInstance** member function when your application terminates.

Return Value

The **HMODULE** returned by the **LoadLibrary** Windows function. If an error occurs when loading the VBX custom-control DLL, the return value is an error value less than the constant value **HINSTANCE_ERROR**. If the DLL is not a proper VBX file, or the custom-control DLL could not be initialized, the error value is 14.

See Also

CVBControl, **CWinApp::EnableVBX**, **CWinApp::UnloadVBXFile**, **::LoadLibrary**

CWinApp::OnContextHelp

Protected **afx_msg void OnContextHelp( );** ♦

Remarks You must add an

```
ON_COMMAND( ID_CONTEXT_HELP, OnContextHelp )
```

statement to your **CWinApp** class message map and also add an accelerator table entry, typically SHIFT+F1, to enable this member function.

OnContextHelp puts the application into Help mode. The cursor changes to an arrow and a question mark, and the user can then move the mouse pointer and press the left mouse button to select a dialog box, window, menu, or command button. This member function retrieves the Help context of the object under the cursor and calls the Windows function **WinHelp** with that Help context.

See Also **CWinApp::OnHelp**, **CWinApp::WinHelp**

CWinApp::OnDDECommand

virtual BOOL OnDDECommand(char* *pszCommand* **);**

pszCommand Points to a DDE command string received by the application.

Remarks Called by the framework when the main frame window receives a DDE execute message. The default implementation checks whether the command is a request to open a document and, if so, opens the specified document. The Windows File Manager usually sends such DDE command strings when the user double-clicks a data file. Override this function to handle other DDE execute commands, such as the command to print.

Return Value Nonzero if the command is handled; otherwise 0.

See Also **CWinApp::EnableShellOpen**

CWinApp::OnFileNew

Protected **afx_msg void OnFileNew( );** ♦

Remarks You must add an

```
ON_COMMAND( ID_FILE_NEW, OnFileNew )
```

statement to your **CWinApp** class message map to enable this member function.

If enabled, this function handles execution of the File New command.

See Technical Note 22 in MSVC\HELP\MFCNOTES.HLP for information on default behavior and guidance on how to override this member function.

See Also **CWinApp::OnFileOpen**

CWinApp::OnFileOpen

Protected **afx_msg void OnFileOpen( );** ♦

Remarks You must add an

```
ON_COMMAND( ID_FILE_OPEN, OnFileOpen )
```

statement to your **CWinApp** class message map to enable this member function.

If enabled, this function handles execution of the File Open command.

For information on default behavior and guidance on how to override this member function, see Technical Note 22.

See Also **CWinApp::OnFileNew**

CWinApp::OnFilePrintSetup

Protected **afx_msg void OnFilePrintSetup( );** ♦

Remarks

You must add an

```
ON_COMMAND( ID_FILE_PRINT_SETUP, OnFilePrintSetup )
```

statement to your **CWinApp** class message map to enable this member function.

If enabled, this function handles execution of the File Print command.

For information on default behavior and guidance on how to override this member function, see Technical Note 22.

See Also **CWinApp::OnFileNew**

CWinApp::OnHelp

Protected **afx_msg void OnHelp( );** ♦

Remarks

You must add an

```
ON_COMMAND( ID_ON_HELP, OnHelp )
```

statement to your **CWinApp** class message map to enable this member function. Usually you will also add an accelerator-key entry for the F1 key. Enabling the F1 key is only a convention, not a requirement.

If enabled, called by the framework when the user presses the F1 key.

The default implementation of this message-handler function determines the Help context that corresponds to the current window, dialog box, or menu item and then calls WINHELP.EXE. If no context is currently available, the function uses the default context.

Override this member function to set the Help context to something other than the window, dialog box, menu item, or toolbar button that currently has the focus. Call **WinHelp** with the desired Help context ID.

See Also **CWinApp::OnContextHelp**, **CWinApp::OnHelpUsing**, **CWinApp::OnHelpIndex**, **CWinApp::WinHelp**

CWinApp::OnHelpIndex

Protected **afx_msg void OnHelpIndex( );** ♦

Remarks You must add an

```
ON_COMMAND( ID_HELP_INDEX, OnHelpIndex )
```

statement to your **CWinApp** class message map to enable this member function.

If enabled, the framework calls this message-handler function when the user of your application selects the Help Index command to invoke **WinHelp** with the standard **HELP_INDEX** topic.

See Also **CWinApp::OnHelp**, **CWinApp::OnHelpUsing**, **CWinApp::WinHelp**

CWinApp::OnHelpUsing

Protected **afx_msg void OnHelpUsing( );** ♦

Remarks You must add an

```
ON_COMMAND( ID_HELP_USING, OnHelpUsing )
```

statement to your **CWinApp** class message map to enable this member function.

The framework calls this message-handler function when the user of your application selects the Help Using command to invoke the **WinHelp** application with the standard **HELP_HELPONHELP** topic.

See Also **CWinApp::OnHelp**, **CWinApp::OnHelpIndex**, **CWinApp::WinHelp**

CWinApp::OnIdle

virtual BOOL OnIdle(LONG *lCount* **);**

lCount A counter incremented each time **OnIdle** is called when the application's message queue is empty. This count is reset to 0 each time a new message is processed. You can use the *lCount* parameter to determine the relative length of time the application has been idle without processing a message.

Remarks

Override this member function to perform idle-time processing. **OnIdle** is called in the default message loop when the application's message queue is empty. Use your override to call your own background idle-handler tasks.

OnIdle should return 0 to indicate that no idle processing time is required. The *lCount* parameter is incremented each time **OnIdle** is called when the message queue is empty and resets to 0 each time a new message is processed. You can call your different idle routines based on this count.

The following summarizes idle loop processing:

1. If the message loop in the Microsoft Foundation Class Library checks the message queue and finds no pending messages, it calls `OnIdle` for the application object and supplies 0 as the *lCount* argument.
2. `OnIdle` performs some processing and returns a nonzero value to indicate it should be called again to do further processing.
3. The message loop checks the message queue again. If no messages are pending, it calls `OnIdle` again, incrementing the *lCount* argument.
4. Eventually, `OnIdle` finishes processing all its idle tasks and returns 0. This tells the message loop to stop calling `OnIdle` until the next message is received from the message queue, at which point the idle cycle restarts with the argument set to 0.

Do not perform lengthy tasks during **OnIdle** because your application cannot process user input until **OnIdle** returns.

Note The default implementation of **OnIdle** updates command user-interface objects such as menu items and toolbar buttons, and it performs internal data structure cleanup. Therefore, if you override **OnIdle**, you must call **CWinApp::OnIdle** with the *lCount* in your overridden version. First call all base-class idle processing (that is, until the base class **OnIdle** returns 0). If you need to perform work before the base-class processing completes, review the base-class implementation to select the proper *lCount* during which to do your work.

Return Value

Nonzero to receive more idle processing time; 0 if no more idle time is needed.

Example

The following example shows how to process two idle tasks using the *lCount* argument to prioritize the tasks. The first task is high priority, and you should do it whenever possible. The second task is less important and should be done only when there is a long pause in user input. Note the call to the base-class version of **OnIdle**.

```
BOOL CMyApp::OnIdle(LONG lCount)
{
    BOOL bMore = CWinApp::OnIdle(lCount);

    if (lCount == 0)
    {
    TRACE("App idle for short period of time\n");
    bMore = TRUE;
    }
    else if (lCount == 10)
    {
    TRACE("App idle for longer amount of time\n");
        bMore = TRUE;
    }
    else if (lCount == 100)
    {
        TRACE("App idle for even longer amount of time\n");
        bMore = TRUE;
    }
    else if (lCount == 1000)
    {
        TRACE("App idle for quite a long period of time\n");
     // bMore is not set to TRUE, no longer need idle
     // IMPORTANT: bMore is not set to FALSE since CWinApp::OnIdle may
     // have more idle tasks to complete.
    }

    return bMore;
     // return TRUE as long as there is any more idle tasks
}
```

CWinApp::OpenDocumentFile

virtual CDocument* OpenDocumentFile(LPCSTR *lpszFileName* **);**

lpszFileName The name of the file to be opened.

Remarks

The framework calls this member function to open the named **CDocument** file for the application. If a document with that name is already open, the first frame window that contains that document will be activated. If an application supports multiple document templates, the framework uses file extension to find the appropriate document template to attempt to load the document. If successful, the document template then creates a frame window and view for the document.

Return Value

A pointer to a **CDocument** if successful; otherwise **NULL**.

CWinApp::PreTranslateMessage

virtual BOOL PreTranslateMessage(MSG* *pMsg* **);**

pMsg A pointer to an **MSG** structure that contains the message to process.

Remarks

Override this function to filter window messages before they are dispatched to the Windows functions **TranslateMessage** and **DispatchMessage**. The default implementation performs accelerator-key translation, so you must call the **CWinApp::PreTranslateMessage** member function in your overridden version.

Return Value

Nonzero if the message was fully processed in **PreTranslateMessage** and should not be processed further. Zero if the message should be processed in the normal way.

See Also

::DispatchMessage, **::TranslateMessage**

CWinApp::ProcessMessageFilter

virtual BOOL ProcessMessageFilter(int *code*, **LPMSG** *lpMsg* **);**

code Specifies a hook code. This member function uses the code to determine how to process *lpMsg*.

lpMsg A pointer to a Windows **MSG** structure.

Remarks

The framework's hook function calls this member function to filter and respond to certain Windows messages. A hook function processes events before they are sent to the application's normal message processing.

If you override this advanced feature, be sure to call the base-class version to maintain the framework's hook processing.

Return Value

Nonzero if the message is processed; otherwise 0.

See Also

MessageProc, **WH_MSGFILTER**

CWinApp::ProcessWndProcException

virtual LRESULT ProcessWndProcException(CException* *e***,**
const MSG* *pMsg* **);**

e A pointer to an uncaught exception.

pMsg An **MSG** structure that contains information about the windows message that caused the framework to throw an exception.

Remarks

The framework calls this member function whenever the handler does not catch an exception thrown in one of your application's message or command handlers.

Do not call this member function directly.

The default implementation of this member function creates a message box. If the uncaught exception originates with a menu, toolbar, or accelerator command failure, the message box displays a "Command failed" message; otherwise, it displays an "Internal application error" message.

Override this member function to provide global handling of your exceptions. Only call the base functionality if you wish the message box to be displayed.

Return Value

The value that should be returned to Windows. Normally this is 0L for windows messages, 1L (**TRUE**) for command messages.

See Also

CWnd::WindowProc, CException

CWinApp::RegisterShellFileTypes

Protected

void RegisterShellFileTypes( ); ♦

Remarks

Call this function to register all of your application's document types with the Windows File Manager. This allows the user to open a data file created by your application by double-clicking it from within File Manager. Call this member function after you call **AddDocTemplate** for each of the document templates in your application. Also call the **EnableShellOpen** member function when you call this member function.

This function iterates through the list of **CDocTemplate** objects that the application maintains and, for each document template, adds entries to the registration database that Windows maintains for file associations. File Manager uses these entries to open a data file when the user double-clicks it. This eliminates the need to ship a .REG file with your application.

If the registration database already associates a given filename extension with another file type, no new association is created. See the **CDocTemplate** class for the format of strings necessary to register this information.

See Also **CDocTemplate**, **CWinApp::EnableShellOpen**, **CWinApp::AddDocTemplate**

CWinApp::Run

virtual int Run( );

Remarks Provides a default message loop. **Run** acquires and dispatches Windows messages until the application receives a **WM_QUIT** message. If the application's message queue currently contains no messages, **Run** calls **OnIdle** to perform idle-time processing. Incoming messages go to the **PreTranslateMessage** member function for special processing and then to the Windows function **TranslateMessage** for standard keyboard translation; finally, the **DispatchMessage** Windows function is called. **Run** is rarely overridden, but you can override it to provide special behavior.

Return Value An **int** value that is returned by **WinMain**.

See Also **WM_QUIT**, **::DispatchMessage**, **::TranslateMessage**, **CWinApp::PreTranslateMessage**

CWinApp::SaveAllModified

virtual BOOL SaveAllModified( );

Remarks Called by the framework to save all documents when the application's main frame window is to be closed, or through a **WM_QUERYENDSESSION** message.

The default implementation of this member function calls the **SaveModified** member function in turn for all modified documents within the application.

Return Value Nonzero if safe to terminate the application; 0 if not safe to terminate the application.

CWinApp::SetDialogBkColor

Protected

void SetDialogBkColor(COLORREF *clrCtlBk* **= RGB(192, 192, 192), COLORREF** *clrCtlText* **= RGB(0, 0, 0));** ♦

clrCtlBk The dialog background color for the application.

clrCtlText The dialog control color for the application.

Remarks

Call this member function from within the **InitInstance** member function to set the default background and text color for dialog boxes and message boxes within your application.

CWinApp::UnloadVBXFile

BOOL UnloadVBXFile(LPCSTR *lpszFileName* **);**

lpszFileName Points to a null-terminated string that specifies the name of the VBX custom-control dynamic-link library (DLL).

Remarks

Call this member function to unload the specified VBX custom-control DLL. For more information, see the **LoadVBXFile** member function.

Return Value

Nonzero if successful; otherwise 0.

See Also

CVBControl, **CWinApp::LoadVBXFile**, **CWinApp::EnableVBX**

CWinApp::WinHelp

virtual void WinHelp(DWORD *dwData*, **UINT** *nCmd* **= HELP_CONTEXT);**

dwData Specifies additional data. The value used depends on the value of the *nCmd* parameter.

nCmd Specifies the type of help requested. For a list of possible values and how they affect the *dwData* parameter, see the **WinHelp** Windows function.

Remarks

Call this member function to invoke the WinHelp application. The framework also calls this function to invoke the WinHelp application. The framework will automatically close the WinHelp application when your application terminates.

See Also

CWinApp::OnContextHelp, **CWinApp::OnHelpUsing**, **CWinApp::OnHelp**, **CWinApp::OnHelpIndex**, **::WinHelp**

CWinApp::WriteProfileInt

BOOL WriteProfileInt(LPCSTR *lpszSection*, **LPCSTR** *lpszEntry*, **int** *nValue* **);**

lpszSection Points to a null-terminated string that specifies the section containing the entry. If the section does not exist, it is created. The name of the section is case independent; the string may be any combination of uppercase and lowercase letters.

lpszEntry Points to a null-terminated string that contains the entry into which the value is to be written. If the entry does not exist in the specified section, it is created.

nValue Contains the value to be written.

Remarks

Call this member function to write the specified value into the specified section of the application's .INI file.

Return Value

Nonzero if successful; otherwise 0.

See Also

CWinApp::GetProfileInt, **CWinApp::WriteProfileString**

CWinApp::WriteProfileString

BOOL WriteProfileString(LPCSTR *lpszSection*, **LPCSTR** *lpszEntry*, **LPCSTR** *lpszValue* **);**

lpszSection Points to a null-terminated string that specifies the section containing the entry. If the section does not exist, it is created. The name of the section is case independent; the string may be any combination of uppercase and lowercase letters.

lpszEntry Points to a null-terminated string that contains the entry into which the value is to be written. If the entry does not exist in the specified section, it is created.

lpszValue Points to the string to be written.

Remarks Call this member function to write the specified string into the specified section of the application's .INI file.

Return Value Nonzero if successful; otherwise 0.

See Also **CWinApp::GetProfileString**, **CWinApp::WriteProfileInt**

Data Members

CWinApp::m_bHelpMode

Remarks **TRUE** if the application is in Help context mode (conventionally invoked with SHIFT+F1); otherwise **FALSE**. In Help context mode, the cursor becomes a question mark and the user can move it about the screen. Examine this flag if you want to implement special handling when in the Help mode. **m_bHelpMode** is a public variable of type **BOOL**.

CWinApp::m_hInstance

Remarks Corresponds to the *hInstance* parameter passed by Windows to **WinMain**. The **m_hInstance** data member is a handle to the current instance of the application running under Windows. This is returned by the global function **AfxGetInstanceHandle**. **m_hInstance** is a public variable of type **HINSTANCE**.

CWinApp::m_hPrevInstance

Remarks

Corresponds to the *hPrevInstance* parameter passed by Windows to **WinMain**.

Identifies the previous instance of the application. The **m_hPrevInstance** data member has the value **NULL** if this is the first instance of the application that is running. **m_hPrevInstance** is a public variable of type **HINSTANCE**.

CWinApp::m_lpCmdLine

Remarks

Corresponds to the *lpCmdLine* parameter passed by Windows to **WinMain**. Points to a null-terminated string that specifies the command line for the application. Use **m_lpCmdLine** to access any command-line arguments the user entered when the application was started. **m_lpCmdLine** is a public variable of type **LPSTR**.

CWinApp::m_nCmdShow

Remarks

Corresponds to the *nCmdShow* parameter passed by Windows to **WinMain**. You should pass **m_nCmdShow** as an argument when you call **ShowWindow** for your application's main window. **m_nCmdShow** is a public variable of type **int**.

CWinApp::m_pMainWnd

Remarks

Use this data member to store a pointer to your application's main window object. The Microsoft Foundation Class Library will automatically terminate your application when the window referred to by **m_pMainWnd** is closed. If you don't store a valid **CWnd** pointer here, many default framework implementations will not work correctly. **m_pMainWnd** is a public variable of type **CWnd***.

Typically, you set this member variable when you override **InitInstance**.

See Also

CWinApp::InitInstance

CWinApp::m_pszAppName

Remarks Specifies the name of the application. The application name can come from the parameter passed to the **CWinApp** constructor, or, if not specified, to the resource string with the ID of **AFX_IDS_APP_TITLE**. If the application name is not found in the resource, it comes from the program's .EXE filename. Returned by the global function **AfxGetAppName**. **m_pszAppName** is a public variable of type **const char***.

CWinApp::m_pszExeName

Remarks Contains the name of the application's executable file without an extension. Unlike **m_pszAppName**, this name cannot contain blanks. **m_pszExeName** is a public variable of type **const char***.

CWinApp::m_pszHelpFilePath

Remarks Contains the path to the application's Help file. The framework expects a single Help file, which must have the same name as the application but with a .HLP extension. **m_pszHelpFilePath** is a public variable of type **const char***.

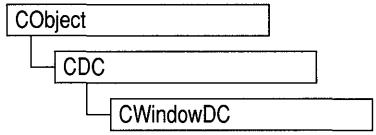
CWinApp::m_pszProfileName

Remarks Contains the name of the application's .INI file. **m_pszProfileName** is a public variable of type **const char***.

See Also **CWinApp::GetProfileString**, **CWinApp::GetProfileInt**, **CWinApp::WriteProfileInt**, **CWinApp::WriteProfileString**

class CWindowDC : public CDC

The **CWindowDC** class is derived from **CDC**. It calls the Windows functions **GetWindowDC** at construction time and **ReleaseDC** at destruction time. This means that a **CWindowDC** object accesses the entire screen area of a **CWnd** (both client and nonclient areas).



#include <afxwin.h>

See Also **CDC**

Construction/Destruction—Public Members

CWindowDC	Constructs a **CWindowDC** object.

Data Members—Protected Members

m_hWnd	The **HWND** to which this **CWindowDC** is attached.

Member Functions

CWindowDC::CWindowDC

CWindowDC(CWnd* *pWnd* **)**
throw(CResourceException);

pWnd The window whose client area the device-context object will access.

Remarks

Constructs a **CWindowDC** object that accesses the entire screen area (both client and nonclient) of the **CWnd** object pointed to by *pWnd*. The constructor calls the Windows function **GetDC**. An exception (of type **CResourceException**) is thrown if the Windows **GetDC** call fails. A device context may not be available if Windows has already allocated all of its available device contexts. Your application competes for the five common display contexts available at any given time under Windows.

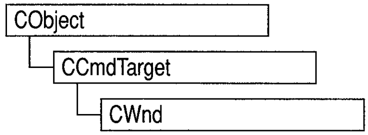
See Also **CDC**, **CClientDC**, **CWnd**

Data Members

CWindowDC::m_hWnd

Remarks The **HWND** of the **CWnd** pointer is used to construct the **CWindowDC** object. **m_hWnd** is a protected variable of type **HWND**.

class CWnd : public CCmdTarget



The **CWnd** class provides the base functionality of all window classes in the Microsoft Foundation Class Library. A **CWnd** object is distinct from a Windows window, but the two are tightly linked. A **CWnd** object is created or destroyed by the **CWnd** constructor and destructor. The Windows window, on the other hand, is a data structure internal to Windows that is created by a **Create** member function and destroyed by the **CWnd** virtual destructor. The **DestroyWindow** function destroys the Windows window without destroying the object. The **CWnd** class and the message-map mechanism hide the **WndProc** function. Incoming Windows notification messages are automatically routed through the message map to the proper **On***Message* **CWnd** member functions. You override an **On***Message* member function to handle a member's particular message in your derived classes.

The **CWnd** class also lets you create a Windows child window for your application. Derive a class from **CWnd**, then add member variables to the derived class to store data specific to your application. Implement message-handler member functions and a message map in the derived class to specify what happens when messages are directed to the window.

You create a child window in two steps. First, call the constructor **CWnd** to construct the **CWnd** object, then call the **Create** member function to create the child window and attach it to the **CWnd** object. When the user terminates your child window, destroy the **CWnd** object, or call the **DestroyWindow** member function to remove the window and destroy its data structures.

Within the Microsoft Foundation Class Library, further classes are derived from **CWnd** to provide specific window types. Many of these classes, including **CFrameWnd**, **CMDIFrameWnd**, **CMDIChildWnd**, **CView**, and **CDialog**, are designed for further derivation. The control classes derived from **CWnd**, such as **CButton**, can be used directly or can be used for further derivation of classes.

#include <afxwin.h>

See Also **CDialog**, **CFrameWnd**, **CView**

Data Members—Public Members

m_hWnd	Indicates the **HWND** attached to this **CWnd**.

Construction/Destruction—Public Members

CWnd	Constructs a **CWnd** object.
DestroyWindow	Destroys the attached Windows window.

Initialization—Public Members

Create	Creates and initializes the child window associated with the **CWnd** object.
PreCreateWindow	Called before the creation of the Windows window attached to this **CWnd** object.
CalcWindowRect	Called to calculate the window rectangle from the client rectangle.
GetStyle	Returns the current window style.
GetExStyle	Returns the window's extended style.
Attach	Attaches a Windows handle to a **CWnd** object.
Detach	Detaches a Windows handle from a **CWnd** object and returns the handle.
SubclassWindow	Attaches a window to a **CWnd** object and makes it route messages through the **CWnd**'s message map.
FromHandle	Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, a temporary **CWnd** object is created and attached.
FromHandlePermanent	Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, **NULL** is returned.
DeleteTempMap	Called automatically by the **CWinApp** idle-time handler and deletes any temporary **CWnd** objects created by **FromHandle**.
GetSafeHwnd	Returns **m_hWnd**, or **NULL** if the **this** pointer is **NULL**.

Window State Functions—Public Members

IsWindowEnabled	Determines if the window is enabled for mouse and keyboard input.
EnableWindow	Enables or disables mouse and keyboard input.
GetActiveWindow	Retrieves the active window.
SetActiveWindow	Activates the window.
GetCapture	Retrieves the **CWnd** that has the mouse capture.
SetCapture	Causes all subsequent mouse input to be sent to the **CWnd**.
GetFocus	Retrieves the **CWnd** that currently has the input focus.
SetFocus	Claims the input focus.
GetDesktopWindow	Retrieves the Windows desktop window.

Window Size and Position—Public Members

GetWindowPlacement	Retrieves the show state and the normal (restored), minimized, and maximized positions of a window.
SetWindowPlacement	Sets the show state and the normal (restored), minimized, and maximized positions for a window.
IsIconic	Determines whether **CWnd** is minimized (iconic).
IsZoomed	Determines whether **CWnd** is maximized.
MoveWindow	Changes the position and/or dimensions of **CWnd**.
SetWindowPos	Changes the size, position, and ordering of child, pop-up, and top-level windows.
ArrangeIconicWindows	Arranges all the minimized (iconic) child windows.
BringWindowToTop	Brings **CWnd** to the top of a stack of overlapping windows.
GetWindowRect	Gets the screen coordinates of **CWnd**.
GetClientRect	Gets the dimensions of the **CWnd** client area.

Window Access Functions—Public Members

ChildWindowFromPoint	Determines which, if any, of the child windows contains the specified point.
FindWindow	Returns the handle of the window, which is identified by its window name and window class.
GetNextWindow	Returns the next (or previous) window in the window manager's list.
GetTopWindow	Returns the first child window that belongs to the **CWnd**.
GetWindow	Returns the window with the specified relationship to this window.
GetLastActivePopup	Determines which pop-up window owned by **CWnd** was most recently active.
IsChild	Indicates whether **CWnd** is a child window or other direct descendant of the specified window.
GetParent	Retrieves the parent window of **CWnd** (if any).
SetParent	Changes the parent window.
WindowFromPoint	Identifies the window that contains the given point.
GetDlgItem	Retrieves the control with the specified ID from the specified dialog box.
GetDlgCtrlID	If the **CWnd** is a child window, calling this function returns its ID value.
GetDescendantWindow	Searches all descendant windows and returns the window with the specified ID.
SendMessageToDescendants	Sends a message to all descendant windows of the window.
GetParentFrame	Returns the **CWnd** object's parent frame window.
UpdateDialogControls	Call to update the state of dialog buttons and other controls.
UpdateData	Initializes or retrieves data from a dialog box.

Update/Painting Functions—Public Members

BeginPaint	Prepares **CWnd** for painting.
EndPaint	Marks the end of painting.
LockWindowUpdate	Disables or reenables drawing in the given window.

GetDC	Retrieves a display context for the client area.
GetDCEx	Retrieves a display context for the client area, and enables clipping while drawing.
RedrawWindow	Updates the specified rectangle or region in the client area.
GetWindowDC	Retrieves the display context for the whole window, including the caption bar, menus, and scroll bars.
ReleaseDC	Releases client and window device contexts, freeing them for use by other applications.
UpdateWindow	Updates the client area.
SetRedraw	Allows changes in **CWnd** to be redrawn or prevents changes from being redrawn.
GetUpdateRect	Retrieves the coordinates of the smallest rectangle that completely encloses the **CWnd** update region.
GetUpdateRgn	Retrieves the **CWnd** update region.
Invalidate	Invalidates the entire client area.
InvalidateRect	Invalidates the client area within the given rectangle by adding that rectangle to the current update region.
InvalidateRgn	Invalidates the client area within the given region by adding that region to the current update region.
ValidateRect	Validates the client area within the given rectangle by removing the rectangle from the current update region.
ValidateRgn	Validates the client area within the given region by removing the region from the current update region.
ShowWindow	Shows or hides the window.
IsWindowVisible	Determines if the window is visible.
ShowOwnedPopups	Shows or hides all pop-up windows owned by the window.
EnableScrollBar	Enables or disables one or both arrows of a scroll bar.

Coordinate Mapping Functions—Public Members

MapWindowPoints	Converts (maps) a set of points from the coordinate space of the **CWnd** to the coordinate space of another window.
ClientToScreen	Converts the client coordinates of a given point or rectangle on the display to screen coordinates.
ScreenToClient	Converts the screen coordinates of a given point or rectangle on the display to client coordinates.

Window Text Functions—Public Members

SetWindowText	Sets the window text or caption title (if it has one) to the specified text.
GetWindowText	Returns the window text or caption title (if it has one).
GetWindowTextLength	Returns the length of the window's text or caption title.
SetFont	Sets the current font.
GetFont	Retrieves the current font.

Scrolling Functions—Public Members

GetScrollPos	Retrieves the current position of a scroll box.
GetScrollRange	Copies the current minimum and maximum scroll-bar positions for the given scroll bar.
ScrollWindow	Scrolls the contents of the client area.
ScrollWindowEx	Scrolls the contents of the client area. Similar to **ScrollWindow**, with additional features.
SetScrollPos	Sets the current position of a scroll box and, if specified, redraws the scroll bar to reflect the new position.
SetScrollRange	Sets minimum and maximum position values for the given scroll bar.
ShowScrollBar	Displays or hides a scroll bar.
EnableScrollBarCtrl	Enables or disables a sibling scroll-bar control.
GetScrollBarCtrl	Returns a sibling scroll-bar control.
RepositionBars	Repositions control bars in the client area.

Drag-Drop Functions—Public Members

DragAcceptFiles	Indicates the window will accept dragged files.

Caret Functions—Public Members

CreateCaret	Creates a new shape for the system caret and gets ownership of the caret.
CreateSolidCaret	Creates a solid block for the system caret and gets ownership of the caret.
CreateGrayCaret	Creates a gray block for the system caret and gets ownership of the caret.
GetCaretPos	Retrieves the client coordinates of the caret's current position.
SetCaretPos	Moves the caret to a specified position.
HideCaret	Hides the caret by removing it from the display screen.
ShowCaret	Shows the caret on the display at the caret's current position. Once shown, the caret begins flashing automatically.

Dialog-Box Item Functions—Public Members

CheckDlgButton	Places a check mark next to or removes a check mark from a button control.
CheckRadioButton	Checks the specified radio button and removes the check mark from all other radio buttons in the specified group of buttons.
GetCheckedRadioButton	Returns the ID of the currently checked radio button in a group of buttons.
DlgDirList	Fills a list box with a file or directory listing.
DlgDirListComboBox	Fills the list box of a combo box with a file or directory listing.
DlgDirSelect	Retrieves the current selection from a list box.
DlgDirSelectComboBox	Retrieves the current selection from the list box of a combo box.
GetDlgItemInt	Translates the text of a control in the given dialog box to an integer value.
GetDlgItemText	Retrieves the caption or text associated with a control.
GetNextDlgGroupItem	Searches for the next (or previous) control within a group of controls.

GetNextDlgTabItem	Retrieves the first control with the **WS_TABSTOP** style that follows (or precedes) the specified control.
IsDlgButtonChecked	Determines whether a button control is checked.
SendDlgItemMessage	Sends a message to the specified control.
SetDlgItemInt	Sets the text of a control to the string that represents an integer value.
SetDlgItemText	Sets the caption or text of a control in the specified dialog box.
SubclassDlgItem	Attaches a Windows control to a **CWnd** object and makes it route messages through the **CWnd**'s message map.

Menu Functions—Public Members

GetMenu	Retrieves a pointer to the specified menu.
SetMenu	Sets the menu to the specified menu.
DrawMenuBar	Redraws the menu bar.
GetSystemMenu	Allows the application to access the Control menu for copying and modification.
HiliteMenuItem	Highlights or removes the highlighting from a top-level (menu-bar) menu item.

Timer Functions—Public Members

SetTimer	Installs a system timer that sends a **WM_TIMER** message when triggered.
KillTimer	Kills a system timer.

Alert Functions—Public Members

FlashWindow	Flashes the window once.
MessageBox	Creates and displays a window that contains an application-supplied message and caption.

Window Message Functions—Public Members

PreTranslateMessage	Used by **CWinApp** to filter window messages before they are dispatched to the **TranslateMessage** and **DispatchMessage** Windows functions.

SendMessage	Sends a message to the **CWnd** object and does not return until it has processed the message.
PostMessage	Places a message in the application queue, then returns without waiting for the window to process the message.

Clipboard Functions—Public Members

ChangeClipboardChain	Removes **CWnd** from the chain of Clipboard viewers.
SetClipboardViewer	Adds **CWnd** to the chain of windows that are notified whenever the contents of the Clipboard are changed.
OpenClipboard	Opens the Clipboard. Other applications will not be able to modify the Clipboard until the Windows **CloseClipboard** function is called.
GetClipboardOwner	Retrieves a pointer to the current owner of the Clipboard.
GetOpenClipboardWindow	Retrieves a pointer to the window that currently has the Clipboard open.
GetClipboardViewer	Retrieves a pointer to the first window in the chain of Clipboard viewers.

Initialization—Protected Members

CreateEx	Creates a Windows overlapped, pop-up, or child window and attaches it to a **CWnd** object.

Operations—Protected Members

GetCurrentMessage	Returns a pointer to the message this window is currently processing. Should only be called when in an **On***Message* message-handler member function.
Default	Calls the default window procedure, which provides default processing for any window messages that an application does not process.

Overridables—Protected Members

GetSuperWndProcAddr	Accesses the default **WndProc** of a subclassed window.
WindowProc	Provides a window procedure for a **CWnd**. The default dispatches messages through the message map.
DefWindowProc	Calls the default window procedure, which provides default processing for any window messages that an application does not process.
PostNcDestroy	This virtual function is called by the default **OnNcDestroy** function after the window has been destroyed.
OnChildNotify	Called by a parent window to give a notifying control a chance to respond to a control notification.
DoDataExchange	For dialog data exchange and validation. Called by **UpdateData**.

Initialization Message Handlers—Protected Members

OnInitMenu	Called when a menu is about to become active.
OnInitMenuPopup	Called when a pop-up menu is about to become active.

System Message Handlers—Protected Members

OnSysChar	Called when a keystroke translates to a system character.
OnSysCommand	Called when the user selects a command from the Control menu, or when the user selects the Maximize or Minimize button.
OnSysDeadChar	Called when a keystroke translates to a system dead character (such as accent characters).
OnSysKeyDown	Called when the user holds down the ALT key and then presses another key.
OnSysKeyUp	Called when the user releases a key that was pressed while the ALT key was held down.
OnCompacting	Called when Windows detects that system memory is low.
OnDevModeChange	Called for all top-level windows when the user changes device-mode settings.

OnFontChange	Called when the pool of font resources changes.
OnPaletteIsChanging	Informs other applications when an application is going to realize its logical palette.
OnPaletteChanged	Called to allow windows that use a color palette to realize their logical palettes and update their client areas.
OnSysColorChange	Called for all top-level windows when a change is made in the system color setting.
OnWindowPosChanging	Called when the size, position, or Z-order is about to change as a result of a call to **SetWindowPos** or another window-management function.
OnWindowPosChanged	Called when the size, position, or Z-order has changed as a result of a call to **SetWindowPos** or another window-management function.
OnDropFiles	Called when the user releases the left mouse button over a window that has registered itself as the recipient of dropped files.
OnSpoolerStatus	Called from Print Manager whenever a job is added to or removed from the Print Manager queue.
OnTimeChange	Called for all top-level windows after the system time changes.
OnWinIniChange	Called for all top-level windows after the Windows initialization file, WIN.INI, is changed.

General Message Handlers—Protected Members

OnCommand	Called when the user selects a command.
OnActivate	Called when **CWnd** is being activated or deactivated.
OnActivateApp	Called when the application is about to be activated or deactivated.
OnCancelMode	Called to allow **CWnd** to cancel any internal modes, such as mouse capture.

OnChildActivate	Called for multiple document interface (MDI) child windows whenever the size or position of **CWnd** changes or **CWnd** is activated.
OnClose	Called as a signal that **CWnd** should be closed.
OnCreate	Called as a part of window creation.
OnCtlColor	Called if **CWnd** is the parent of a control when the control is about to be drawn.
OnDestroy	Called when **CWnd** is being destroyed.
OnEnable	Called when **CWnd** is enabled or disabled.
OnEndSession	Called when the session is ending.
OnEnterIdle	Called to inform an application's main window procedure that a modal dialog box or a menu is entering an idle state.
OnEraseBkgnd	Called when the window background needs erasing.
OnGetMinMaxInfo	Called whenever Windows needs to know the maximized position or dimensions, or the minimum or maximum tracking size.
OnIconEraseBkgnd	Called when **CWnd** is minimized (iconic) and the background of the icon must be filled before painting the icon.
OnKillFocus	Called immediately before **CWnd** loses the input focus.
OnMenuChar	Called when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu.
OnMenuSelect	Called when the user selects a menu item.
OnMove	Called after the position of the **CWnd** has been changed.
OnPaint	Called to repaint a portion of the window.
OnParentNotify	Called when a child window is created or destroyed, or when the user clicks a mouse button while the cursor is over the child window.
OnQueryDragIcon	Called when a minimized (iconic) **CWnd** is about to be dragged by the user.
OnQueryEndSession	Called when the user chooses to end the Windows session.

OnQueryNewPalette	Informs **CWnd** that it is about to receive the input focus.
OnQueryOpen	Called when **CWnd** is an icon and the user requests that the icon be opened.
OnSetFocus	Called after **CWnd** gains the input focus.
OnShowWindow	Called when **CWnd** is to be hidden or shown.
OnSize	Called after the size of **CWnd** has changed.

Control Message Handlers—Protected Members

OnCharToItem	Called by a child list box with the **LBS_WANTKEYBOARDINPUT** style in response to a **WM_CHAR** message.
OnCompareItem	Called to determine the relative position of a new item in a child sorted owner-draw combo box or list box.
OnDeleteItem	Called when an owner-draw child list box or combo box is destroyed or when items are removed from the control.
OnDrawItem	Called when a visual aspect of an owner-draw child button control, combo-box control, list-box control, or menu needs to be drawn.
OnGetDlgCode	Called for a control so the control can process arrow-key and TAB-key input itself.
OnMeasureItem	Called for an owner-draw child combo box, list box, or menu item when the control is created. **CWnd** informs Windows of the dimensions of the control.
OnVKeyToItem	Called by a list box owned by **CWnd** in response to a **WM_KEYDOWN** message.

Input Message Handlers—Protected Members

OnChar	Called when a keystroke translates to a nonsystem character.
OnDeadChar	Called when a keystroke translates to a nonsystem dead character (such as accent characters).
OnHScroll	Called when the user clicks the horizontal scroll bar of **CWnd**.

OnKeyDown	Called when a nonsystem key is pressed.
OnKeyUp	Called when a nonsystem key is released.
OnLButtonDblClk	Called when the user double-clicks the left mouse button.
OnLButtonDown	Called when the user presses the left mouse button.
OnLButtonUp	Called when the user releases the left mouse button.
OnMButtonDblClk	Called when the user double-clicks the middle mouse button.
OnMButtonDown	Called when the user presses the middle mouse button.
OnMButtonUp	Called when the user releases the middle mouse button.
OnMouseActivate	Called when the cursor is in an inactive window and the user presses a mouse button.
OnMouseMove	Called when the mouse cursor moves.
OnRButtonDblClk	Called when the user double-clicks the right mouse button.
OnRButtonDown	Called when the user presses the right mouse button.
OnRButtonUp	Called when the user releases the right mouse button.
OnSetCursor	Called if mouse input is not captured and the mouse causes cursor movement within a window.
OnTimer	Called after each interval specified in **SetTimer**.
OnVScroll	Called when the user clicks the window's vertical scroll bar.

Nonclient-Area Message Handlers—Protected Members

OnNcActivate	Called when the nonclient area needs to be changed to indicate an active or inactive state.
OnNcCalcSize	Called when the size and position of the client area need to be calculated.

OnNcCreate	Called prior to **OnCreate** when the nonclient area is being created.
OnNcDestroy	Called when the nonclient area is being destroyed.
OnNcHitTest	Called by Windows every time the mouse is moved if **CWnd** contains the cursor or has captured mouse input with **SetCapture**.
OnNcLButtonDblClk	Called when the user double-clicks the left mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcLButtonDown	Called when the user presses the left mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcLButtonUp	Called when the user releases the left mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMButtonDblClk	Called when the user double-clicks the middle mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMButtonDown	Called when the user presses the middle mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMButtonUp	Called when the user releases the middle mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcMouseMove	Called when the cursor is moved within a nonclient area of **CWnd**.
OnNcPaint	Called when the nonclient area needs painting.
OnNcRButtonDblClk	Called when the user double-clicks the right mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcRButtonDown	Called when the user presses the right mouse button while the cursor is within a nonclient area of **CWnd**.
OnNcRButtonUp	Called when the user releases the right mouse button while the cursor is within a nonclient area of **CWnd**.

MDI Message Handlers—Protected Members

OnMDIActivate	Called when an MDI child window is activated or deactivated.

Clipboard Message Handlers—Protected Members

OnAskCbFormatName	Called by a Clipboard viewer application when a Clipboard owner will display the Clipboard contents.
OnChangeCbChain	Notifies that a specified window is being removed from the chain.
OnDestroyClipboard	Called when the Clipboard is emptied through a call to the Windows **EmptyClipboard** function.
OnDrawClipboard	Called when the contents of the Clipboard change.
OnHScrollClipboard	Called when a Clipboard owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.
OnPaintClipboard	Called when the client area of the Clipboard viewer needs repainting.
OnRenderAllFormats	Called when the owner application is being destroyed and needs to render all its formats.
OnRenderFormat	Called for the Clipboard owner when a particular format with delayed rendering needs to be rendered.
OnSizeClipboard	Called when the size of the client area of the Clipboard-viewer window has changed.
OnVScrollClipboard	Called when the owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

Member Functions

CWnd::ArrangeIconicWindows

UINT ArrangeIconicWindows( );

Remarks Arranges all the minimized (iconic) child windows. This member function also arranges icons on the desktop window, which covers the entire screen. The **GetDesktopWindow** member function retrieves a pointer to the desktop window object. To arrange iconic MDI child windows in an MDI client window, call **CMDIFrameWnd::MDIIconArrange**.

Return Value The height of one row of icons if the function is successful; otherwise 0.

See Also **CWnd::GetDesktopWindow**, **CMDIFrameWnd::MDIIconArrange**, **::ArrangeIconicWindows**

CWnd::Attach

BOOL Attach(HWND *hWndNew* **);**

hWndNew Specifies a handle to a Windows window.

Remarks Attaches a Windows window to a **CWnd** object.

Return Value Nonzero if successful; otherwise 0.

See Also **CWnd::Detach**, **CWnd::m_hWnd**, **CWnd::SubclassWindow**

CWnd::BeginPaint

CDC* BeginPaint(LPPAINTSTRUCT *lpPaint* **);**

lpPaint Points to the **PAINTSTRUCT** structure that is to receive painting information.

Remarks Prepares **CWnd** for painting and fills a **PAINTSTRUCT** data structure with information about the painting. The paint structure contains a **RECT** data structure that has the smallest rectangle that completely encloses the update region and a flag that specifies whether the background has been erased. The update region is set by the **Invalidate**, **InvalidateRect**, or **InvalidateRgn** member functions and by the system after it sizes, moves, creates, scrolls, or performs any other operation that affects the client area. If the update region is marked for erasing, **BeginPaint** sends an **WM_ONERASEBKGND** message.

Do not call the **BeginPaint** member function except in response to a **WM_PAINT** message. Each call to the **BeginPaint** member function must have a matching call to the **EndPaint** member function. If the caret is in the area to be painted, the **BeginPaint** member function automatically hides the caret to prevent it from being erased.

Return Value

Identifies the device context for **CWnd**. The pointer may be temporary and should not be stored beyond the scope of **EndPaint**.

See Also

CWnd::EndPaint, **CWnd::Invalidate**, **CWnd::InvalidateRgn**, **::BeginPaint**, **CPaintDC**

CWnd::BringWindowToTop

void BringWindowToTop( );

Remarks

Brings **CWnd** to the top of a stack of overlapping windows. In addition, **BringWindowToTop** activates pop-up, top-level, and MDI child windows. The **BringWindowToTop** member function should be used to uncover any window that is partially or completely obscured by any overlapping windows. Calling this function is similar to calling the **SetWindowPos** function to change a window's position in the Z order. The **BringWindowToTop** function does not change the window style to make it a top-level window of the desktop.

See Also

::BringWindowToTop

CWnd::CalcWindowRect

virtual void CalcWindowRect(LPRECT *lpClientRect* **);**

lpClientRect The client rectangle.

Remarks

Call this member function to compute the required size of the window rectangle based on the desired client-rectangle size. The resulting window rectangle (contained in *lpClientRect*) can then be passed to the **Create** member function to create a window whose client area is the desired size.

Called by the framework to size windows prior to creation.

A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window.

See Also **::AdjustWindowRect**

CWnd::ChangeClipboardChain

BOOL ChangeClipboardChain(HWND *hWndNext* **);**

hWndNext Identifies the window that follows **CWnd** in the Clipboard-viewer chain.

Remarks Removes **CWnd** from the chain of Clipboard viewers and makes the window specified by *hWndNext* the descendant of the **CWnd** ancestor in the chain.

Return Value Nonzero if successful; otherwise 0.

See Also **CWnd::SetClipboardViewer**, **::ChangeClipboardChain**

CWnd::CheckDlgButton

void CheckDlgButton(int *nIDButton*, **UINT** *nCheck* **);**

nIDButton Specifies the button to be modified.

nCheck Specifies the action to take. If *nCheck* is nonzero, the **CheckDlgButton** member function places a check mark next to the button; if 0, the check mark is removed. For three-state buttons, if *nCheck* is 2, the button state is indeterminate.

Remarks Selects (places a check mark next to) or clears (removes a check mark from) a button, or it changes the state of a three-state button. The **CheckDlgButton** function sends a **BM_SETCHECK** message to the specified button.

See Also **CWnd::IsDlgButtonChecked**, **CButton::SetCheck**, **::CheckDlgButton**

CWnd::CheckRadioButton

void CheckRadioButton(int *nIDFirstButton*, **int** *nIDLastButton*, **int** *nIDCheckButton* **);**

nIDFirstButton Specifies the integer identifier of the first radio button in the group.

nIDLastButton Specifies the integer identifier of the last radio button in the group.

nIDCheckButton Specifies the integer identifier of the radio button to be checked.

Remarks

Selects (adds a check mark to) a given radio button in a group and clears (removes a check mark from) all other radio buttons in the group. The **CheckRadioButton** function sends a **BM_SETCHECK** message to the specified radio button.

See Also

CWnd::GetCheckedRadioButton, **CButton::SetCheck**, **::CheckRadioButton**

CWnd::ChildWindowFromPoint

CWnd* ChildWindowFromPoint(POINT *point* **) const;**

point Specifies the client coordinates of the point to be tested.

Remarks

Determines which, if any, of the child windows belonging to **CWnd** contains the specified point.

Return Value

Identifies the child window that contains the point. It is **NULL** if the given point lies outside of the client area. If the point is within the client area but is not contained within any child window, **CWnd** is returned.

This member function will return a hidden or disabled child window that contains the specified point. More than one window may contain the given point. However, this function returns only the **CWnd*** of the first window encountered that contains the point. The **CWnd*** that is returned may be temporary and should not be stored for later use.

See Also

CWnd::WindowFromPoint, **::ChildWindowFromPoint**

CWnd::ClientToScreen

void ClientToScreen(LPPOINT *lpPoint* **) const;**

void ClientToScreen(LPRECT *lpRect* **) const;**

lpPoint Points to a **POINT** structure or **CPoint** object that contains the client coordinates to be converted.

lpRect Points to a **RECT** structure or **CRect** object that contains the client coordinates to be converted.

Remarks Converts the client coordinates of a given point or rectangle on the display to screen coordinates. The **ClientToScreen** member function uses the client coordinates in the **POINT** or **RECT** structure or the **CPoint** or **CRect** object pointed to by *lpPoint* or *lpRect* to compute new screen coordinates; it then replaces the coordinates in the structure with the new coordinates. The new screen coordinates are relative to the upper-left corner of the system display. The **ClientToScreen** member function assumes that the given point or rectangle is in client coordinates.

See Also **CWnd::ScreenToClient**, **::ClientToScreen**

CWnd::Create

virtual BOOL Create(LPCSTR *lpszClassName*, **LPCSTR** *lpszWindowName*, **DWORD** *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID*, **CCreateContext*** *pContext* = **NULL);**

lpszClassName Points to a null-terminated character string that names the Windows class (a **WNDCLASS** structure). The class name can be any name registered with the global **AfxRegisterWndClass** function or any of the predefined control-class names. If **NULL**, uses the default **CWnd** attributes.

lpszWindowName Points to a null-terminated character string that contains the window name.

dwStyle Specifies the window style attributes. See below for a description of the possible values.

rect The size and position of the window, in client coordinates of *pParentWnd*.

pParentWnd The parent window.

nID The ID of the child window.

pContext The create context of the window.

Remarks

Creates a Windows child window and attaches it to the **CWnd** object. You construct a child window in two steps. First, invoke the constructor, which constructs the **CWnd** object. Then call **Create**, which creates the Windows child window and attaches it to **CWnd**. **Create** initializes the window's class name and window name and registers values for its style, parent, and ID.

Return Value

Nonzero if successful; otherwise 0.

Window Styles

- **WS_BORDER** Creates a window that has a border.
- **WS_CAPTION** Creates a window that has a title bar (implies the **WS_BORDER** style). This style cannot be used with the **WS_DLGFRAME** style.
- **WS_CHILD** Creates a child window. Cannot be used with the **WS_POPUP** style.
- **WS_CLIPCHILDREN** Excludes the area occupied by child windows when you draw within the parent window. Used when you create the parent window.
- **WS_CLIPSIBLINGS** Clips child windows relative to each other; that is, when a particular child window receives a paint message, the **WS_CLIPSIBLINGS** style clips all other overlapped child windows out of the region of the child window to be updated. (If **WS_CLIPSIBLINGS** is not given and child windows overlap, when you draw within the client area of a child window, it is possible to draw within the client area of a neighboring child window.) For use with the **WS_CHILD** style only.
- **WS_DISABLED** Creates a window that is initially disabled.
- **WS_DLGFRAME** Creates a window with a double border but no title.
- **WS_GROUP** Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined with the **WS_GROUP** style after the first control belong to the same group. The next control with the **WS_GROUP** style ends the style group and starts the next group (that is, one group ends where the next begins).
- **WS_HSCROLL** Creates a window that has a horizontal scroll bar.
- **WS_MAXIMIZE** Creates a window of maximum size.
- **WS_MAXIMIZEBOX** Creates a window that has a Maximize button.
- **WS_MINIMIZE** Creates a window that is initially minimized. For use with the **WS_OVERLAPPED** style only.
- **WS_MINIMIZEBOX** Creates a window that has a Minimize button.

- **WS_OVERLAPPED** Creates an overlapped window. An overlapped window usually has a caption and a border.
- **WS_OVERLAPPEDWINDOW** Creates an overlapped window with the **WS_OVERLAPPED**, **WS_CAPTION**, **WS_SYSMENU**, **WS_THICKFRAME**, **WS_MINIMIZEBOX**, and **WS_MAXIMIZEBOX** styles.
- **WS_POPUP** Creates a pop-up window. Cannot be used with the **WS_CHILD** style.
- **WS_POPUPWINDOW** Creates a pop-up window with the **WS_BORDER**, **WS_POPUP**, and **WS_SYSMENU** styles. The **WS_CAPTION** style must be combined with the **WS_POPUPWINDOW** style to make the Control menu visible.
- **WS_SYSMENU** Creates a window that has a Control-menu box in its title bar. Used only for windows with title bars.
- **WS_TABSTOP** Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the **WS_TABSTOP** style.
- **WS_THICKFRAME** Creates a window with a thick frame that can be used to size the window.
- **WS_VISIBLE** Creates a window that is initially visible.
- **WS_VSCROLL** Creates a window that has a vertical scroll bar.

See Also **CWnd::CWnd**, **CWnd::CreateEx**

CWnd::CreateCaret

void CreateCaret(CBitmap* *pBitmap* **);**

pBitmap Identifies the bitmap that defines the caret shape.

Remarks

Creates a new shape for the system caret and claims ownership of the caret. The bitmap must have previously been created by the **CBitmap::CreateBitmap** member function, the **CreateDIBitmap** Windows function, or the **CBitmap::LoadBitmap** member function. **CreateCaret** automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the **ShowCaret** member function must be called.

The system caret is a shared resource. **CWnd** should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive.

See Also **CBitmap::CreateBitmap**, **::CreateDIBitmap**, **::DestroyCaret**, **CBitmap::LoadBitmap**, **CWnd::ShowCaret**, **::CreateCaret**

CWnd::CreateEx

Protected **BOOL CreateEx(DWORD** *dwExStyle*, **LPCSTR** *lpszClassName*, **LPCSTR** *lpszWindowName*, **DWORD** *dwStyle*, **int** *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **HWND** *hwndParent*, **HMENU** *nIDorHMenu*, **LPSTR** *lpParam* = **NULL);** ♦

dwExStyle Specifies the extended style of the **CWnd** being created. See the "Extended Window Styles" section below for a description of the possible values.

lpszClassName Points to a null-terminated character string that names the Windows class (a **WNDCLASS** structure). The class name can be any name registered with the global **AfxRegisterWndClass** function or any of the predefined control-class names. It must not be **NULL**.

lpszWindowName Points to a null-terminated character string that contains the window name.

dwStyle Specifies the window style attributes. See **CWnd::Create** for a description of the possible values.

x Specifies the initial x-position of the **CWnd** window.

y Specifies the initial top position of the **CWnd** window.

nWidth Specifies the width (in device units) of the **CWnd** window.

nHeight Specifies the height (in device units) of the **CWnd** window.

hwndParent Identifies the parent or owner window of the **CWnd** window being created. Use **NULL** for top-level windows.

nIDorHMenu Identifies a menu or a child-window identifier. The meaning depends on the style of the window.

lpParam Points to a value that is passed to the window through the **CREATESTRUCT** structure.

Remarks

Creates an overlapped, pop-up, or child window with the extended style specified in *dwExStyle*. The **CreateEx** parameters specify the **WNDCLASS**, window title, window style, and (optionally) initial position and size of the window. **CreateEx** also specifies the window's parent (if any) and ID. When **CreateEx** executes, Windows sends the **WM_GETMINMAXINFO**, **WM_NCCREATE**, **WM_NCCALCSIZE**, and **WM_CREATE** messages to the window.

To extend the default message handling, derive a class from **CWnd**, add a message map to the new class, and provide member functions for the above messages. Override **OnCreate**, for example, to perform needed initialization for a new class. Override further **On***Message* message handlers to add further functionality to your derived class.

If the **WS_VISIBLE** style is given, Windows sends the window all the messages required to activate and show the window. If the window style specifies a title bar, the window title pointed to by the *lpszWindowName* parameter is displayed in the title bar. The *dwStyle* parameter can be any combination of window styles.

Return Value

Nonzero if successful; otherwise 0.

Extended Window Styles

- **WS_EX_DLGMODALFRAME** Designates a window with a double border that may (optionally) be created with a title bar when you specify the **WS_CAPTION** style flag in the *dwStyle* parameter.
- **WS_EX_NOPARENTNOTIFY** Specifies that a child window created with this style will not send the **WM_PARENTNOTIFY** message to its parent window when the child window is created or destroyed.

Windows 3.1 Only

- **WS_EX_ACCEPTFILES** Specifies that a window created with this style accepts drag-and-drop files.
- **WS_EX_TOPMOST** Specifies that a window created with this style should be placed above all nontopmost windows and stay above them even when the window is deactivated. An application can use the **SetWindowPos** member function to add or remove this attribute.
- **WS_EX_TRANSPARENT** Specifies that a window created with this style is to be transparent. That is, any windows that are beneath the window are not obscured by the window. A window created with this style receives **WM_PAINT** messages only after all sibling windows beneath it have been updated. ♦

See Also

::CreateWindowEx

CWnd::CreateGrayCaret

void CreateGrayCaret(int *nWidth*, **int** *nHeight* **);**

nWidth Specifies the width of the caret (in logical units). If this parameter is 0, the width is set to the system-defined window-border width.

nHeight Specifies the height of the caret (in logical units). If this parameter is 0, the height is set to the system-defined window-border height.

Remarks

Creates a gray rectangle for the system caret and claims ownership of the caret. The caret shape can be a line or a block. The parameters *nWidth* and *nHeight* specify the caret's width and height (in logical units); the exact width and height (in pixels) depend on the mapping mode. The system's window-border width or height can be retrieved by the **GetSystemMetrics** Windows function with the **SM_CXBORDER** and **SM_CYBORDER** indexes. Using the window-border width or height ensures that the caret will be visible on a high-resolution display.

The **CreateGrayCaret** member function automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the **ShowCaret** member function must be called. The system caret is a shared resource. **CWnd** should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive.

See Also

::DestroyCaret, **::GetSystemMetrics**, **CWnd::ShowCaret**, **::CreateCaret**

CWnd::CreateSolidCaret

void CreateSolidCaret(int *nWidth*, **int** *nHeight* **);**

nWidth Specifies the width of the caret (in logical units). If this parameter is 0, the width is set to the system-defined window-border width.

nHeight Specifies the height of the caret (in logical units). If this parameter is 0, the height is set to the system-defined window-border height.

Remarks

Creates a solid rectangle for the system caret and claims ownership of the caret. The caret shape can be a line or block. The parameters *nWidth* and *nHeight* specify the caret's width and height (in logical units); the exact width and height (in pixels) depend on the mapping mode. The system's window-border width or height can be retrieved by the **GetSystemMetrics** Windows function with the

SM_CXBORDER and **SM_CYBORDER** indexes. Using the window-border width or height ensures that the caret will be visible on a high-resolution display.

The **CreateSolidCaret** member function automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the **ShowCaret** member function must be called. The system caret is a shared resource. **CWnd** should create a caret only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive.

See Also **::DestroyCaret**, **::GetSystemMetrics**, **CWnd::ShowCaret**, **::CreateCaret**

CWnd::CWnd

CWnd( );

Remarks Constructs a **CWnd** object. The Windows window is not created and attached until the **CreateEx** or **Create** member function is called.

See Also **CWnd::CreateEx**, **CWnd::Create**

CWnd::Default

Protected **LRESULT Default( );** ♦

Remarks Calls the default window procedure. The default window procedure provides default processing for any window message that an application does not process. This member function ensures that every message is processed.

Return Value Depends on the message sent.

See Also **CWnd::DefWindowProc**, **::DefWindowProc**

CWnd::DefWindowProc

Protected

virtual LRESULT DefWindowProc(UINT *message*, **WPARAM** *wParam*, **LPARAM** *lParam* **);** ♦

message Specifies the Windows message to be processed.

wParam Specifies additional message-dependent information.

lParam Specifies additional message-dependent information.

Remarks

Calls the default window procedure, which provides default processing for any window message that an application does not process. This member function ensures that every message is processed. It should be called with the same parameters as those received by the window procedure.

Return Value

Depends on the message sent.

See Also

CWnd::Default, **::DefWindowProc**

CWnd::DeleteTempMap

static void PASCAL DeleteTempMap( );

Remarks

Called automatically by the idle time handler of the **CWinApp** object. Deletes any temporary **CWnd** objects created by the **FromHandle** member function.

See Also

CWnd::FromHandle

CWnd::DestroyWindow

virtual BOOL DestroyWindow( );

Remarks

Destroys the Windows window attached to the **CWnd** object. The **DestroyWindow** member function sends appropriate messages to the window to deactivate it and remove the input focus. It also destroys the window's menu, flushes the application queue, destroys outstanding timers, removes Clipboard ownership, and breaks the Clipboard-viewer chain if **CWnd** is at the top of the viewer chain. It sends **WM_DESTROY** and **WM_NCDESTROY** messages to the window. It does not destroy the **CWnd** object.

If the window is the parent of any windows, these child windows are automatically destroyed when the parent window is destroyed. The **DestroyWindow** member function destroys child windows first and then the window itself. The **DestroyWindow** member function also destroys modeless dialog boxes created by **CDialog::Create**.

If the **CWnd** being destroyed is a child window and does not have the **WS_EX_NOPARENTNOTIFY** style set, then the **WM_PARENTNOTIFY** message is sent to the parent.

Return Value Nonzero if the window is destroyed; otherwise 0.

See Also **CWnd::OnDestroy**, **CWnd::Detach**, **::DestroyWindow**

CWnd::Detach

HWND Detach( );

Remarks Detaches a Windows handle from a **CWnd** object and returns the handle.

Return Value A **HWND** to the Windows object.

See Also **CWnd::Attach**

CWnd::DlgDirList

int DlgDirList(LPSTR *lpPathSpec*, **int** *nIDListBox*, **int** *nIDStaticPath*, **UINT** *nFileType* **);**

lpPathSpec Points to a null-terminated string that contains the path or filename. **DlgDirList** modifies this string, which should be long enough to contain the modifications. For more information, see the following "Remarks" section.

nIDListBox Specifies the identifier of a list box. If *nIDListBox* is 0, **DlgDirList** assumes that no list box exists and does not attempt to fill one.

nIDStaticPath Specifies the identifier of the static-text control used to display the current drive and directory. If *nIDStaticPath* is 0, **DlgDirList** assumes that no such text control is present.

nFileType Specifies the attributes of the files to be displayed. It can be any combination of the following values:

- **DDL_READWRITE** Read-write data files with no additional attributes.
- **DDL_READONLY** Read-only files.
- **DDL_HIDDEN** Hidden files.
- **DDL_SYSTEM** System files.
- **DDL_DIRECTORY** Directories.
- **DDL_ARCHIVE** Archives.
- **DDL_POSTMSGS** **LB_DIR** flag. If the **LB_DIR** flag is set, Windows places the messages generated by **DlgDirList** in the application's queue; otherwise, they are sent directly to the dialog-box procedure.
- **DDL_DRIVES** Drives. If the **DDL_DRIVES** flag is set, the **DDL_EXCLUSIVE** flag is set automatically. Therefore, to create a directory listing that includes drives and files, you must call **DlgDirList** twice: once with the **DDL_DRIVES** flag set and once with the flags for the rest of the list.
- **DDL_EXCLUSIVE** Exclusive bit. If the exclusive bit is set, only files of the specified type are listed; otherwise normal files and files of the specified type are listed.

Remarks

Fills a list box with a file or directory listing. **DlgDirList** sends **LB_RESETCONTENT** and **LB_DIR** messages to the list box. It fills the list box specified by *nIDListBox* with the names of all files that match the path given by *lpPathSpec*. The *lpPathSpec* parameter has the following form:

⟦*drive*:⟧ ⟦ ⟦\u⟧*directory*⟦*directory*⟧...\u⟧ ⟦*filename*⟧

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard. The wildcards are a question mark (**?**), which means match any character, and an asterisk (*****), meaning match any number of characters.

If you specify a 0-length string for *lpPathSpec*, or if you specify only a directory name but do not include any file specification, the string will be changed to "*.*". If *lpPathSpec* includes a drive and/or directory name, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by *nIDStaticPath* is also updated with the new drive and/or directory name. After the list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the path.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CWnd::DlgDirListComboBox, **::DlgDirList**

CWnd::DlgDirListComboBox

int DlgDirListComboBox(LPSTR *lpPathSpec*, **int** *nIDComboBox*, **int** *nIDStaticPath*, **UINT** *nFileType* **);**

lpPathSpec Points to a null-terminated string that contains the path or filename. **DlgDirListComboBox** modifies this string, which should be long enough to contain the modifications. For more information, see the following "Remarks" section.

nIDComboBox Specifies the identifier of a combo box in a dialog box. If *nIDComboBox* is 0, **DlgDirListComboBox** assumes that no combo box exists and does not attempt to fill one.

nIDStaticPath Specifies the identifier of the static-text control used to display the current drive and directory. If *nIDStaticPath* is 0, **DlgDirListComboBox** assumes that no such text control is present.

nFileType Specifies DOS file attributes of the files to be displayed. It can be any combination of the following values:

- **DDL_READWRITE** Read-write data files with no additional attributes.
- **DDL_READONLY** Read-only files.
- **DDL_HIDDEN** Hidden files.
- **DDL_SYSTEM** System files.
- **DDL_DIRECTORY** Directories.
- **DDL_ARCHIVE** Archives.
- **DDL_POSTMSGS** **CB_DIR** flag. If the **CB_DIR** flag is set, Windows places the messages generated by **DlgDirListComboBox** in the application's queue; otherwise, they are sent directly to the dialog-box procedure.
- **DDL_DRIVES** Drives. If the **DDL_DRIVES** flag is set, the **DDL_EXCLUSIVE** flag is set automatically. Therefore, to create a directory listing that includes drives and files, you must call **DlgDirListComboBox** twice: once with the **DDL_DRIVES** flag set and once with the flags for the rest of the list.
- **DDL_EXCLUSIVE** Exclusive bit. If the exclusive bit is set, only files of the specified type are listed; otherwise normal files and files of the specified type are listed.

Remarks

Fills the list box of a combo box with a file or directory listing. **DlgDirListComboBox** sends **CB_RESETCONTENT** and **CB_DIR** messages to the combo box. It fills the list box of the combo box specified by *nIDComboBox* with the names of all files that match the path given by *lpPathSpec*. The *lpPathSpec* parameter has the following form:

[[*drive*:]] [[[[\u]]*directory*[[*directory*]]...\u]] [[*filename*]]

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard. The wildcards are a question mark (**?**), which means match any character, and an asterisk (*****), which means match any number of characters.

If you specify a zero-length string for *lpPathSpec*, or if you specify only a directory name but do not include any file specification, the string will be changed to "*.*". If *lpPathSpec* includes a drive and/or directory name, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by *nIDStaticPath* is also updated with the new drive and/or directory name. After the combo-box list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the path.

Return Value

Specifies the outcome of the function. It is nonzero if a listing was made, even an empty listing. A 0 return value implies that the input string did not contain a valid search path.

See Also

CWnd::DlgDirList, **CWnd::DlgDirSelect**, **::DlgDirListComboBox**

CWnd::DlgDirSelect

BOOL DlgDirSelect(LPSTR *lpString*, **int** *nIDListBox* **);**

lpString Points to a buffer that is to receive the current selection in the list box.

nIDListBox Specifies the integer ID of a list box in the dialog box.

Remarks

Retrieves the current selection from a list box. It assumes that the list box has been filled by the **DlgDirList** member function and that the selection is a drive letter, a file, or a directory name. The **DlgDirSelect** member function copies the selection to the buffer given by *lpString*. If there is no selection, *lpString* does not change.

DlgDirSelect sends **LB_GETCURSEL** and **LB_GETTEXT** messages to the list box. It does not allow more than one filename to be returned from a list box. The list box must not be a multiple-selection list box.

Return Value

Nonzero if successful; otherwise 0.

See Also

CWnd::DlgDirList, **CWnd::DlgDirListComboBox**, **CWnd::DlgDirSelectComboBox**, **::DlgDirSelect**

CWnd::DlgDirSelectComboBox

BOOL DlgDirSelectComboBox(LPSTR *lpString*, **int** *nIDComboBox* **);**

lpString Points to a buffer that is to receive the selected path.

nIDComboBox Specifies the integer ID of the combo box in the dialog box.

Remarks

Retrieves the current selection from the list box of a combo box. It assumes that the list box has been filled by the **DlgDirListComboBox** member function and that the selection is a drive letter, a file, or a directory name. The **DlgDirSelectComboBox** member function copies the selection to the specified buffer. If there is no selection, the contents of the buffer are not changed.

DlgDirSelectComboBox sends **CB_GETCURSEL** and **CB_GETLBTEXT** messages to the combo box. It does not allow more than one filename to be returned from a combo box.

Return Value

Nonzero if successful; otherwise 0.

See Also

CWnd::DlgDirListComboBox, **::DlgDirSelectComboBox**

CWnd::DoDataExchange

Protected

virtual void DoDataExchange(CDataExchange* *pDX* **);**

pDX A pointer to a **CDataExchange** object.

Remarks

Called by the framework to exchange and validate dialog data.

Never call this function directly. It is called by the **UpdateData** member function. Call **UpdateData** to initialize a dialog box's controls or retrieve data from a dialog box. When you derive an application-specific dialog class from **CDialog**, you need to override this member function if you wish to utilize the framework's automatic data exchange and validation. ClassWizard will write an overridden version of this member function for you containing the desired "data map" of dialog data exchange (DDX) and validation (DDV) global function calls.

To automatically generate an overridden version of this member function, first create a dialog resource with App Studio, then derive an application-specific dialog class. Then invoke ClassWizard and use it associate variables, data, and validation ranges with various controls in the new dialog box. ClassWizard then writes the overridden **DoDataExchange**, which contains a data map. The following is an example DDX/DDV code block generated by ClassWizard:

```
void CPenWidthsDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CPenWidthsDlg)

        DDX_Text(pDX, IDC_THIN_PEN_WIDTH, m_nThinWidth);

        DDV_MinMaxInt(pDX, m_nThinWidth, 1, 20);

        DDX_Text(pDX, IDC_THICK_PEN_WIDTH, m_nThickWidth);

        DDV_MinMaxInt(pDX, m_nThickWidth, 1, 20);
    //}}AFX_DATA_MAP
}
```

ClassWizard will maintain the code within the //{{ and //}} delimiters. You should not modify this code.

The **DoDataExchange** overridden member function must precede the macro statements in your source file.

For more information on dialog data exchange and validation, see Chapter 7 of the *Class Library User's Guide*, or see Chapter 9 of the *App Studio User's Guide*. For a description of the DDX_ and DDV_ macros generated by ClassWizard, see Technical Note 26 in MSVC\HELP\MFCNOTES.HLP.

See Also **CWnd::UpdateData**

CWnd::DragAcceptFiles

Windows 3.1 Only **void DragAcceptFiles(BOOL** *bAccept* = **TRUE);** ♦

bAccept Flag that indicates whether dragged files are accepted.

Remarks Call this member function from within the main window in your application's **CWinApp::InitInstance** function to indicate that your main window and all child windows accept dropped files from the Windows File Manager.

To discontinue receiving dragged files, call the member function with *bAccept* equal to **FALSE**.

See Also **::DragAcceptFiles, WM_DROPFILES**

CWnd::DrawMenuBar

void DrawMenuBar( );

Remarks Redraws the menu bar. If a menu bar is changed after Windows has created the window, call this function to draw the changed menu bar.

See Also **::DrawMenuBar**

CWnd::EnableScrollBar

Windows 3.1 Only **BOOL EnableScrollBar(int** *nSBFlags*, **UINT** *nArrowFlags* = **ESB_ENABLE_BOTH);** ♦

nSBFlags Specifies the scroll-bar type. Can have one of the following values:

- **SB_BOTH** Enables or disables the arrows of the horizontal and vertical scroll bars associated with the window.
- **SB_HORZ** Enables or disables the arrows of the horizontal scroll bar associated with the window.
- **SB_VERT** Enables or disables the arrows of the vertical scroll bar associated with the window.

nArrowFlags Specifies whether the scroll-bar arrows are enabled or disabled and which arrows are enabled or disabled. Can have one of the following values:

- **ESB_ENABLE_BOTH** Enables both arrows of a scroll bar (default).
- **ESB_DISABLE_LTUP** Disables the left arrow of a horizontal scroll bar or the up arrow of a vertical scroll bar.
- **ESB_DISABLE_RTDN** Disables the right arrow of a horizontal scroll bar or the down arrow of a vertical scroll bar.
- **ESB_DISABLE_BOTH** Disables both arrows of a scroll bar.

Remarks Enables or disables one or both arrows of a scroll bar.

Return Value Nonzero if the arrows are enabled or disabled as specified. Otherwise it is 0, which indicates that the arrows are already in the requested state or that an error occurred.

See Also **CWnd::ShowScrollBar**, **CScrollBar::EnableScrollBar**

CWnd::EnableScrollBarCtrl

void EnableScrollBarCtrl(int *nBar*, **BOOL** *bEnable* = **TRUE);**

nBar The scroll-bar identifier.

bEnable Specifies whether the scroll-bar is to be enabled or disabled.

Remarks Call this member function to enable or disable the scroll bar for this window. If the window has a sibling scroll-bar control, that scroll bar is used; otherwise the window's own scroll bar is used.

See Also **CWnd::GetScrollBarCtrl**

CWnd::EnableWindow

BOOL EnableWindow(BOOL *bEnable* = **TRUE);**

bEnable Specifies whether the given window is to be enabled or disabled. If this parameter is **TRUE**, the window will be enabled. If this parameter is **FALSE**, the window will be disabled.

Remarks

Enables or disables mouse and keyboard input. When input is disabled, input such as mouse clicks and keystrokes is ignored. When input is enabled, the window processes all input. If the enabled state is changing, the **WM_ENABLE** message is sent before this function returns. If disabled, all child windows are implicitly disabled, although they are not sent **WM_ENABLE** messages.

A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the main window must be enabled before the dialog box is destroyed. Otherwise, another window will get the input focus and be activated. If a child window is disabled, it is ignored when Windows tries to determine which window should get mouse messages. By default, a window is enabled when it is created. An application can specify the **WS_DISABLED** style in the **Create** or **CreateEx** member function to create a window that is initially disabled. After a window has been created, an application can also use the **EnableWindow** member function to enable or disable the window. An application can use this function to enable or disable a control in a dialog box. A disabled control cannot receive the input focus nor can a user access it.

Return Value

Indicates the state before the **EnableWindow** member function was called. The return value is nonzero if the window was previously disabled. The return value is 0 if the window was previously enabled or an error occurred.

See Also

::EnableWindow, **CWnd::OnEnable**

CWnd::EndPaint

void EndPaint(LPPAINTSTRUCT *lpPaint* **);**

lpPaint Points to a **PAINTSTRUCT** structure that contains the painting information retrieved by the **BeginPaint** member function.

Remarks

Marks the end of painting in the given window. The **EndPaint** member function is required for each call to the **BeginPaint** member function, but only after painting is complete. If the caret was hidden by the **BeginPaint** member function, **EndPaint** restores the caret to the screen.

See Also

CWnd::BeginPaint, **::EndPaint**, **CPaintDC**

CWnd::FindWindow

static CWnd* PASCAL FindWindow(LPCSTR *lpszClassName*, **LPCSTR** *lpszWindowName* **);**

lpszClassName Points to a null-terminated string that specifies the window's class name (a **WNDCLASS** structure). If *lpszClassName* is **NULL**, all class names match.

lpszWindowName Points to a null-terminated string that specifies the window name (the window's title). If *lpszWindowName* is **NULL**, all window names match.

Remarks

Returns the top-level **CWnd** whose window class is given by *lpszClassName* and whose window name, or title, is given by *lpszWindowName*. This function does not search child windows.

Return Value

Identifies the window that has the specified class name and window name. It is **NULL** if no such window is found. The **CWnd*** may be temporary and should not be stored for later use.

See Also

::FindWindow

CWnd::FlashWindow

BOOL FlashWindow(BOOL *bInvert* **);**

bInvert Specifies whether the **CWnd** is to be flashed or returned to its original state. The **CWnd** is flashed from one state to the other if *bInvert* is **TRUE**. If *bInvert* is **FALSE**, the window is returned to its original state (either active or inactive).

Remarks

Flashes the given window once. For successive flashing, create a system timer and repeatedly call **FlashWindow**. Flashing the **CWnd** means changing the appearance of its title bar as if the **CWnd** were changing from inactive to active status, or vice versa. (An inactive title bar changes to an active title bar; an active title bar changes to an inactive title bar.) Typically, a window is flashed to inform the user that it requires attention but that it does not currently have the input focus.

The *bInvert* parameter should be **FALSE** only when the window is getting the input focus and will no longer be flashing; it should be **TRUE** on successive calls while waiting to get the input focus. This function always returns nonzero for minimized windows. If the window is minimized, **FlashWindow** will simply flash the window's icon; *bInvert* is ignored for minimized windows.

Return Value
Nonzero if the window was active before the call to the **FlashWindow** member function; otherwise 0.

See Also
::FlashWindow

CWnd::FromHandle

static CWnd* PASCAL FromHandle(HWND *hWnd* **);**

hWnd An **HWND** of a Windows window.

Return Value
Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, a temporary **CWnd** object is created and attached. The pointer may be temporary and shouldn't be stored beyond immediate use.

See Also
CWnd::DeleteTempMap

CWnd::FromHandlePermanent

static CWnd* PASCAL FromHandlePermanent(HWND *hWnd* **);**

hWnd An **HWND** of a Windows window.

Remarks
Returns a pointer to a **CWnd** object when given a handle to a window. If a **CWnd** object is not attached to the handle, **NULL** is returned.

This function, unlike **FromHandle**, does not create temporary objects.

Return Value
A pointer to a **CWnd** object.

See Also
CWnd::FromHandle

CWnd::GetActiveWindow

static CWnd* PASCAL GetActiveWindow( );

Remarks Retrieves a pointer to the active window. The active window is either the window that has the current input focus or the window explicitly made active by the **SetActiveWindow** member function.

Return Value The active window or **NULL** if no window was active at the time of the call. The pointer may be temporary and should not be stored for later use.

See Also **CWnd::SetActiveWindow, ::GetActiveWindow**

CWnd::GetCapture

static CWnd* PASCAL GetCapture( );

Remarks Retrieves the window that has the mouse capture. Only one window has the mouse capture at any given time. A window receives the mouse capture when the **SetCapture** member function is called. This window receives mouse input whether or not the cursor is within its borders.

Return Value Identifies the window that has the mouse capture. It is **NULL** if no window has the mouse capture. The return value may be temporary and should not be stored for later use.

See Also **CWnd::SetCapture, ::GetCapture**

CWnd::GetCaretPos

static CPoint PASCAL GetCaretPos( );

Remarks Retrieves the client coordinates of the caret's current position and returns them as a **CPoint**. The caret position is given in the client coordinates of the **CWnd** window.

Return Value **CPoint** object containing the coordinates of the caret's position.

See Also **::GetCaretPos**

CWnd::GetCheckedRadioButton

int GetCheckedRadioButton(int *nIDFirstButton*, **int** *nIDLastButton* **);**

nIDFirstButton Specifies the integer identifier of the first radio button in the group.

nIDLastButton Specifies the integer identifier of the last radio button in the group.

Remarks Retrieves the ID of the currently checked radio button in the specified group.

Return Value ID of the checked radio button, or 0 if none is selected.

See Also **CWnd::CheckRadioButton**

CWnd::GetClientRect

void GetClientRect(LPRECT *lpRect* **) const;**

lpRect Points to a **RECT** structure or a **CRect** object to receive the client coordinates. The **left** and **top** members will be 0. The **right** and **bottom** members will contain the width and height of the window.

Remarks Copies the client coordinates of the **CWnd** client area into the structure pointed to by *lpRect*. The client coordinates specify the upper-left and lower-right corners of the client area. Since client coordinates are relative to the upper-left corners of the **CWnd** client area, the coordinates of the upper-left corner are (0,0).

See Also **CWnd::GetWindowRect**, **::GetClientRect**

CWnd::GetClipboardOwner

static CWnd* PASCAL GetClipboardOwner( );

Remarks Retrieves the current owner of the Clipboard. The Clipboard can still contain data even if it is not currently owned.

Return Value Identifies the window that owns the Clipboard if successful; otherwise, **NULL**. The returned pointer may be temporary and shouldn't be stored for later use.

See Also **CWnd::GetClipboardViewer**, **::GetClipboardOwner**

CWnd::GetClipboardViewer

static CWnd* PASCAL GetClipboardViewer( );

Remarks Retrieves the first window in the Clipboard-viewer chain.

Return Value Identifies the window currently responsible for displaying the Clipboard if successful; otherwise **NULL** (for example, if there is no viewer). The returned pointer may be temporary and should not be stored for later use.

See Also **CWnd::GetClipboardOwner**, **::GetClipboardViewer**

Return Value Returns a pointer to the message the window is currently processing. Should only be called when in an **On***Message* handler.

CWnd::GetDC

CDC* GetDC( );

Remarks Retrieves a pointer to a common, class, or private device context for the client area depending on the class style specified for the **CWnd**. For common device contexts, **GetDC** assigns default attributes to the context each time it is retrieved. For class and private contexts, **GetDC** leaves the previously assigned attributes unchanged. The device context can be used in subsequent graphics device interface (GDI) functions to draw in the client area.

Unless the device context belongs to a window class, the **ReleaseDC** member function must be called to release the context after painting. Since only five common device contexts are available at any given time, failure to release a device context can prevent other applications from accessing a device context. A device context belonging to the **CWnd** class is returned by the **GetDC** member function if **CS_CLASSDC**, **CS_OWNDC**, or **CS_PARENTDC** was specified as a style in the **WNDCLASS** structure when the class was registered.

Return Value Identifies the device context for the **CWnd** client area if successful; otherwise, the return value is **NULL**. The pointer may be temporary and should not be stored for later use.

See Also **CWnd::ReleaseDC**, **::GetDC**, **CClientDC**

CWnd::GetDCEx

Windows 3.1 Only **CDC* GetDCEx(CRgn*** *prgnClip*, **DWORD** *flags* **);** ♦

prgnClip Identifies a clipping region that may be combined with the visible region of the client window.

flags Can have one of the following preset values:

- **DCX_CACHE** Returns a device context from the cache rather than the **OWNDC** or **CLASSDC** window. Overrides **CS_OWNDC** and **CS_CLASSDC**.
- **DCX_CLIPCHILDREN** Excludes the visible regions of all child windows below the **CWnd** window.
- **DCX_CLIPSIBLINGS** Excludes the visible regions of all sibling windows above the **CWnd** window.
- **DCX_EXCLUDERGN** Excludes the clipping region identified by *prgnClip* from the visible region of the returned device context.
- **DCX_INTERSECTRGN** Intersects the clipping region identified by *prgnClip* within the visible region of the returned device context.
- **DCX_LOCKWINDOWUPDATE** Allows drawing even if there is a **LockWindowUpdate** call in effect that would otherwise exclude this window. This value is used for drawing during tracking.
- **DCX_PARENTCLIP** Uses the visible region of the parent window and ignores the parent window's **WS_CLIPCHILDREN** and **WS_PARENTDC** style bits. This value sets the device context's origin to the upper-left corner of the **CWnd** window.
- **DCX_WINDOW** Returns a device context that corresponds to the window rectangle rather than the client rectangle.

Remarks Retrieves the handle of a device context for the **CWnd** window. The device context can be used in subsequent GDI functions to draw in the client area. This function, which is an extension to the **GetDC** function, gives an application more control over how and whether a device context for a window is clipped. Unless the device context belongs to a window class, the **ReleaseDC** function must be called to release the context after drawing. Since only five common device contexts are available at any given time, failure to release a device context can prevent other applications from gaining access to a device context.

In order to obtain a cached device context, an application must specify **DCX_CACHE**. If **DCX_CACHE** is not specified and the window is neither **CS_OWNDC** nor **CS_CLASSDC**, this function returns **NULL**. A device context with special characteristics is returned by the **GetDCEx** function if the **CS_CLASSDC**, **CS_OWNDC**, or **CS_PARENTDC** style was specified in the **WNDCLASS** structure when the class was registered. For more information about these characteristics, see the description of the **WNDCLASS** structure in the *Windows Programmer's Reference, Volume 3*.

Return Value The device context for the specified window if the function is successful; otherwise **NULL**.

See Also **CWnd::BeginPaint**, **CWnd::GetDC**, **CWnd::GetWindowDC**, **CWnd::ReleaseDC**, **::GetDCEx**

CWnd::GetDescendantWindow

CWnd* GetDescendantWindow(int *nID* **) const;**

nID Specifies the identifier of the control or child window to be retrieved.

Remarks Call this member function to find the descendant window specified by the given ID. This member function searches the entire tree of child windows, not just those that are immediate children.

Return Value A pointer to a **CWnd** object, or **NULL** if no child window is found.

See Also **CWnd::GetParentFrame**, **CWnd::IsChild**, **CWnd::GetDlgItem**

CWnd::GetDesktopWindow

static CWnd* PASCAL GetDesktopWindow( );

Remarks Returns the Windows desktop window. The desktop window covers the entire screen and is the area on top of which all icons and other windows are painted.

Return Value Identifies the Windows desktop window. This pointer may be temporary and should not be stored for later use.

See Also **::GetDesktopWindow**

CWnd::GetDlgCtrlID

int GetDlgCtrlID( ) const;

Remarks Returns the window or control ID value for any child window, not just that of a control in a dialog box. Since top-level windows do not have an ID value, the return value of this function is invalid if the **CWnd** is a top-level window.

Return Value The numeric identifier of the **CWnd** child window if the function is successful; otherwise 0.

See Also **::GetDlgCtrlID**

CWnd::GetDlgItem

CWnd* GetDlgItem(int *nID* **) const;**

nID Specifies the identifier of the control or child window to be retrieved.

Remarks Retrieves a pointer to the specified control or child window in a dialog box or other window. The pointer returned is usually cast to the type of control identified by *nID*.

Return Value A pointer to the given control or child window. If no control with the integer ID given by the *nID* parameter exists, the value is **NULL**. The returned pointer may be temporary and should not be stored.

See Also **CWnd::Create**, **CWnd::GetWindow**, **CWnd::GetDescendantWindow**, **CWnd::GetWindow**, **::GetDlgItem**

CWnd::GetDlgItemInt

UINT GetDlgItemInt(int *nID*, **BOOL*** *lpTrans* **= NULL,**
BOOL *bSigned* **= TRUE) const;**

nID Specifies the integer identifier of the dialog-box control to be translated.

lpTrans Points to the Boolean variable that is to receive the translated flag.

bSigned Specifies whether the value to be retrieved is signed.

Remarks

Retrieves the text of the control identified by *nID*. It translates the text of the specified control in the given dialog box into an integer value by stripping any extra spaces at the beginning of the text and converting decimal digits. It stops the translation when it reaches the end of the text or encounters any nonnumeric character.

If *bSigned* is **TRUE**, **GetDlgItemInt** checks for a minus sign (–) at the beginning of the text and translates the text into a signed number. Otherwise, it creates an unsigned value. It sends a **WM_GETTEXT** message to the control.

Return Value

Specifies the translated value of the dialog-box item text. Since 0 is a valid return value, *lpTrans* must be used to detect errors. If a signed return value is desired, cast it as an **int** type. The function returns 0 if the translated number is greater than 32,767 (for signed numbers) or 65,535 (for unsigned).

When errors occur, such as encountering nonnumeric characters and exceeding the above maximum, **GetDlgItemInt** copies 0 to the location pointed to by *lpTrans*. If there are no errors, *lpTrans* receives a nonzero value. If *lpTrans* is **NULL**, **GetDlgItemInt** does not warn about errors.

See Also

CWnd::GetDlgItemText, **::GetDlgItemInt**

CWnd::GetDlgItemText

int GetDlgItemText(int *nID*, **LPSTR** *lpStr*, **int** *nMaxCount* **) const;**

nID Specifies the integer identifier of the control whose title is to be retrieved.

lpStr Points to the buffer to receive the control's title or text.

nMaxCount Specifies the maximum length (in bytes) of the string to be copied to *lpStr*. If the string is longer than *nMaxCount*, it is truncated.

Remarks

Retrieves the title or text associated with a control in a dialog box. The **GetDlgItemText** member function copies the text to the location pointed to by *lpStr* and returns a count of the number of bytes it copies.

Return Value

Specifies the actual number of bytes copied to the buffer, not including the terminating null character. The value is 0 if no text is copied.

See Also

CWnd::GetDlgItem, **CWnd::GetDlgItemInt**, **::GetDlgItemText**, **WM_GETTEXT**

CWnd::GetExStyle

DWORD GetExStyle( ) const;

Return Value The window's extended style.

See Also **CWnd::GetStyle**, **::GetExStyle**, **::GetWindowLong**

CWnd::GetFocus

static CWnd* PASCAL GetFocus( );

Remarks Retrieves a pointer to the **CWnd** that currently has the input focus.

Return Value A pointer to the window that has the current focus, or **NULL** if there is no focus window. The pointer may be temporary and should not be stored for later use.

See Also **CWnd::GetActiveWindow**, **CWnd::GetCapture**, **CWnd::SetFocus**, **::GetFocus**

CWnd::GetFont

CFont* GetFont( ) const;

Remarks Gets the current font for this window.

Return Value A pointer to the current font. The pointer may be temporary and should not be stored for later use.

See Also **CWnd::SetFont**, **WM_GETFONT**, **CFont**

CWnd::GetLastActivePopup

CWnd* GetLastActivePopup( ) const;

Remarks This function determines which pop-up window owned by **CWnd** was most recently active.

Return Value Identifies the most recently active pop-up window. The return value will be the window itself if any of the following conditions are met:

- The window itself was most recently active
- The window does not own any pop-up windows
- The window is not a top-level window or is owned by another window

The pointer may be temporary and should not be stored for later use.

See Also **::GetLastActivePopup**

CWnd::GetMenu

CMenu* GetMenu( ) const;

Remarks Retrieves a pointer to the menu for this window. This function should not be used for child windows because they do not have a menu.

Return Value Identifies the menu. The value is **NULL** if **CWnd** has no menu. The return value is undefined if **CWnd** is a child window. The returned pointer may be temporary and should not be stored for later use.

See Also **::GetMenu**

CWnd::GetNextDlgGroupItem

CWnd* GetNextDlgGroupItem(CWnd* *pWndCtl*, **BOOL** *bPrevious* = **FALSE) const;**

pWndCtl Identifies the control to be used as the starting point for the search.

bPrevious Specifies how the function is to search the group of controls in the dialog box. If **TRUE**, the function searches for the previous control in the group; if **FALSE**, it searches for the next control in the group.

Remarks Searches for the previous (or next) control within a group of controls in a dialog box. A group of controls begins with a control that was created with the **WS_GROUP** style and ends with the last control that was not created with the **WS_GROUP** style. By default, the **GetNextDlgGroupItem** member function returns a pointer to the next control in the group. If *pWndCtl* identifies the first control in the group and *bPrevious* is **TRUE**, **GetNextDlgGroupItem** returns a pointer to the last control in the group.

Return Value

Pointer to the previous (or next) control in the group if the member function is successful. The returned pointer may be temporary and should not be stored for later use.

See Also

CWnd::GetNextDlgTabItem, **::GetNextDlgGroupItem**

CWnd::GetNextDlgTabItem

CWnd* GetNextDlgTabItem(CWnd* *pWndCtl*, **BOOL** *bPrevious* = **FALSE)**
const;

pWndCtl Identifies the control to be used as the starting point for the search.

bPrevious Specifies how the function is to search the dialog box. If **TRUE**, the function searches for the previous control in the dialog box; if **FALSE**, it searches for the next control.

Remarks

Retrieves a pointer to the first control that was created with the **WS_TABSTOP** style and that precedes (or follows) the specified control.

Return Value

Pointer to the previous (or next) control that has the **WS_TABSTOP** style, if the member function is successful. The returned pointer may be temporary and should not be stored for later use.

See Also

CWnd::GetNextDlgGroupItem, **::GetNextDlgTabItem**

CWnd::GetNextWindow

CWnd* GetNextWindow(UINT *nFlag* = **GW_HWNDNEXT) const;**

nFlag Specifies whether the function returns a pointer to the next window or the previous window. It can be either **GW_HWNDNEXT**, which returns the window that follows the **CWnd** object on the window manager's list, or **GW_HWNDPREV**, which returns the previous window on the window manager's list.

Remarks Searches for the next (or previous) window in the window manager's list. The window manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows. If **CWnd** is a top-level window, the function searches for the next (or previous) top-level window; if **CWnd** is a child window, the function searches for the next (or previous) child window.

Return Value Identifies the next (or the previous) window in the window manager's list if the member function is successful. The returned pointer may be temporary and should not be stored for later use.

See Also **::GetNextWindow**

CWnd::GetOpenClipboardWindow

Windows 3.1 Only **static CWnd* PASCAL GetOpenClipboardWindow( );** ♦

Remarks Retrieves the handle of the window that currently has the Clipboard open.

Return Value The handle of the window that currently has the Clipboard open if the function is successful; otherwise **NULL**.

See Also **CWnd::GetClipboardOwner**, **CWnd::GetClipboardViewer**, **CWnd::OpenClipboard**, **::GetOpenClipboardWindow**

CWnd::GetParent

CWnd* GetParent( ) const;

Remarks Retrieves the parent window (if any).

Return Value Identifies the parent window if the member function is successful. Otherwise, the value is **NULL**, which indicates an error or no parent window. The returned pointer may be temporary and should not be stored for later use.

See Also **::GetParent**

CWnd::GetParentFrame

CFrameWnd* GetParentFrame( ) const;

Remarks Call this member function to retrieve the parent frame window. The member function searches up the parent chain until a **CFrameWnd** (or derived class) object is found.

Return Value A pointer to a frame window if successful; otherwise **NULL**.

See Also **CWnd::GetDescendantWindow**, **CWnd::GetParent**, **CFrameWnd::GetActiveView**

CWnd::GetSafeHwnd

HWND GetSafeHwnd( ) const;

Return Value Returns the window handle for a window. Returns **NULL** if the **CWnd** is not attached to a window or if it is used with a **NULL** **CWnd** pointer.

CWnd::GetScrollBarCtrl

virtual CScrollBar* GetScrollBarCtrl(int *nBar* **) const;**

nBar Specifies the type of scroll bar. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the position of the horizontal scroll bar.
- **SB_VERT** Retrieves the position of the vertical scroll bar.

Remarks Call this member function to obtain a pointer to the specified sibling scroll bar or splitter window. This member function does not operate on scroll bars created when the **WS_HSCROLL** or **WS_VSCROLL** bits are set during the creation of a window. The **CWnd** implementation of this function simply returns **NULL**. Derived classes, such as **CView**, implement the described functionality.

Return Value A sibling scroll-bar control, or **NULL** if none.

See Also **CWnd::EnableScrollBarCtrl**

CWnd::GetScrollPos

int GetScrollPos(int *nBar* **) const;**

nBar Specifies the scroll bar to examine. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the position of the horizontal scroll bar.
- **SB_VERT** Retrieves the position of the vertical scroll bar.

Remarks

Retrieves the current position of the scroll box of a scroll bar. The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 50 to 100 and the scroll box is in the middle of the bar, the current position is 75.

Return Value

Specifies the current position of the scroll box in the scroll bar if successful; otherwise 0.

See Also

::GetScrollPos, **CScrollBar::GetScrollPos**

CWnd::GetScrollRange

void GetScrollRange(int *nBar*, **LPINT** *lpMinPos*, **LPINT** *lpMaxPos* **) const;**

nBar Specifies the scroll bar to examine. The parameter can take one of the following values:

- **SB_HORZ** Retrieves the position of the horizontal scroll bar.
- **SB_VERT** Retrieves the position of the vertical scroll bar.

lpMinPos Points to the integer variable that is to receive the minimum position.

lpMaxPos Points to the integer variable that is to receive the maximum position.

Remarks

Copies the current minimum and maximum scroll-bar positions for the given scroll bar to the locations specified by *lpMinPos* and *lpMaxPos*. If **CWnd** does not have a scroll bar, then the **GetScrollRange** member function copies 0 to *lpMinPos* and *lpMaxPos*. The default range for a standard scroll bar is 0 to 100. The default range for a scroll-bar control is empty (both values are 0).

See Also

::GetScrollRange

CWnd::GetStyle

DWORD GetStyle( ) const;

Return Value The window's style.

See Also **::GetWindowLong, CWnd::CreateEx**

CWnd::GetSuperWndProcAddr

Protected **virtual WNDPROC* GetSuperWndProcAddr( ); ♦**

Return Value The address in which to store the default **WndProc** for this class.

CWnd::GetSystemMenu

CMenu* GetSystemMenu(BOOL *bRevert* **) const;**

bRevert Specifies the action to be taken. If *bRevert* is **FALSE**, **GetSystemMenu** returns a handle to a copy of the Control menu currently in use. This copy is initially identical to the Control menu but can be modified. If *bRevert* is **TRUE**, **GetSystemMenu** resets the Control menu back to the default state. The previous, possibly modified, Control menu, if any, is destroyed. The return value is undefined in this case.

Remarks Allows the application to access the Control menu for copying and modification. Any window that does not use **GetSystemMenu** to make its own copy of the Control menu receives the standard Control menu. The pointer returned by this function can be used with the **CMenu::AppendMenu**, **CMenu::InsertMenu**, or **CMenu::ModifyMenu** functions to change the Control menu.

The Control menu initially contains items identified with various ID values such as **SC_CLOSE**, **SC_MOVE**, and **SC_SIZE**. Items on the Control menu generate **WM_SYSCOMMAND** messages. All predefined Control-menu items have ID numbers greater than 0xF000. If an application adds items to the Control menu, it should use ID numbers less than F000.

Windows may automatically dim items on the standard Control menu. **CWnd** can carry out its own checking or dimming by responding to the **WM_INITMENU** messages, which are sent before any menu is displayed.

Return Value Identifies a copy of the Control menu if *bRevert* is **FALSE**. If *bRevert* is **TRUE**, the return value is undefined. The returned pointer may be temporary and should not be stored for later use.

See Also **CMenu::AppendMenu**, **CMenu::InsertMenu**, **CMenu::ModifyMenu**, **::GetSystemMenu**

CWnd::GetTopWindow

CWnd* GetTopWindow( ) const;

Remarks Searches for the top-level child window that belongs to **CWnd**. If **CWnd** has no children, this function returns **NULL**.

Return Value Identifies the top-level child window in a **CWnd** linked list of child windows. If no child windows exist, the value is **NULL**. The returned pointer may be temporary and should not be stored for later use.

See Also **::GetTopWindow**

CWnd::GetUpdateRect

BOOL GetUpdateRect(LPRECT *lpRect*, **BOOL** *bErase* = **FALSE);**

lpRect Points to a **CRect** object or **RECT** structure that is to receive the client coordinates of the update that encloses the update region.

Windows 3.1 Only Set this parameter to **NULL** to determine whether an update region exists within the **CWnd**. If *lpRect* is **NULL**, the **GetUpdateRect** member function returns nonzero if an update region exists and 0 if one does not. This provides a way to determine whether a **WM_PAINT** message resulted from an invalid area. Do not set this parameter to **NULL** in Windows version 3.0 and earlier. ♦

bErase Specifies whether the background in the update region is to be erased.

Remarks Retrieves the coordinates of the smallest rectangle that completely encloses the update region. If **CWnd** was created with the **CS_OWNDC** style and the mapping mode is not **MM_TEXT**, the **GetUpdateRect** member function gives the rectangle in logical coordinates. Otherwise, **GetUpdateRect** gives the rectangle in client coordinates. If there is no update region, **GetUpdateRect** sets the rectangle to be empty (sets all coordinates to 0).

The *bErase* parameter specifies whether **GetUpdateRect** should erase the background of the update region. If *bErase* is **TRUE** and the update region is not empty, the background is erased. To erase the background, **GetUpdateRect** sends the **WM_ERASEBKGND** message. The update rectangle retrieved by the **BeginPaint** member function is identical to that retrieved by the **GetUpdateRect** member function. The **BeginPaint** member function automatically validates the update region, so any call to **GetUpdateRect** made immediately after a call to **BeginPaint** retrieves an empty update region.

Return Value Specifies the status of the update region. The value is nonzero if the update region is not empty; otherwise 0. If the *lpRect* parameter is set to **NULL**, the return value is nonzero if an update region exists; otherwise 0.

See Also **CWnd::BeginPaint**, **::GetUpdateRect**, **CWnd::OnPaint**, **CWnd::RedrawWindow**

CWnd::GetUpdateRgn

int GetUpdateRgn(CRgn* *pRgn*, **BOOL** *bErase* = **FALSE);**

pRgn Identifies the update region.

bErase Specifies whether the background will be erased and nonclient areas of child windows will be drawn. If the value is **FALSE**, no drawing is done.

Remarks Retrieves the update region into a region identified by *pRgn*. The coordinates of this region are relative to the upper-left corner (client coordinates). The **BeginPaint** member function automatically validates the update region, so any call to **GetUpdateRgn** made immediately after a call to **BeginPaint** retrieves an empty update region.

Return Value Specifies a short-integer flag that indicates the type of resulting region. The value can take any one of the following:

- **SIMPLEREGION** The region has no overlapping borders.
- **COMPLEXREGION** The region has overlapping borders.
- **NULLREGION** The region is empty.
- **ERROR** No region was created.

See Also **CWnd::BeginPaint**, **::GetUpdateRgn**

CWnd::GetWindow

CWnd* GetWindow(UINT *nCmd* **) const;**

nCmd Specifies the relationship between **CWnd** and the returned window. It can take one of the following values:

- **GW_CHILD** Identifies the **CWnd** first child window.
- **GW_HWNDFIRST** If **CWnd** is a child window, returns the first sibling window. Otherwise, it returns the first top-level window in the list.
- **GW_HWNDLAST** If **CWnd** is a child window, returns the last sibling window. Otherwise, it returns the last top-level window in the list.
- **GW_HWNDNEXT** Returns the next window on the window manager's list.
- **GW_HWNDPREV** Returns the previous window on the window manager's list.
- **GW_OWNER** Identifies the **CWnd** owner.

Return Value

Returns a pointer to the window requested, or **NULL** if none. The returned pointer may be temporary and should not be stored for later use.

See Also

CWnd::GetParent, **CWnd::GetNextWindow**, **::GetWindow**

CWnd::GetWindowDC

CDC* GetWindowDC( );

Remarks

Retrieves the display context for the entire window, including caption bar, menus, and scroll bars. A window display context permits painting anywhere in **CWnd**, since the origin of the context is the upper-left corner of **CWnd** instead of the client area. Default attributes are assigned to the display context each time it retrieves the context. Previous attributes are lost. **GetWindowDC** is intended to be used for special painting effects within the **CWnd** nonclient area. Painting in nonclient areas of any window is not recommended.

The **GetSystemMetrics** Windows function can be used to retrieve the dimensions of various parts of the nonclient area, such as the caption bar, menu, and scroll bars. After painting is complete, the **ReleaseDC** member function must be called to release the display context. Failure to release the display context will seriously affect painting requested by applications due to limitations on the number of device contexts that can be open at the same time.

Return Value Identifies the display context for the given window if the function is successful; otherwise **NULL**. The returned pointer may be temporary and should not be stored for later use.

See Also **::GetSystemMetrics**, **CWnd::ReleaseDC**, **::GetWindowDC**, **CWnd::GetDC**, **CWindowDC**

CWnd::GetWindowPlacement

Windows 3.1 Only **BOOL GetWindowPlacement(WINDOWPLACEMENT FAR*** *lpwndpl* **) const;** ♦

lpwndpl Points to the **WINDOWPLACEMENT** structure that receives the show state and position information.

Remarks Retrieves the show state and the normal (restored), minimized, and maximized positions of a window. The **flags** member of the **WINDOWPLACEMENT** structure retrieved by this function is always 0. If **CWnd** is maximized, the **showCmd** member of **WINDOWPLACEMENT** is **SW_SHOWMAXIMIZED**. If the window is minimized, it is **SW_SHOWMINIMIZED.** It is **SW_SHOWNORMAL** otherwise.

Return Value Nonzero if the function is successful; otherwise 0.

See Also **CWnd::SetWindowPlacement**, **::GetWindowPlacement**

CWnd::GetWindowRect

void GetWindowRect(LPRECT *lpRect* **) const;**

lpRect Points to a **CRect** object or a **RECT** structure that will receive the screen coordinates of the upper-left and lower-right corners.

Remarks Copies the dimensions of the bounding rectangle of the **CWnd** object to the structure pointed to by *lpRect*. The dimensions are given in screen coordinates relative to the upper-left corner of the display screen. The dimensions of the caption, border, and scroll bars, if present, are included.

See Also **CWnd::GetClientRect**, **CWnd::MoveWindow**, **CWnd::SetWindowPos**, **::GetWindowRect**

CWnd::GetWindowText

int GetWindowText(LPSTR *lpszStringBuf*, **int** *nMaxCount* **) const;**

void GetWindowText(CString& *rString* **) const;**

lpszStringBuf Points to the buffer that is to receive the copied string of the window's title.

nMaxCount Specifies the maximum number of characters to be copied to the buffer. If the string is longer than the number of characters specified in *nMaxCount*, it is truncated.

rString A **CString** object that is to receive the copied string of the window's title.

Remarks

Copies the **CWnd** caption title (if it has one) into the buffer pointed to by *lpszStringBuf* or into the destination string *rString*. If the **CWnd** object is a control, the **GetWindowText** member function copies the text within the control instead of copying the caption. This member function causes the **WM_GETTEXT** message to be sent to the **CWnd** object.

Return Value

Specifies the length, in bytes, of the copied string, not including the terminating null character. It is 0 if **CWnd** has no caption or if the caption is empty.

See Also

CWnd::SetWindowText, **WM_GETTEXT**, **CWnd::GetWindowTextLength**

CWnd::GetWindowTextLength

int GetWindowTextLength( ) const;

Remarks

Returns the length of the **CWnd** object caption title. If **CWnd** is a control, the **GetWindowTextLength** member function returns the length of the text within the control instead of the caption. This member function causes the **WM_GETTEXTLENGTH** message to be sent to the **CWnd** object.

Return Value

Specifies the text length, not including any null-termination character. The value is 0 if no such text exists.

See Also

::GetWindowTextLength, **WM_GETTEXTLENGTH**, **CWnd::GetWindowText**

CWnd::HideCaret

void HideCaret( );

Remarks

Hides the caret by removing it from the display screen. Although the caret is no longer visible, it can be displayed again by using the **ShowCaret** member function. Hiding the caret does not destroy its current shape. Hiding is cumulative. If **HideCaret** has been called five times in a row, the **ShowCaret** member function must be called five times before the caret will be shown.

See Also

CWnd::ShowCaret, **::HideCaret**

CWnd::HiliteMenuItem

BOOL HiliteMenuItem(CMenu* *pMenu*, **UINT** *nIDHiliteItem*, **UINT** *nHilite* **);**

pMenu Identifies the top-level menu that contains the item to be highlighted.

nIDHiliteItem Specifies the menu item to be highlighted, depending on the value of the *nHilite* parameter.

nHilite Specifies whether the menu item is highlighted or the highlight is removed. It can be a combination of **MF_HILITE** or **MF_UNHILITE** with **MF_BYCOMMAND** or **MF_BYPOSITION**. The values can be combined using the bitwise-OR operator. These values have the following meanings:

- **MF_BYCOMMAND** Interprets *nIDHiliteItem* as the menu-item ID (the default interpretation).
- **MF_BYPOSITION** Interprets *nIDHiliteItem* as the zero-based offset of the menu item.
- **MF_HILITE** Highlights the item. If this value is not given, the highlight is removed from the item.
- **MF_UNHILITE** Removes the highlight from the item.

Remarks

Highlights or removes the highlight from a top-level (menu-bar) menu item. The **MF_HILITE** and **MF_UNHILITE** flags can be used only with this member function; they cannot be used with the **ModifyMenu** member function.

Return Value

Specifies whether the menu item was highlighted. Nonzero if the item was highlighted; otherwise 0.

See Also

CMenu::ModifyMenu, **::HiliteMenuItem**

CWnd::Invalidate

void Invalidate(BOOL *bErase* **= TRUE);**

bErase Specifies whether the background within the update region is to be erased.

Remarks

Invalidates the entire client area of **CWnd**. The client area is marked for painting when the next **WM_PAINT** message occurs. The region can also be validated before a **WM_PAINT** message occurs by the **ValidateRect** or **ValidateRgn** member function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is **TRUE**, the background is erased when the **BeginPaint** member function is called; if *bErase* is **FALSE**, the background remains unchanged. If *bErase* is **TRUE** for any part of the update region, the background in the entire region, not just in the given part, is erased. Windows sends a **WM_PAINT** message whenever the **CWnd** update region is not empty and there are no other messages in the application queue for that window.

See Also

CWnd::BeginPaint, **CWnd::ValidateRect**, **CWnd::ValidateRgn**, **::InvalidateRect**

CWnd::InvalidateRect

void InvalidateRect(LPCRECT *lpRect*, **BOOL** *bErase* **= TRUE);**

lpRect Points to a **CRect** object or a **RECT** structure that contains the rectangle (in client coordinates) to be added to the update region. If *lpRect* is **NULL**, the entire client area is added to the region.

bErase Specifies whether the background within the update region is to be erased.

Remarks

Invalidates the client area within the given rectangle by adding that rectangle to the **CWnd** update region. The invalidated rectangle, along with all other areas in the update region, is marked for painting when the next **WM_PAINT** message is sent. The invalidated areas accumulate in the update region until the region is processed when the next **WM_PAINT** call occurs, or until the region is validated by the **ValidateRect** or **ValidateRgn** member function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is **TRUE**, the background is erased when the **BeginPaint** member function is called; if *bErase* is **FALSE**, the background remains unchanged. If *bErase* is **TRUE** for any part of the update region, the background in the entire region is erased, not just in the given part. Windows sends a **WM_PAINT** message whenever the **CWnd** update region is not empty and there are no other messages in the application queue for that window.

See Also **CWnd::BeginPaint**, **CWnd::ValidateRect**, **CWnd::ValidateRgn**, **::InvalidateRect**

CWnd::InvalidateRgn

void InvalidateRgn(CRgn* *pRgn*, **BOOL** *bErase* = **TRUE);**

pRgn Identifies the region to be added to the update region. The region is assumed to have client coordinates. If this parameter is **NULL**, the entire client area is added to the update region.

bErase Specifies whether the background within the update region is to be erased.

Remarks Invalidates the client area within the given region by adding it to the current update region of **CWnd**. The invalidated region, along with all other areas in the update region, is marked for painting when the **WM_PAINT** message is next sent. The invalidated areas accumulate in the update region until the region is processed when a **WM_PAINT** message is next sent, or until the region is validated by the **ValidateRect** or **ValidateRgn** member function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is **TRUE**, the background is erased when the **BeginPaint** member function is called; if *bErase* is **FALSE**, the background remains unchanged. If *bErase* is **TRUE** for any part of the update region, the background in the entire region, not just in the given part, is erased. Windows sends a **WM_PAINT** message whenever the **CWnd** update region is not empty and there are no other messages in the application queue for that window. The given region must have been previously created by one of the region functions.

See Also **CWnd::BeginPaint**, **CWnd::ValidateRect**, **CWnd::ValidateRgn**, **::InvalidateRgn**

CWnd::IsChild

BOOL IsChild(const CWnd* *pWnd* **) const;**

pWnd Identifies the window to be tested.

Remarks Indicates whether the window specified by *pWnd* is a child window or other direct descendant of **CWnd**. A child window is the direct descendant of **CWnd** if the **CWnd** object is in the chain of parent windows that leads from the original pop-up window to the child window.

Return Value Specifies the outcome of the function. The value is nonzero if the window identified by *pWnd* is a child window of **CWnd**; otherwise 0.

See Also **::IsChild**

CWnd::IsDlgButtonChecked

UINT IsDlgButtonChecked(int *nIDButton* **) const;**

nIDButton Specifies the integer identifier of the button control.

Remarks Determines whether a button control has a check mark next to it. If the button is a three-state control, the member function determines if it is dimmed, checked, or neither.

Return Value Nonzero if the given control is checked, and 0 if it is not checked. Only radio buttons and check boxes can be checked. For three-state buttons, the return value can be 2 if the button is indeterminate. This member function returns 0 for a pushbutton.

See Also **::IsDlgButtonChecked**, **CButton::GetCheck**

CWnd::IsIconic

BOOL IsIconic( ) const;

Remarks Specifies whether **CWnd** is minimized (iconic).

Return Value Nonzero if **CWnd** is minimized; otherwise 0.

See Also **::IsIconic**

CWnd::IsWindowEnabled

BOOL IsWindowEnabled( ) const;

Remarks Specifies whether **CWnd** is enabled for mouse and keyboard input.

Return Value Nonzero if **CWnd** is enabled; otherwise 0.

See Also **::IsWindowEnabled**

CWnd::IsWindowVisible

BOOL IsWindowVisible( ) const;

Remarks Determines the visibility state of the given window. A window possesses a visibility state indicated by the **WS_VISIBLE** style bit. When this style bit is set with a call to the **ShowWindow** member function, the window is displayed and subsequent drawing to the window is displayed as long as the window has the style bit set. Any drawing to a window that has the **WS_VISIBLE** style will not be displayed if the window is covered by other windows or is clipped by its parent window.

Return Value Nonzero if **CWnd** is visible (has the **WS_VISIBLE** style bit set, and parent window is visible). Since the return value reflects the state of the **WS_VISIBLE** style bit, the return value may be nonzero even though **CWnd** is totally obscured by other windows.

See Also **CWnd::ShowWindow**, **::IsWindowVisible**

CWnd::IsZoomed

BOOL IsZoomed( ) const;

Remarks Determines whether **CWnd** has been maximized.

Return Value Nonzero if **CWnd** is maximized; otherwise 0.

See Also **::IsZoomed**

CWnd::KillTimer

BOOL KillTimer(int *nIDEvent* **);**

nIDEvent The value of the timer event passed to **SetTimer**.

Remarks Kills the timer event identified by *nIDEvent* from the earlier call to **SetTimer**. Any pending **WM_TIMER** messages associated with the timer are removed from the message queue.

Return Value Specifies the outcome of the function. The value is nonzero if the event was killed. It is 0 if the **KillTimer** member function could not find the specified timer event.

See Also **CWnd::SetTimer**, **::KillTimer**

CWnd::LockWindowUpdate

Windows 3.1 Only **BOOL LockWindowUpdate( );** ♦

Remarks Disables or reenables drawing in the given window. A locked window cannot be moved. Only one window can be locked at a time.

If an application with a locked window (or any locked child windows) calls the **GetDC**, **GetDCEx**, or **BeginPaint** Windows function, the called function returns a device context whose visible region is empty. This will occur until the application unlocks the window by calling the **LockWindowUpdate** member function.

While window updates are locked, the system keeps track of the bounding rectangle of any drawing operations to device contexts associated with a locked window. When drawing is reenabled, this bounding rectangle is invalidated in the locked window and its child windows to force an eventual **WM_PAINT** message to update the screen. If no drawing has occurred while the window updates were locked, no area is invalidated.

The **LockWindowUpdate** member function does not make the given window invisible and does not clear the **WS_VISIBLE** style bit.

Return Value Nonzero if the function is successful. It is 0 if a failure occurs or if the **LockWindowUpdate** function has been used to lock another window.

See Also **CWnd::GetDCEx**, **::LockWindowUpdate**

CWnd::MapWindowPoints

Windows 3.1 Only **void MapWindowPoints(CWnd*** *pwndTo*, **LPRECT** *lpRect* **) const;** ♦

void MapWindowPoints(CWnd* *pwndTo*, **LPPOINT** *lpPoint*, **UINT** *nCount* **) const;**

pwndTo Identifies the window to which points are converted. If this parameter is **NULL**, the points are converted to screen coordinates.

lpRect Specifies the rectangle whose points are to be converted.

lpPoint A pointer to an array of **POINT** structures that contain the set of points to be converted.

nCount Specifies the number of **POINT** structures in the array pointed to by *lpPoint*.

Remarks Converts (maps) a set of points from the coordinate space of the **CWnd** to the coordinate space of another window.

See Also **CWnd::ClientToScreen**, **CWnd::ScreenToClient**, **::MapWindowPoints**

CWnd::MessageBox

int MessageBox(LPCSTR *lpszText*, **LPCSTR** *lpszCaption* = **NULL, UINT** *nType* = **MB_OK);**

lpszText Points to a **CString** object or null-terminated string containing the message to be displayed.

lpszCaption Points to a **CString** object or null-terminated string to be used for the message-box caption. If *lpszCaption* is **NULL**, the default caption "Error" is used.

nType Specifies the contents and behavior of the message box.

Remarks Creates and displays a window that contains an application-supplied message and caption, plus a combination of the predefined icons and pushbuttons described in the "Message-Box Styles" list. This manual shows this list in the **AfxMessageBox** global function description. Use the global function **AfxMessageBox** instead of this member function to implement a message box in your application.

Return Value Specifies the outcome of the function. It is 0 if there is not enough memory to create the message box.

See Also **::MessageBox**, **AfxMessageBox**

CWnd::MoveWindow

void MoveWindow(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*,
BOOL *bRepaint* = **TRUE);**

void MoveWindow(LPCRECT *lpRect*, **BOOL** *bRepaint* = **TRUE);**

x Specifies the new position of the left side of the **CWnd**.

y Specifies the new position of the top of the **CWnd**.

nWidth Specifies the new width of the **CWnd**.

nHeight Specifies the new height of the **CWnd**.

bRepaint Specifies whether **CWnd** is to be repainted. If **TRUE**, **CWnd** receives a **WM_PAINT** message in its **OnPaint** message handler as usual. If this parameter is **FALSE**, no repainting of any kind occurs. This applies to the client area, to the nonclient area (including the title and scroll bars), and to any part of the parent window uncovered as a result of **Cwnd**'s move. When this parameter is **FALSE**, the application must explicitly invalidate or redraw any parts of **CWnd** and parent window that must be redrawn.

lpRect The **CRect** object or **RECT** structure that specifies the new size and position.

Remarks Changes the position and dimensions. For a top-level **CWnd** object, the *x* and *y* parameters are relative to the upper-left corner of the screen. For a child **CWnd** object, they are relative to the upper-left corner of the parent window's client area. The **MoveWindow** function sends the **WM_GETMINMAXINFO** message. Handling this message gives **CWnd** the opportunity to modify the default values for the largest and smallest possible windows. If the parameters to the **MoveWindow** member function exceed these values, the values can be replaced by the minimum or maximum values in the **WM_GETMINMAXINFO** handler.

See Also **CWnd::SetWindowPos**, **WM_GETMINMAXINFO**, **::MoveWindow**

CWnd::OnActivate

Protected

afx_msg void OnActivate(UINT *nState*, **CWnd*** *pWndOther*, **BOOL** *bMinimized* **);** ♦

nState Specifies whether the **CWnd** is being activated or deactivated. It can be one of the following values:

- **WA_INACTIVE** The window is being deactivated.
- **WA_ACTIVE** The window is being activated through some method other than a mouse click (for example, by use of the keyboard interface to select the window).
- **WA_CLICKACTIVE** The window is being activated by a mouse click.

pWndOther Pointer to the **CWnd** being activated or deactivated. The pointer can be **NULL**, and it may be temporary.

bMinimized Specifies the minimized state of the **CWnd** being activated or deactivated. A value of **TRUE** indicates the window is minimized.

If **TRUE**, the **CWnd** is being activated; otherwise deactivated.

Remarks

Called when a **CWnd** object is being activated or deactivated. First, the main window being deactivated has **OnActivate** called, and then the main window being activated has **OnActivate** called.

If the **CWnd** object is activated with a mouse click, it will also receive an **OnMouseActivate** member function call.

See Also

WM_MOUSEACTIVATE, WM_NCACTIVATE, WM_ACTIVATE

CWnd::OnActivateApp

Protected

afx_msg void OnActivateApp(BOOL *bActive*, **HTASK** *hTask* **);** ♦

bActive Specifies whether the **CWnd** is being activated or deactivated. **TRUE** means the **CWnd** is being activated. **FALSE** means the **CWnd** is being deactivated.

hTask Specifies a task handle. If *bActive* is **TRUE**, the handle identifies the task that owns the **CWnd** being deactivated. If *bActive* is **FALSE**, the handle identifies the task that owns the **CWnd** being activated.

Remarks Called for all top-level windows of the task being activated and for all top-level windows of the task being deactivated.

See Also **WM_ACTIVATEAPP**

CWnd::OnAskCbFormatName

Protected **afx_msg void OnAskCbFormatName(UINT** *nMaxCount*, **LPSTR** *lpszString* **);** ♦

nMaxCount Specifies the maximum number of bytes to copy.

lpszString Points to the buffer where the copy of the format name is to be stored.

Remarks Called when the Clipboard contains a data handle for the **CF_OWNERDISPLAY** format (that is, when the Clipboard owner will display the Clipboard contents). The Clipboard owner should provide a name for its format. Override this member function and copy the name of the **CF_OWNERDISPLAY** format into the specified buffer, not exceeding the maximum number of bytes specified.

See Also **WM_ASKCBFORMATNAME**

CWnd::OnCancelMode

Protected **afx_msg void OnCancelMode( );** ♦

Remarks Called to inform **CWnd** to cancel any internal mode. If the **CWnd** object has the focus, its **OnCancelMode** member function is called when a dialog box or message box is displayed. This gives the **CWnd** the opportunity to cancel modes such as mouse capture.

The default implementation responds by calling the **ReleaseCapture** Windows function. Override this member function in your derived class to handle other modes.

See Also **CWnd::Default**, **::ReleaseCapture**, **WM_CANCELMODE**

CWnd::OnChangeCbChain

Protected

afx_msg void OnChangeCbChain(HWND *hWndRemove*, **HWND** *hWndAfter* **);** ♦

hWndRemove Specifies the window handle that is being removed from the Clipboard-viewer chain.

hWndAfter Specifies the window handle that follows the window being removed from the Clipboard-viewer chain.

Remarks

Called for each window in the Clipboard-viewer chain to notify it that a window is being removed from the chain. Each **CWnd** object that receives an **OnChangeCbChain** call should use the **SendMessage** Windows function to send the **WM_CHANGECBCHAIN** message to the next window in the Clipboard-viewer chain (the handle returned by **SetClipboardViewer**). If *hWndRemove* is the next window in the chain, the window specified by *hWndAfter* becomes the next window, and Clipboard messages are passed on to it.

See Also

CWnd::ChangeClipboardChain, **::SendMessage**

CWnd::OnChar

Protected

afx_msg void OnChar(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Contains the virtual-key code value of the key.

nRepCnt Contains the repeat count, the number of times the keystroke is repeated when user holds down the key.

nFlags Contains the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description of nFlags
0–7	Scan code (OEM-dependent value).
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call; 0 if the key is up).
15	Transition state (1 if the key is being released; 0 if the key is being pressed).

Remarks

Called when a keystroke translates to a nonsystem character. This function is called before the **OnKeyUp** member function and after the **OnKeyDown** member function are called. **OnChar** contains the value of the keyboard key being pressed or released. Since there is not necessarily a one-to-one correspondence between keys pressed and **OnChar** calls generated, the information in *nFlags* is generally not useful to applications. The information in *nFlags* applies only to the most recent call to the **OnKeyUp** member function or the **OnKeyDown** member function that precedes the call to **OnChar**.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

See Also

WM_CHAR, **WM_KEYDOWN**, **WM_KEYUP**

CWnd::OnCharToItem

Protected

afx_msg int OnCharToItem(UINT *nChar*, **CListBox*** *pListBox*, **UINT** *nIndex* **);** ♦

nChar Specifies the value of the key pressed by the user.

pListBox Specifies a pointer to the list box. It may be temporary.

nIndex Specifies the current caret position.

Remarks

Called when a list box with the **LBS_WANTKEYBOARDINPUT** style sends its owner a **WM_CHARTOITEM** message in response to a **WM_CHAR** message.

Return Value

Specifies the action that the application performed in response to the call. A return value of –2 indicates that the application handled all aspects of selecting the item and wants no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the keystroke. A return value of 0 or greater specifies the zero-based index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

See Also

WM_CHAR, **WM_CHARTOITEM**

CWnd::OnChildActivate

Protected **afx_msg void OnChildActivate( );** ♦

Remarks If the **CWnd** object is a multiple document interface (MDI) child window, **OnChildActivate** is called when the user clicks the window's title bar or when the window is activated, moved, or sized.

See Also **CWnd::SetWindowPos**, **WM_CHILDACTIVATE**

CWnd::OnChildNotify

Protected **virtual BOOL OnChildNotify(UINT** *message*, **WPARAM** *wParam*, **LPARAM** *lParam*, **LRESULT*** *pLResult* **);**

message A Windows message number sent to a parent window.

wParam The **wparam** associated with the message.

lParam The **lparam** associated with the message.

pLResult A pointer to a value to be returned from the parent's window procedure. This pointer will be **NULL** if no return value is expected.

Remarks Called by this window's parent window when it receives a notification message that applies to this window. Never call this member function directly.

The default implementation of this member function returns 0, which means that the parent should handle the message. Override this member function to extend the manner in which a control responds to notification messages.

Return Value Nonzero if this window handles the message sent to its parent; otherwise 0.

CWnd::OnClose

Protected **afx_msg void OnClose( );** ♦

Remarks Called as a signal that the **CWnd** or an application is to terminate. The default implementation calls **DestroyWindow**.

See Also **CWnd::DestroyWindow**, **WM_CLOSE**

CWnd::OnCommand

Protected **virtual BOOL OnCommand(WPARAM** *wParam*, **LPARAM** *lParam* **);** ♦

wParam Identifies the command ID of the menu item or control.

lParam The low-order word of *lParam* identifies the control that sends the message if the message is from a control. Otherwise, the low-order word is 0. The high-order word of *lParam* specifies the notification message if the message is from a control. If the message is from an accelerator, the high-order word is 1. If the message is from a menu, the high-order word is 0.

Remarks Called when the user selects an item from a menu, when a child control sends a notification message, or when an accelerator keystroke is translated. **OnCommand** processes the message map for control notification and **ON_COMMAND** entries, and calls the appropriate member function. Override this member function in your derived class to handle the **WM_COMMAND** message. An override will not process the message map unless the base class **OnCommand** is called.

Return Value An application returns nonzero if it processes this message; otherwise 0.

See Also **WM_COMMAND**, **CCmdTarget::OnCmdMsg**

CWnd::OnCompacting

Protected **afx_msg void OnCompacting(UINT** *nCpuTime* **);** ♦

nCpuTime Specifies the ratio of CPU time currently spent by Windows compacting memory to CPU time spent performing other operations. For example, 8000h represents 50 percent of CPU time spent compacting memory.

Remarks Called for all top-level windows when Windows detects that more than 12.5 percent of system time over a 30- to 60-second interval is being spent compacting memory. This indicates that system memory is low. When a **CWnd** object receives this call, it should free as much memory as possible, taking into account the current level of activity of the application and the total number of applications running in Windows. The application can call the **GetNumTasks** Windows function to determine how many applications are running.

See Also **::GetNumTasks**, **WM_COMPACTING**

CWnd::OnCompareItem

Protected

afx_msg int OnCompareItem(int *nIDCtl*, **LPCOMPAREITEMSTRUCT** *lpCompareItemStruct* **);** ♦

Windows 3.1 Only

nIDCtl The identifier of the control that sent the **WM_COMPAREITEM** message. ♦

lpCompareItemStruct Contains a long pointer to a **COMPAREITEMSTRUCT** data structure that contains the identifiers and application-supplied data for two items in the combo or list box.

Remarks

Specifies the relative position of a new item in a child sorted owner-draw combo or list box. If a combo or list box is created with the **CBS_SORT** or **LBS_SORT** style, Windows sends the combo-box or list-box owner a **WM_COMPAREITEM** message whenever the application adds a new item.

Two items in the combo or list box are reformed in a **COMPAREITEMSTRUCT** structure pointed to by *lpCompareItemStruct*. **OnCompareItem** should return a value that indicates which of the items should appear before the other. Typically, Windows makes this call several times until it finds the new item's exact position.

If the **hwndItem** member of the **COMPAREITEMSTRUCT** structure belongs to a **CListBox** or **CComboBox** object, then the **CompareItem** virtual function of the appropriate class is called. Override **CComboBox::CompareItem** or **CListBox::CompareItem** in your derived **CListBox** or **CComboBox** class to do the item comparison.

Return Value

Indicates the relative position of the two items. It may be any of the following values:

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

COMPAREITEMSTRUCT Structure

A **COMPAREITEMSTRUCT** data structure has this form:

```
typedef struct tagCOMPAREITEMSTRUCT {
    UINT   CtlType;
    UINT   CtlID;
    HWND   hwndItem;
    UINT   itemID1;
    DWORD  itemData1;
    UINT   itemID2;
    DWORD  itemData2;
} COMPAREITEMSTRUCT;
```

Members The **COMPAREITEMSTRUCT** members are as follows:

CtlType **ODT_LISTBOX** (which specifies an owner-draw list box) or **ODT_COMBOBOX** (which specifies an owner-draw combo box).

CtlID The control ID for the list box or combo box.

hwndItem The window handle of the control.

itemID1 The index of the first item in the list box or combo box being compared.

itemData1 Application-supplied data for the first item being compared. This value was passed in the call that added the item to the combo or list box.

itemID2 Index of the second item in the list box or combo box being compared.

itemData2 Application-supplied data for the second item being compared. This value was passed in the call that added the item to the combo or list box.

See Also **WM_COMPAREITEM**, **CListBox::CompareItem**, **CComboBox::CompareItem**

CWnd::OnCreate

Protected **afx_msg int OnCreate(LPCREATESTRUCT** *lpCreateStruct* **);** ♦

lpCreateStruct Points to a **CREATESTRUCT** structure that contains information about the **CWnd** object being created.

Remarks Called when an application requests that the Windows window be created by calling the **Create** or **CreateEx** member function. The **CWnd** object receives this call after the window is created but before it becomes visible. **OnCreate** is called before the **Create** or **CreateEx** member function returns. Override this member function to perform any needed initialization of a derived class. The **CREATESTRUCT** structure contains copies of the parameters used to create the window.

Return Value **OnCreate** must return 0 to continue the creation of the **CWnd** object. If the application returns –1, the window will be destroyed.

CREATESTRUCT Structure

A **CREATESTRUCT** structure has the following form:

```
typedef struct tagCREATESTRUCT {
   void FAR* lpCreateParams;
   HINSTANCE hInstance;
   HMENU     hMenu;
   HWND      hwndParent;
   int       cy;
   int       cx;
   int       y;
   int       x;
   LONG      style;
   LPCSTR    lpszName;
   LPCSTR    lpszClass;
   DWORD     dwExStyle;
} CREATESTRUCT;
```

Members

The **CREATESTRUCT** members are as follows:

lpCreateParams Points to data to be used to create the window.

hInstance Identifies the module-instance handle of the module that owns the new window.

hMenu Identifies the menu to be used by the new window. If a child window, contains the integer ID.

hwndParent Identifies the window that owns the new window. This member is **NULL** if the new window is a top-level window.

cy Specifies the height of the new window.

cx Specifies the width of the new window.

y Specifies the y-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window; otherwise, coordinates are relative to the screen origin.

x Specifies the x-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window; otherwise, coordinates are relative to the screen origin.

style Specifies the new window's style.

lpszName Points to a null-terminated string that specifies the new window's name.

lpszClass Points to a null-terminated string that specifies the new window's Windows class name (a **WNDCLASS** structure).

dwExStyle Specifies the extended style for the new window.

See Also **CWnd::CreateEx**, **CWnd::OnNcCreate**, **WM_CREATE**, **CWnd::Default**, **CWnd::FromHandle**

CWnd::OnCtlColor

Protected **afx_msg HBRUSH OnCtlColor(CDC*** *pDC*, **CWnd*** *pWnd*, **UINT** *nCtlColor* **);** ♦

pDC Contains a pointer to the display context for the child window. May be temporary.

pWnd Contains a pointer to the control asking for the color. May be temporary.

nCtlColor Contains one of the following values, specifying the type of control:

- **CTLCOLOR_BTN** Button control
- **CTLCOLOR_DLG** Dialog box
- **CTLCOLOR_EDIT** Edit control
- **CTLCOLOR_LISTBOX** List-box control
- **CTLCOLOR_MSGBOX** Message box
- **CTLCOLOR_SCROLLBAR** Scroll-bar control
- **CTLCOLOR_STATIC** Static control

Remarks Called when a child control is about to be drawn. Most controls send this message to their parent (usually a dialog box) to prepare the *pDC* for drawing the control using the correct colors.

To change the text color, call the **SetTextColor** member function with the desired red, green, and blue (RGB) values. To change the background color of a single-line edit control, set the brush handle in both the **CTLCOLOR_EDIT** and **CTLCOLOR_MSGBOX** message codes, and call the **CDC::SetBkColor** function in response to the **CTLCOLOR_EDIT** code.

OnCtlColor will not be called for the list box of a drop-down combo box because the drop-down list box is actually a child of the combo box and not a child of the window. To change the color of the drop-down list box, create a **CComboBox** with an override of **OnCtlColor** that checks for **CTLCOLOR_LISTBOX** in the *nCtlColor* parameter. In this handler, the **SetBkColor** member function must be used to set the background color for the text.

Return Value

OnCtlColor must return a handle to the brush that is to be used for painting the control background.

See Also

CDC::SetBkColor, **WM_CTLCOLOR**

CWnd::OnDeadChar

Protected

afx_msg void OnDeadChar(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the dead-key character value.

nRepCnt Specifies the repeat count.

nFlags Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

Remarks

Called when the **OnKeyUp** member function and the **OnKeyDown** member functions are called. This member function can be used to specify the character value of a dead key. A dead key is a key, such as the umlaut (double-dot) character, that is combined with other characters to form a composite character. For example, the umlaut-O character consists of the dead key, umlaut, and the O key.

An application typically uses **OnDeadChar** to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret. Since there is not necessarily a one-to-one correspondence between keys pressed and **OnDeadChar** calls, the information in *nFlags* is generally not useful to applications. The information in *nFlags* applies only to the most recent call to the **OnKeyUp** member function or the **OnKeyDown** member function that precedes the **OnDeadChar** call.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

See Also **WM_DEADCHAR**

CWnd::OnDeleteItem

Protected

afx_msg void OnDeleteItem(int *nIDCtl*, **LPDELETEITEMSTRUCT** *lpDeleteItemStruct*); ♦

nIDCtl The identifier of the control that sent the **WM_DELETEITEM** message.

lpDeleteItemStruct Specifies a long pointer to a **DELETEITEMSTRUCT** data structure that contains information about the deleted list-box item. This structure is described later.

Remarks

Called to inform the owner of an owner-draw list box or combo box that the list box or combo box is destroyed or that items have been removed by **CComboBox::DeleteString**, **CListBox::DeleteString**, **CComboBox::ResetContent**, or **CListBox::ResetContent**.

If the **hwndItem** member of the **DELETEITEMSTRUCT** structure belongs to a combo box or list box, then the **DeleteItem** virtual function of the appropriate class is called. Override the **DeleteItem** member function of the appropriate control's class to delete item-specific data.

DELETEITEM-STRUCT Structure

A **DELETEITEMSTRUCT** data structure has this form:

```
typedef struct tagDELETEITEMSTRUCT {
    UINT  CtlType
    UINT  CtlID;
    UINT  itemID;
    HWND  hwndItem;
    DWORD itemData;
} DELETEITEMSTRUCT;
```

Members

The **DELETEITEMSTRUCT** members are as follows:

CtlType Contains **ODT_LISTBOX** (which specifies an owner-draw list box) or **ODT_COMBOBOX** (which specifies an owner-draw combo box).

CtlID Contains the control ID for the list box or combo box.

itemID Contains the index of the item in the list box or combo box being removed.

hwndItem Contains the window handle of the control.

itemData Contains the value passed to the control by **CComboBox::AddString**, **CComboBox::InsertString**, **CListBox::AddString**, or **CListBox::InsertString**.

See Also

CComboBox::DeleteString, **CListBox::DeleteString**, **CComboBox::ResetContent**, **CListBox::ResetContent**, **WM_DELETEITEM**, **CListBox::DeleteItem**, **CComboBox::DeleteItem**

CWnd::OnDestroy

Protected

afx_msg void OnDestroy( ); ♦

Remarks

Called to inform the **CWnd** object that it is being destroyed. **OnDestroy** is called after the **CWnd** object is removed from the screen. **OnDestroy** is called first for the **CWnd** being destroyed, then for the child windows of **CWnd** as they are destroyed. It can be assumed that all child windows still exist while **OnDestroy** runs. If the **CWnd** object being destroyed is part of the Clipboard-viewer chain (set by calling the **SetClipboardViewer** member function), the **CWnd** must remove itself from the Clipboard-viewer chain by calling the **ChangeClipboardChain** member function before returning from the **OnDestroy** function.

See Also

CWnd::ChangeClipboardChain, **CWnd::DestroyWindow**, **CWnd::SetClipboardViewer**

CWnd::OnDestroyClipboard

Protected **afx_msg void OnDestroyClipboard( );** ♦

Remarks Called for the Clipboard owner when the Clipboard is emptied through a call to the **EmptyClipboard** Windows function.

See Also **::EmptyClipboard, WM_DESTROYCLIPBOARD**

CWnd::OnDevModeChange

Protected **afx_msg void OnDevModeChange(LPSTR** *lpDeviceName* **);** ♦

lpDeviceName Points to the device name specified in the Windows initialization file, WIN.INI.

Remarks Called for all top-level **CWnd** objects when the user changes device-mode settings. Applications that handle the **WM_DEVMODECHANGE** message may reinitialize their device-mode settings. Applications that use the Windows **ExtDeviceMode** function to save and restore device settings typically do not process this function. This function is not called when the user changes the default printer from Control Panel. In this case, the **OnWinIniChange** function is called.

See Also **WM_DEVMODECHANGE**

CWnd::OnDrawClipboard

Protected **afx_msg void OnDrawClipboard( );** ♦

Remarks Called for each window in the Clipboard-viewer chain when the contents of the Clipboard change. Only applications that have joined the Clipboard-viewer chain by calling the **SetClipboardViewer** member function need to respond to this call. Each window that receives an **OnDrawClipboard** call should call the **SendMessage** Windows function to pass a **WM_DRAWCLIPBOARD** message on to the next window in the Clipboard-viewer chain. The handle of the next window is returned by the **SetClipboardViewer** member function; it may be modified in response to an **OnChangeCbChain** member function call.

See Also **::SendMessage, CWnd::SetClipboardViewer, WM_CHANGECBCHAIN, WM_DRAWCLIPBOARD**

CWnd::OnDrawItem

Protected

afx_msg void OnDrawItem(int *nIDCtl*, **LPDRAWITEMSTRUCT** *lpDrawItemStruct* **);** ♦

Windows 3.1 Only

nIDCtl Contains the identifier of the control that sent the **WM_DRAWITEM** message. If a menu sent the message, *nIDCtl* contains 0. ♦

lpDrawItemStruct Specifies a long pointer to a **DRAWITEMSTRUCT** structure that has information about the item to be drawn and the type of drawing required.

Remarks

Called for the owner of an owner-draw button control, combo-box control, list-box control, or menu when a visual aspect of the control or menu has changed. The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing operation that is to be performed. The data in this member allows the owner of the control to determine what drawing action is required. Before returning from processing this message, an application should ensure that the device context identified by the **hDC** member of the **DRAWITEMSTRUCT** structure is restored to the default state.

If the **hwndItem** member belongs to a **CButton**, **CMenu**, **CListBox** or **CComboBox** object, then the **DrawItem** virtual function of the appropriate class is called. Override the **DrawItem** member function of the appropriate control's class to draw the item.

DRAWITEM-STRUCT Structure

A **DRAWITEMSTRUCT** structure has this form:

```
typedef struct tagDRAWITEMSTRUCT {
    UINT   CtlType;
    UINT   CtlID;
    UINT   itemID;
    UINT   itemAction;
    UINT   itemState;
    HWND   hwndItem;
    HDC    hDC;
    RECT   rcItem;
    DWORD  itemData;
} DRAWITEMSTRUCT;
```

Members

The **DRAWITEMSTRUCT** members are as follows:

CtlType The control type. The values for control types are as follows:

- **ODT_BUTTON** Owner-draw button
- **ODT_COMBOBOX** Owner-draw combo box
- **ODT_LISTBOX** Owner-draw list box
- **ODT_MENU** Owner-draw menu

CtlID The control ID for a combo box, list box, or button. This member is not used for a menu.

itemID The menu-item ID for a menu or the index of the item in a list box or combo box. For an empty list box or combo box, this member is a negative value, which allows the application to draw only the focus rectangle at the coordinates specified by the **rcItem** member even though there are no items in the control. The user can thus be shown whether the list box or combo box has the input focus. The setting of the bits in the **itemAction** member determines whether the rectangle is to be drawn as though the list box or combo box has input focus.

itemAction Defines the drawing action required. This will be one or more of the following bits:

- **ODA_DRAWENTIRE** This bit is set when the entire control needs to be drawn.
- **ODA_FOCUS** This bit is set when the control gains or loses input focus. The **itemState** member should be checked to determine whether the control has focus.
- **ODA_SELECT** This bit is set when only the selection status has changed. **ItemState** should be checked to determine the new selection state.

itemState Specifies the visual state of the item after the current drawing action takes place. That is, if a menu item is to be dimmed, the state flag **ODS_GRAYED** will be set. The state flags are as follows:

- **ODS_CHECKED** This bit is set if the menu item is to be checked. This bit is used only in a menu.
- **ODS_DISABLED** This bit is set if the item is to be drawn as disabled.
- **ODS_FOCUS** This bit is set if the item has input focus.
- **ODS_GRAYED** This bit is set if the item is to be dimmed. This bit is used only in a menu.
- **ODS_SELECTED** This bit is set if the item's status is selected.

hwndItem Specifies the window handle of the control for combo boxes, list boxes, and buttons. Specifies the handle of the menu (**HMENU**) that contains the item for menus.

hDC Identifies a device context. This device context must be used when performing drawing operations on the control.

rcItem A rectangle in the device context specified by the **hDC** member that defines the boundaries of the control to be drawn. Windows automatically clips anything the owner draws in the device context for combo boxes, list boxes, and buttons, but it does not clip menu items. When drawing menu items, the owner must not draw outside the boundaries of the rectangle defined by the **rcItem** member.

itemData For a combo box or list box, this member contains the value that was passed to the list box by one of the following:

CComboBox::AddString
CComboBox::InsertString
CListBox::AddString
CListBox::InsertString

For a menu, this member contains the value that was passed to the menu by one of the following:

CMenu::AppendMenu
CMenu::InsertMenu
CMenu::ModifyMenu

See Also **WM_DRAWITEM**, **CButton::DrawItem**, **CMenu::DrawItem**, **CListBox::DrawItem**, **CComboBox::DrawItem**

CWnd::OnDropFiles

Windows 3.1 Only
Protected

afx_msg void OnDropFiles(HDROP *hDropInfo* **);** ♦

hDropInfo A pointer to an internal data structure that describes the dropped files. This handle is used by the **DragFinish**, **DragQueryFile**, and **DragQueryPoint** Windows functions to retrieve information about the dropped files.

Remarks Called when the user releases the left mouse button over a window that has registered itself as the recipient of dropped files. Typically, a derived class will be designed to support dropped files and it will register itself during window construction.

See Also **CWnd::DragAcceptFiles**, **WM_DROPFILES**, **::DragAcceptFiles**, **::DragFinish**, **::DragQueryFile**, **::DragQueryPoint**

CWnd::OnEnable

Protected **afx_msg void OnEnable(BOOL** *bEnable* **);** ♦

bEnable Specifies whether the **CWnd** object has been enabled or disabled. This parameter is **TRUE** if the **CWnd** has been enabled; it is **FALSE** if the **CWnd** has been disabled.

Remarks Called when an application changes the enabled state of the **CWnd** object. **OnEnable** is called before the **EnableWindow** member function returns, but after the window enabled state (**WS_DISABLED** style bit) has changed.

See Also **CWnd::EnableWindow**, **WM_ENABLE**

CWnd::OnEndSession

Protected **afx_msg void OnEndSession(BOOL** *bEnding* **);** ♦

bEnding Specifies whether or not the session is being ended. It is **TRUE** if the session is being ended; otherwise **FALSE**.

Remarks Called after the **CWnd** object has returned a nonzero value from an **OnQueryEndSession** member function call. The **OnEndSession** call informs the **CWnd** object whether the session is actually ending. If *bEnding* is **TRUE**, Windows can terminate any time after all applications have returned from processing this call. Consequently, have an application perform all tasks required for termination within **OnEndSession**. You do not need to call the **DestroyWindow** member function or **PostQuitMessage** Windows function when the session is ending.

See Also **CWnd::DestroyWindow**, **CWnd::OnQueryEndSession**, **::ExitWindows**, **::PostQuitMessage**, **WM_QUERYENDSESSION**, **CWnd::Default**, **WM_ENDSESSION**

CWnd::OnEnterIdle

Protected **afx_msg void OnEnterIdle(UINT** *nWhy*, **CWnd*** *pWho* **);** ♦

nWhy Specifies whether the message is the result of a dialog box or a menu being displayed. This parameter can be one of the following values:

- **MSGF_DIALOGBOX** The system is idle because a dialog box is being displayed.
- **MSGF_MENU** The system is idle because a menu is being displayed.

pWho Specifies a pointer to the dialog box (if *nWhy* is **MSGF_DIALOGBOX**), or the window that contains the displayed menu (if *nWhy* is **MSGF_MENU**). This pointer may be temporary and should not be stored for later use.

Remarks A call to **OnEnterIdle** informs an application's main window procedure that a modal dialog box or a menu is entering an idle state. A modal dialog box or menu enters an idle state when no messages are waiting in its queue after it has processed one or more previous messages.

See Also **WM_ENTERIDLE**

CWnd::OnEraseBkgnd

Protected **afx_msg BOOL OnEraseBkgnd(CDC*** *pDC* **);** ♦

pDC Specifies the device-context object.

Remarks Called when the **CWnd** object background needs erasing (for example, when resized). It is called to prepare an invalidated region for painting.

The default implementation erases the background using the window class background brush specified by the **hbrBackground** member of the window class structure. If the **hbrBackground** member is **NULL**, your overridden version of **OnEraseBkgnd** should erase the background color. Your version should also align the origin of the intended brush with the **CWnd** coordinates by first calling **UnrealizeObject** for the brush, and then selecting the brush.

An overridden **OnEraseBkgnd** should return nonzero in response to **WM_ERASEBKGND** if it processes the message and erases the background; this indicates that no further erasing is required. If it returns 0, the window will remain marked as needing to be erased. (Typically, this means that the **fErase** member of the **PAINTSTRUCT** structure will be **TRUE**.) Windows assumes the background is computed with the **MM_TEXT** mapping mode. If the device context is using any other mapping mode, the area erased may not be within the visible part of the client area.

Return Value Nonzero if it erases the background; otherwise 0.

See Also **WM_ICONERASEBKGND**, **CGdiObject::UnrealizeObject**, **WM_ERASEBKGND**

CWnd::OnFontChange

Protected **afx_msg void OnFontChange( );** ♦

Remarks All top-level windows in the system receive an **OnFontChange** call after the application changes the pool of font resources. An application that adds or removes fonts from the system (for example, through the **AddFontResource** or **RemoveFontResource** Windows function) should send the **WM_FONTCHANGE** message to all top-level windows. To send this message, use the **SendMessage** Windows function with the *hWnd* parameter set to 0xFFFF.

See Also **::AddFontResource**, **::RemoveFontResource**, **::SendMessage**, **WM_FONTCHANGE**

CWnd::OnGetDlgCode

Protected **afx_msg UINT OnGetDlgCode( );** ♦

Remarks Normally, Windows handles all arrow-key and TAB-key input to a **CWnd** control. By overriding **OnGetDlgCode**, a **CWnd** control can choose a particular type of input to process itself. The default **OnGetDlgCode** functions for the predefined control classes return a code appropriate for each class.

Return Value One or more of the following values, indicating which type of input the application processes:

- **DLGC_BUTTON** Button (generic).
- **DLGC_DEFPUSHBUTTON** Default pushbutton.
- **DLGC_HASSETSEL** **EM_SETSEL** messages.

- **DLGC_UNDEFPUSHBUTTON** No default pushbutton processing. (An application can use this flag with **DLGC_BUTTON** to indicate that it processes button input but relies on the system for default pushbutton processing.)
- **DLGC_RADIOBUTTON** Radio button.
- **DLGC_STATIC** Static control.
- **DLGC_WANTALLKEYS** All keyboard input.
- **DLGC_WANTARROWS** Arrow keys.
- **DLGC_WANTCHARS** **WM_CHAR** messages.
- **DLGC_WANTMESSAGE** All keyboard input. The application passes this message on to the control.
- **DLGC_WANTTAB** TAB key.

See Also **WM_GETDLGCODE**

CWnd::OnGetMinMaxInfo

Protected **afx_msg void OnGetMinMaxInfo(MINMAXINFO FAR*** *lpMMI* **);** ♦

lpMMI Points to a **MINMAXINFO** structure that contains information about a window's maximized size and position and its minimum and maximum tracking size. For more about this structure, see the "MINMAXINFO Structure" section.

Remarks

Called whenever Windows needs to know the maximized position or dimensions, or the minimum or maximum tracking size. The maximized size is the size of the window when its borders are fully extended. The maximum tracking size of the window is the largest window size that can be achieved by using the borders to size the window. The minimum tracking size of the window is the smallest window size that can be achieved by using the borders to size the window. Windows fills in an array of points specifying default values for the various positions and dimensions. The application may change these values in **OnGetMinMaxInfo**.

MINMAXINFO Structure

The **MINMAXINFO** structure has the following form:

```
typedef struct tagMINMAXINFO {
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

Members

The **MINMAXINFO** members are as follows:

ptReserved Reserved for internal use.

ptMaxSize Specifies the maximized width (**point.x**) and the maximized height (**point.y**) of the window.

ptMaxPosition Specifies the position of the left side of the maximized window (**point.x**) and the position of the top of the maximized window (**point.y**).

ptMinTrackSize Specifies the minimum tracking width (**point.x**) and the minimum tracking height (**point.y**) of the window.

ptMaxTrackSize Specifies the maximum tracking width (**point.x**) and the maximum tracking height (**point.y**) of the window.

See Also

WM_GETMINMAXINFO

CWnd::OnHScroll

Protected

afx_msg void OnHScroll(UINT *nSBCode*, **UINT** *nPos*, **CScrollBar*** *pScrollBar* **);** ♦

nSBCode Specifies a scroll-bar code that indicates the user's scrolling request. This parameter can be one of the following:

- **SB_LEFT** Scroll to far left.
- **SB_LINELEFT** Scroll left.
- **SB_LINERIGHT** Scroll right.
- **SB_PAGELEFT** Scroll one page left.
- **SB_PAGERIGHT** Scroll one page right.
- **SB_RIGHT** Scroll to far right.
- **SB_THUMBPOSITION** Scroll to absolute position. The current position is specified by the *nPos* parameter.
- **SB_THUMBTRACK** Drag scroll box to specified position. The current position is specified by the *nPos* parameter.

nPos Specifies the scroll-box position if the scroll-bar code is **SB_THUMBPOSITION** or **SB_THUMBTRACK**; otherwise not used. Depending on the initial scroll range, *nPos* may be negative and should be cast to an **int** if necessary.

pScrollBar If the scroll message came from a scroll-bar control, contains a pointer to the control. If the user clicked a window's scroll bar, this parameter is **NULL**. The pointer may be temporary and should not be stored for later use.

Remarks Called when the user clicks a window's horizontal scroll bar. The **SB_THUMBTRACK** scroll-bar code typically is used by applications that give some feedback while the scroll box is being dragged. If an application scrolls the contents controlled by the scroll bar, it must also reset the position of the scroll box with the **SetScrollPos** member function.

See Also **CWnd::SetScrollPos**, **WM_VSCROLL**, **WM_HSCROLL**

CWnd::OnHScrollClipboard

Protected **afx_msg void OnHScrollClipboard(CWnd*** *pClipAppWnd*, **UINT** *nSBCode*, **UINT** *nPos* **);** ♦

pClipAppWnd Specifies a pointer to a Clipboard-viewer window. The pointer may be temporary and should not be stored for later use.

nSBCode Specifies one of the following scroll-bar codes in the low-order word:

- **SB_BOTTOM** Scroll to lower right.
- **SB_ENDSCROLL** End scroll.
- **SB_LINEDOWN** Scroll one line down.
- **SB_LINEUP** Scroll one line up.
- **SB_PAGEDOWN** Scroll one page down.
- **SB_PAGEUP** Scroll one page up.
- **SB_THUMBPOSITION** Scroll to the absolute position. The current position is provided in *nPos*.
- **SB_TOP** Scroll to upper left.

nPos Contains the scroll-box position if the scroll-bar code is **SB_THUMBPOSITION**; otherwise not used.

Remarks The Clipboard owner's **OnHScrollClipboard** member function is called by the Clipboard viewer when the Clipboard data has the **CF_OWNERDISPLAY** format and there is an event in the Clipboard viewer's horizontal scroll bar. The owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

See Also **CWnd::OnVScrollClipboard**, **WM_HSCROLLCLIPBOARD**

CWnd::OnIconEraseBkgnd

Protected **afx_msg void OnIconEraseBkgnd(CDC*** *pDC* **);** ♦

pDC Specifies the device-context object of the icon. May be temporary and should not be stored for later use.

Remarks Called for a minimized (iconic) **CWnd** object when the background of the icon must be filled before painting the icon. **CWnd** receives this call only if a class icon is defined for the window default implementation; otherwise **OnEraseBkgnd** is called. The **DefWindowProc** member function fills the icon background with the background brush of the parent window.

See Also **CWnd::OnEraseBkgnd**, **WM_ICONERASEBKGND**

CWnd::OnInitMenu

Protected **afx_msg void OnInitMenu(CMenu*** *pMenu* **);** ♦

pMenu Specifies the menu to be initialized. May be temporary and should not be stored for later use.

Remarks Called when a menu is about to become active. The call occurs when the user clicks an item on the menu bar or presses a menu key. Override this member function to modify the menu before it is displayed. **OnInitMenu** is only called when a menu is first accessed; **OnInitMenu** is called only once for each access. This means, for example, that moving the mouse across several menu items while holding down the button does not generate new calls. This call does not provide information about menu items.

See Also **CWnd::OnInitMenuPopup**, **WM_INITMENU**

CWnd::OnInitMenuPopup

Protected **afx_msg void OnInitMenuPopup(CMenu*** *pPopupMenu*, **UINT** *nIndex*, **BOOL** *bSysMenu* **);** ♦

pPopupMenu Specifies the menu object of the pop-up menu. May be temporary and should not be stored for later use.

nIndex Specifies the index of the pop-up menu in the main menu.

bSysMenu **TRUE** if the pop-up menu is the Control menu; otherwise **FALSE**.

Remarks

Called when a pop-up menu is about to become active. This allows an application to modify the pop-up menu before it is displayed without changing the entire menu.

See Also

CWnd::OnInitMenu, **WM_INITMENUPOPUP**

CWnd::OnKeyDown

Protected

afx_msg void OnKeyDown(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the virtual-key code of the given key.

nRepCnt Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).

nFlags Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description
0–7	Scan code (OEM-dependent value).
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For a **WM_KEYDOWN** message, the key-transition bit (bit 15) is 0 and the context-code bit (bit 13) is 0.

Remarks

Called when a nonsystem key is pressed. A nonsystem key is a keyboard key that is pressed when the ALT key is not pressed or a keyboard key that is pressed when **CWnd** has the input focus. Because of auto-repeat, more than one **OnKeyDown** call may occur before an **OnKeyUp** member function call is made. The bit that indicates the previous key state can be used to determine whether the **OnKeyDown** call is the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

See Also **WM_CHAR, WM_KEYUP, WM_KEYDOWN**

CWnd::OnKeyUp

Protected **afx_msg void OnKeyUp(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the virtual-key code of the given key.

nRepCnt Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).

nFlags Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Description
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For a **WM_KEYUP** message, the key-transition bit (bit 15) is 1 and the context-code bit (bit 13) is 0.

Remarks Called when a nonsystem key is released. A nonsystem key is a keyboard key that is pressed when the ALT key is not pressed or a keyboard key that is pressed when the **CWnd** has the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

See Also **WM_CHAR**, **WM_KEYUP**, **CWnd::Default**, **WM_KEYDOWN**

CWnd::OnKillFocus

Protected **afx_msg void OnKillFocus(CWnd*** *pNewWnd* **);** ♦

pNewWnd Specifies a pointer to the window that receives the input focus (may be **NULL** or may be temporary).

Remarks Called immediately before losing the input focus. If the **CWnd** object is displaying a caret, the caret should be destroyed at this point.

See Also **CWnd::SetFocus**, **WM_KILLFOCUS**

CWnd::OnLButtonDblClk

Protected **afx_msg void OnLButtonDblClk(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user double-clicks the left mouse button. Only windows that have the **CS_DBLCLKS WNDCLASS** style will receive **OnLButtonDblClk** calls. This is the default for Microsoft Foundation class windows. Windows calls **OnLButtonDblClk** when the user presses, releases, and then presses the left mouse button again within the system's double-click time limit. Double-clicking the left mouse button actually generates four events: **WM_LBUTTONDOWN**, **WM_LBUTTONUP** messages, the **WM_LBUTTONDBLCLK** call, and another **WM_LBUTTONUP** message when the button is released.

See Also **CWnd::OnLButtonDown**, **CWnd::OnLButtonUp**, **WM_LBUTTONDBLCLK**

CWnd::OnLButtonDown

Protected **afx_msg void OnLButtonDown(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user presses the left mouse button.

See Also **CWnd::OnLButtonDblClk**, **CWnd::OnLButtonUp**, **WM_LBUTTONDOWN**

CWnd::OnLButtonUp

Protected **afx_msg void OnLButtonUp(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user releases the left mouse button.

See Also **CWnd::OnLButtonDblClk**, **CWnd::OnLButtonDown**, **WM_LBUTTONUP**

CWnd::OnMButtonDblClk

Protected **afx_msg void OnMButtonDblClk(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user double-clicks the middle mouse button. Only windows that have the **CS_DBLCLKS WNDCLASS** style will receive **OnMButtonDblClk** calls. This is the default for all Microsoft Foundation class windows. Windows generates an **OnMButtonDblClk** call when the user presses, releases, and then presses the middle mouse button again within the system's double-click time limit. Double-clicking the middle mouse button actually generates four events: **WM_MBUTTONDOWN** and **WM_MBUTTONUP** messages, the **WM_MBUTTONDBLCLK** call, and another **WM_MBUTTONUP** message.

See Also **CWnd::OnMButtonDown**, **CWnd::OnMButtonUp**, **WM_MBUTTONDBLCLK**

CWnd::OnMButtonDown

Protected **afx_msg void OnMButtonDown(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user presses the middle mouse button.

See Also **CWnd::OnMButtonDblClk**, **CWnd::OnMButtonUp**, **WM_MBUTTONDOWN**

CWnd::OnMButtonUp

Protected **afx_msg void OnMButtonUp(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user releases the middle mouse button.

See Also **CWnd::OnMButtonDblClk**, **CWnd::OnMButtonDown**, **WM_MBUTTONUP**

CWnd::OnMDIActivate

Protected

afx_msg void OnMDIActivate(BOOL *bActivate*, **CWnd*** *pActivateWnd*, **CWnd*** *pDeactivateWnd* **);** ♦

bActivate **TRUE** if the child is being activated and **FALSE** if it is being deactivated.

pActivateWnd Contains a pointer to the MDI child window to be activated. When received by an MDI child window, *pActivateWnd* contains a pointer to the child window being activated. This pointer may be temporary and should not be stored for later use.

pDeactivateWnd Contains a pointer to the MDI child window being deactivated. This pointer may be temporary and should not be stored for later use.

Remarks

Called for the child window being deactivated and the child window being activated. An MDI child window is activated independently of the MDI frame window. When the frame becomes active, the child window that was last activated with a **OnMDIActivate** call receives an **WM_NCACTIVATE** message to draw an active window frame and caption bar, but it does not receive another **OnMDIActivate** call.

See Also

CMDIFrameWnd::MDIActivate, **WM_MDIACTIVATE**

CWnd::OnMeasureItem

Protected

afx_msg void OnMeasureItem(int *nIDCtl*, **LPMEASUREITEMSTRUCT** *lpMeasureItemStruct* **);** ♦

Windows 3.1 Only

nIDCtl The ID of the control. ♦

lpMeasureItemStruct Points to a **MEASUREITEMSTRUCT** data structure that contains the dimensions of the owner-draw control.

Remarks

Called by the framework for the owner of an owner-draw button, combo box, list box, or menu item when the control is created.

Override this member function and fill in the **MEASUREITEMSTRUCT** data structure pointed to by *lpMeasureItemStruct* and return; this informs Windows of the dimensions of the control and allows Windows to process user interaction with the control correctly.

If a list box or combo box is created with the **LBS_OWNERDRAWVARIABLE** or **CBS_OWNERDRAWVARIABLE** style, the framework calls this function for the owner for each item in the control; otherwise this function is called once. Windows initiates the call to **OnMeasureItem** for the owner of combo boxes and list boxes created with the **OWNERDRAWFIXED** style before sending the **WM_INITDIALOG** message. As a result, when the owner receives this call, Windows has not yet determined the height and width of the font used in the control; function calls and calculations that require these values should occur in the main function of the application or library.

If the item being measured is a **CMenu**, **CListBox** or **CComboBox** object, then the **MeasureItem** virtual function of the appropriate class is called. Override the **MeasureItem** member function of the appropriate control's class to calculate and set the size of each item.

MEASUREITEMSTRUCT Structure

A **MEASUREITEMSTRUCT** data structure has the following form:

```
typedef struct tagMEASUREITEMSTRUCT {
    UINT   CtlType;
    UINT   CtlID;
    UINT   itemID;
    UINT   itemWidth;
    UINT   itemHeight;
    DWORD  itemData
} MEASUREITEMSTRUCT;
```

Failure to fill out the proper members in the **MEASUREITEMSTRUCT** structure will cause improper operation of the control.

Members

The **MEASUREITEMSTRUCT** members are as follows:

CtlType Contains the control type. The values for control types are as follows:

- **ODT_COMBOBOX** Owner-draw combo box
- **ODT_LISTBOX** Owner-draw list box
- **ODT_MENU** Owner-draw menu

CtlID Contains the control ID for a combo box, list box, or button. This member is not used for a menu.

itemID Contains the menu-item ID for a menu or the list-box-item ID for a variable-height combo box or list box. This member is not used for a fixed-height combo box or list box, or for a button.

itemWidth Specifies the width of a menu item. The owner of the owner-draw menu item must fill this member before it returns from the message.

itemHeight Specifies the height of an individual item in a list box or a menu. Before it returns from the message, the owner of the owner-draw combo box, list box, or menu item must fill out this member. The maximum height of a list box item is 255.

itemData For a combo box or list box, this member contains the value that was passed to the list box by one of the following:

CComboBox::AddString
CComboBox::InsertString
ListBox::AddString
ListBox::InsertString

For a menu, this member contains the value that was passed to the menu by one of the following:

CMenu::AppendMenu
CMenu::InsertMenu
CMenu::ModifyMenu

See Also **CMenu::MeasureItem**, **CListBox::MeasureItem**, **CComboBox::MeasureItem**, **WM_MEASUREITEM**

CWnd::OnMenuChar

Protected **afx_msg LRESULT OnMenuChar(UINT** *nChar*, **UINT** *nFlags*, **CMenu*** *pMenu* **);** ♦

nChar Specifies the ASCII character that the user pressed.

nFlags Contains the **MF_POPUP** flag if the menu is a pop-up menu. It contains the **MF_SYSMENU** flag if the menu is a Control menu.

pMenu Contains a pointer to the selected **CMenu**. The pointer may be temporary and should not be stored.

Remarks Called when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu. It is sent to the **CWnd** that owns the menu. **OnMenuChar** is also called when the user presses ALT and any other key, even if the key does not correspond to a mnemonic character. In this case, *pMenu* points to the menu owned by the **CWnd**, and *nFlags* is 0.

Return Value

The high-order word of the return value should contain one of the following command codes:

Value	Description
0	Tells Windows to discard the character that the user pressed and creates a short beep on the system speaker.
1	Tells Windows to close the current menu.
2	Informs Windows that the low-order word of the return value contains the item number for a specific item. This item is selected by Windows.

The low-order word is ignored if the high-order word contains 0 or 1. Applications should process this message when accelerator (shortcut) keys are used to select bitmaps placed in a menu.

See Also

WM_MENUCHAR

CWnd::OnMenuSelect

Protected

afx_msg void OnMenuSelect(UINT *nItemID*, **UINT** *nFlags*,
HMENU *hSysMenu* **);** ♦

nItemID Identifies the item selected. If the selected item is a menu item, *nItemID* contains the menu-item ID. If the selected item contains a pop-up menu, *nItemID* contains the pop-up menu handle.

nFlags Contains a combination of the following menu flags:

- **MF_BITMAP** Item is a bitmap.
- **MF_CHECKED** Item is checked.
- **MF_DISABLED** Item is disabled.
- **MF_GRAYED** Item is dimmed.
- **MF_MOUSESELECT** Item was selected with a mouse.
- **MF_OWNERDRAW** Item is an owner-draw item.
- **MF_POPUP** Item contains a pop-up menu.
- **MF_SEPARATOR** Item is a menu-item separator.
- **MF_SYSMENU** Item is contained in the Control menu.

hSysMenu If *nFlags* contains **MF_SYSMENU**, identifies the menu associated with the message; otherwise unused.

Remarks

If the **CWnd** object is associated with a menu, **OnMenuSelect** is called when the user selects a menu item. If *nFlags* contains 0xFFFF and *hSysMenu* contains 0, Windows has closed the menu because the user pressed the ESC key or clicked outside the menu.

See Also

WM_MENUSELECT

CWnd::OnMouseActivate

Protected

afx_msg int OnMouseActivate(CWnd* *pDesktopWnd*, **UINT** *nHitTest*, **UINT** *message* **);** ♦

pDesktopWnd Specifies a pointer to the top-level parent window of the window being activated. The pointer may be temporary and should not be stored.

nHitTest Specifies the hit-test area code. A hit test is a test that determines the location of the cursor.

message Specifies the mouse message number.

Remarks

Called when the cursor is in an inactive window and the user presses a mouse button. The default implementation passes this message to the parent window before any processing occurs. If the parent window returns **TRUE**, processing is halted.

For a description of the individual hit-test area codes, see the **OnNcHitTest** member function.

Return Value

Specifies whether to activate the **CWnd** and whether to discard the mouse event. It must be one of the following values:

Windows 3.1 Only

- **MA_ACTIVATE** Activate **CWnd** object.
- **MA_NOACTIVATE** Do not activate **CWnd** object.
- **MA_ACTIVATEANDEAT** Activate **CWnd** object and discard the mouse event.
- **MA_NOACTIVATEANDEAT** Do not activate **CWnd** object and discard the mouse event. ♦

See Also

CWnd::OnNcHitTest, **WM_MOUSEACTIVATE**

CWnd::OnMouseMove

Protected **afx_msg void OnMouseMove(UINT** *nFlags*, **CPoint** *point*); ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the mouse cursor moves. If the mouse is not captured, the **WM_MOUSEMOVE** message is received by the **CWnd** object beneath the mouse cursor; otherwise, the message goes to the window that has captured the mouse.

See Also **CWnd::SetCapture**, **CWnd::OnNCHitTest**, **WM_MOUSEMOVE**

CWnd::OnMove

Protected **afx_msg void OnMove(int** *x*, **int** *y*); ♦

x Specifies the new x-coordinate location of the upper-left corner of the client area. This new location is given in screen coordinates for overlapped and pop-up windows, and parent-client coordinates for child windows.

y Specifies the new y-coordinate location of the upper-left corner of the client area. This new location is given in screen coordinates for overlapped and pop-up windows, and parent-client coordinates for child windows.

Remarks Called after the **CWnd** object has been moved.

See Also **WM_MOVE**

CWnd::OnNcActivate

Protected **afx_msg BOOL OnNcActivate(BOOL** *bActive* **);** ♦

bActive Specifies when a caption bar or icon needs to be changed to indicate an active or inactive state. The *bActive* parameter is **TRUE** if an active caption or icon is to be drawn. It is **FALSE** for an inactive caption or icon.

Remarks

Called when the nonclient area needs to be changed to indicate an active or inactive state. The default implementation draws the title bar and title-bar text in their active colors if *bActive* is **TRUE** and in their inactive colors if *bActive* is **FALSE**.

Return Value

Nonzero if Windows should proceed with default processing; 0 to prevent the caption bar or icon from being deactivated.

See Also

CWnd::Default, **WM_NCACTIVATE**

CWnd::OnNcCalcSize

Protected **afx_msg void OnNcCalcSize(BOOL** *bCalcValidRects*, **NCCALCSIZE_PARAMS FAR*** *lpncsp* **);** ♦

bCalcValidRects Specifies whether the application should specify which part of the client area contains valid information. Windows will copy the valid information to the specified area within the new client area. If this parameter is **TRUE**, the application should specify which part of the client area is valid.

lpncsp Points to a **NCCALCSIZE_PARAMS** data structure that contains information an application can use to calculate the new size and position of the **CWnd** rectangle (including client area, borders, caption, scroll bars, and so on).

Remarks

Called when the size and position of the client area needs to be calculated. By processing this message, an application can control the contents of the window's client area when the size or position of the window changes.

Regardless of the value of *bCalcValidRects*, the first rectangle in the array specified by the **rgrc** structure member of the **NCCALCSIZE_PARAMS** structure contains the coordinates of the window. For a child window, the coordinates are relative to the parent window's client area. For top-level windows, the coordinates are screen coordinates. An application should modify the **rgrc[0]** rectangle to reflect the size and position of the client area. The **rgrc[1]** and **rgrc[2]** rectangles are valid only if *bCalcValidRects* is **TRUE**. In this case, the **rgrc[1]** rectangle contains the coordinates of the window before it was moved or resized.

The **rgrc[2]** rectangle contains the coordinates of the window's client area before the window was moved. All coordinates are relative to the parent window or screen.

The default implementation calculates the size of the client area based on the window characteristics (presence of scroll bars, menu, and so on), and places the result in *lpncsp*.

MCCALCSIZE_ PARAMS Structure Windows 3.1 Only

An **NCCALCSIZE_PARAMS** structure has this form:

```
typedef struct tagNCCALCSIZE_PARAMS {
    RECT          rgrc[3];
    WINDOWPOS FAR* lppos;
} NCCALCSIZE_PARAMS;
```

The **NCCALCSIZE_PARAMS** structure contains information that an application can use while processing the **WM_NCCALCSIZE** message to calculate the size, position, and valid contents of the client area of a window. ♦

Members

An **NCCALCSIZE_PARAMS** structure has the following members:

rgrc Specifies an array of rectangles. The first contains the new coordinates of a window that has been moved or resized. The second contains the coordinates of the window before it was moved or resized. The third contains the coordinates of the client area of a window before it was moved or resized. If the window is a child window, the coordinates are relative to the client area of the parent window. If the window is a top-level window, the coordinates are relative to the screen.

lppos Points to a **WINDOWPOS** structure that contains the size and position values specified in the operation that caused the window to be moved or resized.

See Also

WM_NCCALCSIZE, **CWnd::MoveWindow**, **CWnd::SetWindowPos**

CWnd::OnNcCreate

Protected

afx_msg BOOL OnNcCreate(LPCREATESTRUCT *lpCreateStruct* **);** ♦

lpCreateStruct Points to the **CREATESTRUCT** data structure for **CWnd**.

Remarks

Called prior to the **WM_CREATE** message when the **CWnd** object is first created.

Return Value

Nonzero if the nonclient area is created. It is 0 if an error occurs; the **Create** function will return **failure** in this case.

See Also

CWnd::CreateEx, **WM_NCCREATE**

CWnd::OnNcDestroy

Protected **afx_msg void OnNcDestroy( );** ♦

Remarks Called by the framework when the nonclient area is being destroyed, and is the last member function called when the Windows window is destroyed. The default implementation performs some cleanup, then calls the virtual member function **PostNcDestroy**. Override **PostNcDestroy** if you want to perform your own cleanup, such as a **delete this** operation. If you override **OnNcDestroy**, you must call **OnNcDestroy** in your base class to ensure that any memory internally allocated for the window is freed.

See Also **CWnd::DestroyWindow**, **CWnd::OnNcCreate**, **WM_NCDESTROY**, **CWnd::Default**, **CWnd::PostNcDestroy**

CWnd::OnNcHitTest

Protected **afx_msg UINT OnNcHitTest(CPoint** *point* **);** ♦

point Contains the x- and y-coordinates of the cursor. These coordinates are always screen coordinates.

Remarks Called for the **CWnd** object that contains the cursor (or the **CWnd** object that used the **SetCapture** member function to capture the mouse input) every time the mouse is moved.

Return Value One of the following values, which indicate the current mouse position:

- **HTBORDER** In the border of a window that does not have a sizing border.
- **HTBOTTOM** In the lower horizontal border of the window.
- **HTBOTTOMLEFT** In the lower-left corner of the window border.
- **HTBOTTOMRIGHT** In the lower-right corner of the window border.
- **HTCAPTION** In a title-bar area.
- **HTCLIENT** In a client area.
- **HTERROR** On the screen background or on a dividing line between windows (same as **HTNOWHERE** except that the **DefWndProc** Windows function produces a system beep to indicate an error).
- **HTGROWBOX** In a size box.
- **HTHSCROLL** In the horizontal scroll bar.
- **HTLEFT** In the left border of the window.

- **HTMAXBUTTON** In a Maximize button.
- **HTMENU** In a menu area.
- **HTMINBUTTON** In a Minimize button.
- **HTNOWHERE** On the screen background or on a dividing line between windows.
- **HTREDUCE** In a Minimize button.
- **HTRIGHT** In the right border of the window.
- **HTSIZE** In a size box (same as **HTGROWBOX**).
- **HTSYSMENU** In a Control menu or in a Close button in a child window.
- **HTTOP** In the upper horizontal border of the window.
- **HTTOPLEFT** In the upper-left corner of the window border.
- **HTTOPRIGHT** In the upper-right corner of the window border.
- **HTTRANSPARENT** In a window currently covered by another window.
- **HTVSCROLL** In the vertical scroll bar.
- **HTZOOM** In a Maximize button.

See Also **CWnd::GetCapture**, **WM_NCHITTEST**

CWnd::OnNcLButtonDblClk

Protected **afx_msg void OnNcLButtonDblClk(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user double-clicks the left mouse button while the cursor is within a nonclient area of **CWnd**. If appropriate, the **WM_SYSCOMMAND** message is sent.

See Also **WM_NCLBUTTONDBLCLK**, **CWnd::OnNcHitTest**

CWnd::OnNcLButtonDown

Protected **afx_msg void OnNcLButtonDown(UINT** *nHitTest*, **CPoint** *point*); ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user presses the left mouse button while the cursor is within a nonclient area of the **CWnd** object. If appropriate, the **WM_SYSCOMMAND** is sent.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcLButtonDblClk**, **CWnd::OnNcLButtonUp**, **CWnd::OnSysCommand**, **WM_NCLBUTTONDOWN**, **CWnd::Default**

CWnd::OnNcLButtonUp

Protected **afx_msg void OnNcLButtonUp(UINT** *nHitTest*, **CPoint** *point*); ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user releases the left mouse button while the cursor is within a nonclient area. If appropriate, **WM_SYSCOMMAND** is sent.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcLButtonDown**, **CWnd::OnSysCommand**, **WM_NCLBUTTONUP**

CWnd::OnNcMButtonDblClk

Protected **afx_msg void OnNcMButtonDblClk(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user double-clicks the middle mouse button while the cursor is within a nonclient area.

See Also **CWnd::OnNcHitTest, CWnd::OnNcMButtonDown, CWnd::OnNcMButtonUp, WM_NCMBUTTONDBLCLK**

CWnd::OnNcMButtonDown

Protected **afx_msg void OnNcMButtonDown(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user presses the middle mouse button while the cursor is within a nonclient area.

See Also **CWnd::OnNcHitTest, CWnd::OnNcMButtonDblClk, CWnd::OnNcMButtonUp, WM_NCMBUTTONDOWN**

CWnd::OnNcMButtonUp

Protected **afx_msg void OnNcMButtonUp(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user releases the middle mouse button while the cursor is within a nonclient area.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcMButtonDblClk**, **CWnd::OnNcMButtonDown**, **WM_NCMBUTTONUP**

CWnd::OnNcMouseMove

Protected **afx_msg void OnNcMouseMove(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the cursor is moved within a nonclient area. If appropriate, the **WM_SYSCOMMAND** message is sent.

See Also **CWnd::OnNcHitTest**, **CWnd::OnSysCommand**, **WM_NCMOUSEMOVE**

CWnd::OnNcPaint

Protected **afx_msg void OnNcPaint( );** ♦

Remarks Called when the nonclient area needs to be painted. The default implementation paints the window frame. An application can override this call and paint its own custom window frame. The clipping region is always rectangular, even if the shape of the frame is altered.

See Also **WM_NCPAINT**

CWnd::OnNcRButtonDblClk

Protected **afx_msg void OnNcRButtonDblClk(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test determines the cursor's location.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user double-clicks the right mouse button while the cursor is within a nonclient area of **CWnd**.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcRButtonDown**, **CWnd::OnNcRButtonUp**, **WM_NCRBUTTONDBLCLK**

CWnd::OnNcRButtonDown

Protected **afx_msg void OnNcRButtonDown(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test determines the cursor's location.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user presses the right mouse button while the cursor is within a nonclient area.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcRButtonDblClk**, **CWnd::OnNcRButtonUp**, **WM_NCRBUTTONDOWN**

CWnd::OnNcRButtonUp

Protected **afx_msg void OnNcRButtonUp(UINT** *nHitTest*, **CPoint** *point* **);** ♦

nHitTest Specifies the hit-test code. A hit test determines the cursor's location.

point Specifies a **CPoint** object that contains the x- and y-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Remarks Called when the user releases the right mouse button while the cursor is within a nonclient area.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcRButtonDblClk**, **CWnd::OnNcRButtonDown**, **WM_NCRBUTTONUP**

CWnd::OnPaint

Protected **afx_msg void OnPaint( );** ♦

Remarks Called when Windows or an application makes a request to repaint a portion of an application's window. The **WM_PAINT** message is sent when the **UpdateWindow** or **RedrawWindow** member function is called.

Windows 3.1 Only A window may receive internal paint messages as a result of calling the **RedrawWindow** member function with the **RDW_INTERNALPAINT** flag set. In this case, the window may not have an update region. An application should call the **GetUpdateRect** member function to determine whether the window has an update region. If **GetUpdateRect** returns 0, the application should not call the **BeginPaint** and **EndPaint** member functions.

It is an application's responsibility to check for any necessary internal repainting or updating by looking at its internal data structures for each **WM_PAINT** message because a **WM_PAINT** message may have been caused by both an invalid area and a call to the **RedrawWindow** member function with the **RDW_INTERNALPAINT** flag set. An internal **WM_PAINT** message is sent only once by Windows. After an internal **WM_PAINT** message is sent to a window by the **UpdateWindow** member function, no further **WM_PAINT** messages will be sent or posted until the window is invalidated or until the **RedrawWindow** member function is called again with the **RDW_INTERNALPAINT** flag set. ♦

See Also **CWnd::BeginPaint**, **CWnd::EndPaint**, **CWnd::RedrawWindow**, **CPaintDC**

CWnd::OnPaintClipboard

Protected **afx_msg void OnPaintClipboard(CWnd*** *pClipAppWnd*, **HGLOBAL** *hPaintStruct* **);** ♦

pClipAppWnd Specifies a pointer to the Clipboard-application window. The pointer may be temporary and should not be stored for later use.

hPaintStruct Identifies a **PAINTSTRUCT** data structure that defines what part of the client area to paint.

Remarks

A Clipboard owner's **OnPaintClipboard** member function is called by a Clipboard viewer when the Clipboard owner has placed data on the Clipboard in the **CF_OWNERDISPLAY** format and the Clipboard viewer's client area needs repainting. To determine whether the entire client area or just a portion of it needs repainting, the Clipboard owner must compare the dimensions of the drawing area given in the **rcpaint** member of the **PAINTSTRUCT** structure to the dimensions given in the most recent **OnSizeClipboard** member function call.

OnPaintClipboard should use the **GlobalLock** Windows function to lock the memory that contains the **PAINTSTRUCT** data structure and unlock that memory with the **GlobalUnlock** Windows function before it exits.

See Also

::GlobalLock, **::GlobalUnlock**, **CWnd::OnSizeClipboard**, **WM_PAINTCLIPBOARD**

CWnd::OnPaletteChanged

Protected

afx_msg void OnPaletteChanged(CWnd* *pFocusWnd* **);** ♦

pFocusWnd Specifies a pointer to the window that caused the system palette to change. The pointer may be temporary and should not be stored.

Remarks

Called for all top-level windows after the window with input focus has realized its logical palette thereby changing the system palette. This call allows a window without the input focus that uses a color palette to realize its logical palette and update its client area. The **OnPaletteChanged** member function is called for all top-level and overlapped windows, including the one that changed the system palette and caused the **WM_PALETTECHANGED** message to be sent. If any child window uses a color palette, this message must be passed on to it. To avoid an infinite loop, the window shouldn't realize its palette unless it determines that *pFocusWnd* does not contain a pointer to itself.

See Also

::RealizePalette, **WM_PALETTECHANGED**, **CWnd::OnPaletteIsChanging**, **CWnd::OnQueryNewPalette**

CWnd::OnPaletteIsChanging

Windows 3.1 Only
Protected

afx_msg void OnPaletteIsChanging(CWnd* *pRealizeWnd* **);** ♦

pRealizeWnd Specifies the window that is about to realize its logical palette.

Remarks

Informs applications that an application is going to realize its logical palette.

See Also

CWnd::OnPaletteChanged, **CWnd::OnQueryNewPalette**, **::OnPaletteIsChanging**

CWnd::OnParentNotify

Protected

afx_msg void OnParentNotify(UINT *message*, **LPARAM** *lParam* **);** ♦

message Specifies the event for which the parent is being notified. It can be any of these values:

- **WM_CREATE** The child window is being created.
- **WM_DESTROY** The child window is being destroyed.
- **WM_LBUTTONDOWN** The user has placed the mouse cursor over the child window and clicked the left mouse button.
- **WM_MBUTTONDOWN** The user has placed the mouse cursor over the child window and clicked the middle mouse button.
- **WM_RBUTTONDOWN** The user has placed the mouse cursor over the child window and clicked the right mouse button.

lParam If *message* is **WM_CREATE** or **WM_DESTROY**, specifies the window handle of the child window in the low-order word and the identifier of the child window in the high-order word; otherwise *lParam* contains the x- and y-coordinates of the cursor. The x-coordinate is in the low-order word and the y-coordinate is in the high-order word.

Remarks

A parent's **OnParentNotify** member function is called when its child window is created or destroyed, or when the user clicks a mouse button while the cursor is over the child window. When the child window is being created, the system calls **OnParentNotify** just before the **Create** member function that creates the window returns. When the child window is being destroyed, the system calls **OnParentNotify** before any processing takes place to destroy the window. **OnParentNotify** is called for all ancestor windows of the child window, including the top-level window.

All child windows except those that have the **WS_EX_NOPARENTNOTIFY** style send this message to their parent windows. By default, child windows in a dialog box have the **WS_EX_NOPARENTNOTIFY** style unless the child window was created without this style by calling the **CreateEx** member function.

See Also **CWnd::OnCreate**, **CWnd::OnDestroy**, **CWnd::OnLButtonDown**, **CWnd::OnMButtonDown**, **CWnd::OnRButtonDown**, **WM_PARENTNOTIFY**

CWnd::OnQueryDragIcon

Protected **afx_msg HCURSOR OnQueryDragIcon( );** ♦

Remarks Called by a minimized (iconic) window that does not have an icon defined for its class. The system makes this call to obtain the cursor to display while the user drags the minimized window. If an application returns the handle of an icon or cursor, the system converts it to black-and-white. If an application returns a handle, the handle must identify a monochrome cursor or icon compatible with the display driver's resolution. The application can call the **CWinApp::LoadCursor** or **CWinApp::LoadIcon** member functions to load a cursor or icon from the resources in its executable file and to obtain this handle.

Return Value A doubleword value that contains a cursor or icon handle in the low-order word. The cursor or icon must be compatible with the display driver's resolution. If the application returns **NULL**, the system displays the default cursor. The default return value is **NULL**.

See Also **CWinApp::LoadCursor**, **CWinApp::LoadIcon**, **WM_QUERYDRAGICON**

CWnd::OnQueryEndSession

Protected **afx_msg BOOL OnQueryEndSession( );** ♦

Remarks Called when the user chooses to end the Windows session or when an application calls the **ExitWindows** Windows function. If any application returns 0, the Windows session is not ended. Windows stops calling **OnQueryEndSession** as soon as one application returns 0 and sends the **WM_ENDSESSION** message with a parameter value of **FALSE** for any application that has already returned nonzero.

Return Value Nonzero if an application can be conveniently shut down; otherwise 0.

See Also **::ExitWindows**, **CWnd::OnEndSession**, **WM_QUERYENDSESSION**

CWnd::OnQueryNewPalette

Protected **afx_msg BOOL OnQueryNewPalette( );** ♦

Remarks Called when the **CWnd** object is about to receive the input focus, giving the **CWnd** an opportunity to realize its logical palette when it receives the focus.

Return Value Nonzero if the **CWnd** realizes its logical palette; otherwise 0.

See Also **CWnd::Default**, **CWnd::OnPaletteChanged**, **WM_QUERYNEWPALETTE**

CWnd::OnQueryOpen

Protected **afx_msg BOOL OnQueryOpen( );** ♦

Remarks Called when the **CWnd** object is minimized and the user requests that the **CWnd** be restored to its preminimized size and position. While in **OnQueryOpen**, **CWnd** should not perform any action that would cause an activation or focus change (for example, creating a dialog box).

Return Value Nonzero if the icon can be opened, or 0 to prevent the icon from being opened.

See Also **WM_QUERYOPEN**

CWnd::OnRButtonDblClk

Protected **afx_msg void OnRButtonDblClk(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinates of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Called when the user double-clicks the right mouse button. Only windows that have the **CS_DBLCLKS WNDCLASS** style can receive **OnRButtonDblClk** calls. This is the default for windows within the Microsoft Foundation Class Library. Windows calls **OnRButtonDblClk** when the user presses, releases, and then again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four events: **WM_RBUTTONDOWN** and **WM_RBUTTONUP** messages, the **OnRButtonDblClk** call, and another **WM_RBUTTONUP** message when the button is released.

See Also

CWnd::OnRButtonDown, CWnd::OnRButtonUp, WM_RBUTTONDBLCLK

CWnd::OnRButtonDown

Protected

afx_msg void OnRButtonDown(UINT *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_RBUTTON** Set if the right mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinates of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks

Called when the user presses the right mouse button.

See Also

CWnd::OnRButtonDblClk, CWnd::OnRButtonUp, WM_RBUTTONDOWN

CWnd::OnRButtonUp

Protected **afx_msg void OnRButtonUp(UINT** *nFlags*, **CPoint** *point* **);** ♦

nFlags Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

- **MK_CONTROL** Set if the CTRL key is down.
- **MK_LBUTTON** Set if the left mouse button is down.
- **MK_MBUTTON** Set if the middle mouse button is down.
- **MK_SHIFT** Set if the SHIFT key is down.

point Specifies the x- and y-coordinates of the cursor. These coordinates are always relative to the upper-left corner of the window.

Remarks Called when the user releases the right mouse button.

See Also **CWnd::OnRButtonDblClk**, **CWnd::OnRButtonDown**, **WM_RBUTTONUP**

CWnd::OnRenderAllFormats

Protected **afx_msg void OnRenderAllFormats( );** ♦

Remarks The Clipboard owner's **OnRenderAllFormats** member function is called when the owner application is being destroyed. The Clipboard owner should render the data in all the formats it is capable of generating and pass a data handle for each format to the Clipboard by calling the **SetClipboardData** Windows function. This ensures that the Clipboard contains valid data even though the application that rendered the data is destroyed. The application should call the **OpenClipboard** member function before calling the **SetClipboardData** Windows function and call the **CloseClipboard** Windows function afterward.

See Also **::CloseClipboard**, **CWnd::OpenClipboard**, **::SetClipboardData**, **CWnd::OnRenderFormat**, **WM_RENDERALLFORMATS**

CWnd::OnRenderFormat

Protected **afx_msg void OnRenderFormat(UINT** *nFormat* **);** »

nFormat Specifies the Clipboard format.

Remarks The Clipboard owner's **OnRenderFormat** member function is called when a particular format with delayed rendering needs to be rendered. The receiver should render the data in that format and pass it to the Clipboard by calling the **SetClipboardData** Windows function. Do not call the **OpenClipboard** member function or the **CloseClipboard** Windows function from within **OnRenderFormat**.

See Also **::CloseClipboard**, **CWnd::OpenClipboard**, **::SetClipboardData**, **WM_RENDERFORMAT**

CWnd::OnSetCursor

Protected **afx_msg BOOL OnSetCursor(CWnd*** *pWnd*, **UINT** *nHitTest*, **UINT** *message* **);** »

pWnd Specifies a pointer to the window that contains the cursor. The pointer may be temporary and should not be stored for later use.

nHitTest Specifies the hit-test area code. The hit test determines the cursor's location.

message Specifies the mouse message number.

Remarks Called if mouse input is not captured and the mouse causes cursor movement within the **CWnd** object. The default implementation calls the parent window's **OnSetCursor** before processing. If the parent window returns **TRUE**, further processing is halted. Calling the parent window gives the parent window control over the cursor's setting in a child window. The default implementation sets the cursor to an arrow if it is not in the client area or to the registered-class cursor if it is.

If *nHitTest* is **HTERROR** and *message* is a mouse button-down message, the **MessageBeep** member function is called. The *message* parameter is 0 when **CWnd** enters menu mode.

Return Value Nonzero to halt further processing, or 0 to continue.

See Also **CWnd::OnNcHitTest**, **WM_SETCURSOR**

CWnd::OnSetFocus

Protected **afx_msg void OnSetFocus(CWnd*** *pOldWnd* **);** ♦

pOldWnd Contains the **CWnd** object that loses the input focus (may be **NULL**). The pointer may be temporary and should not be stored for later use.

Remarks Called after gaining the input focus. To display a caret, **CWnd** should call the appropriate caret functions at this point.

See Also **WM_SETFOCUS**

CWnd::OnShowWindow

Protected **afx_msg void OnShowWindow(BOOL** *bShow*, **UINT** *nStatus* **);** ♦

bShow Specifies whether a window is being shown. It is **TRUE** if the window is being shown; it is **FALSE** if the window is being hidden.

nStatus Specifies the status of the window being shown. It is 0 if the message is sent because of a **ShowWindow** member function call; otherwise *nStatus* is one of the following:

- **SW_PARENTCLOSING** Parent window is closing (being made iconic) or a pop-up window is being hidden.
- **SW_PARENTOPENING** Parent window is opening (being displayed) or a pop-up window is being shown.

Remarks Called when the **CWnd** object is about to be hidden or shown. A window is hidden or shown when the **ShowWindow** member function is called, when an overlapped window is maximized or restored, or when an overlapped or pop-up window is closed (made iconic) or opened (displayed on the screen). When an overlapped window is closed, all pop-up windows associated with that window are hidden.

See Also **WM_SHOWWINDOW**

CWnd::OnSize

Protected

afx_msg void OnSize(UINT *nType*, **int** *cx*, **int** *cy* **);** ♦

nType Specifies the type of resizing requested. This parameter can be one of the following values:

- **SIZE_MAXIMIZED** Window has been maximized.
- **SIZE_MINIMIZED** Window has been minimized.
- **SIZE_RESTORED** Window has been resized, but neither **SIZE_MINIMIZED** nor **SIZE_MAXIMIZED** applies.
- **SIZE_MAXHIDE** Message is sent to all pop-up windows when some other window is maximized.
- **SIZE_MAXSHOW** Message is sent to all pop-up windows when some other window has been restored to its former size.

cx Specifies the new width of the client area.

cy Specifies the new height of the client area.

Remarks

Called after the window's size has changed. If the **SetScrollPos** or **MoveWindow** member function is called for a child window from **OnSize**, the *bRedraw* parameter of **SetScrollPos** or **MoveWindow** should be nonzero to cause the **CWnd** to be repainted.

See Also

CWnd::MoveWindow, **CWnd::SetScrollPos**, **WM_SIZE**

CWnd::OnSizeClipboard

Protected

afx_msg void OnSizeClipboard(CWnd* *pClipAppWnd*, **HGLOBAL** *hRect* **);** ♦

pClipAppWnd Identifies the Clipboard-application window. The pointer may be temporary and should not be stored.

hRect Identifies a global memory object. The memory object contains a **RECT** data structure that specifies the area for the Clipboard owner to paint.

Remarks

The Clipboard owner's **OnSizeClipboard** member function is called by the Clipboard viewer when the Clipboard contains data with the **CF_OWNERDISPLAY** attribute and the size of the client area of the Clipboard-viewer window has changed. The **OnSizeClipboard** member function is called with a null rectangle (0,0,0,0) as the new size when the Clipboard application is about to be destroyed or minimized. This permits the Clipboard owner to free its display resources. Within **OnSizeClipboard**, an application must use the **GlobalLock** Windows function to lock the memory that contains the **RECT** data structure. Have the application unlock that memory with the **GlobalUnlock** Windows function before it yields or returns control.

See Also

::GlobalLock, **::GlobalUnlock**, **::SetClipboardData**, **CWnd::SetClipboardViewer**, **WM_SIZECLIPBOARD**

CWnd::OnSpoolerStatus

Protected

afx_msg void OnSpoolerStatus(UINT *nStatus*, **UINT** *nJobs* **);** ♦

nStatus Specifies the **SP_JOBSTATUS** flag.

nJobs Specifies the number of jobs remaining in the Print Manager queue.

Remarks

Called from Print Manager whenever a job is added to or removed from the Print Manager queue. This call is for informational purposes only.

See Also

WM_SPOOLERSTATUS

CWnd::OnSysChar

Protected

afx_msg void OnSysChar(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the ASCII-character key code of a Control-menu key.

nRepCnt Specifies the repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).

nFlags The *nFlags* parameter can have these values:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

Remarks

Called if **CWnd** has the input focus and the **WM_SYSKEYUP** and **WM_SYSKEYDOWN** messages are translated. It specifies the virtual-key code of the Control-menu key. When the context code is 0, **WM_SYSCHAR** can pass the **WM_SYSCHAR** message to the **TranslateAccelerator** Windows function, which will handle it as though it were a normal key message instead of a Control-menu key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

See Also

::TranslateAccelerator, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**, **WM_SYSCHAR**

CWnd::OnSysColorChange

Protected

afx_msg void OnSysColorChange( ); ♦

Remarks

Called for all top-level windows when a change is made in the system color setting. Windows calls **OnSysColorChange** for any window that is affected by a system color change. Applications that have brushes that use the existing system colors should delete those brushes and re-create them with the new system colors.

See Also

::SetSysColors, **WM_SYSCOLORCHANGE**

CWnd::OnSysCommand

Protected **afx_msg void OnSysCommand(UINT** *nID*, **LPARAM** *lParam* **);** ♦

nID Specifies the type of system command requested. This parameter can be one of the following values, with meanings as given:

- **SC_CLOSE** Close the **CWnd** object.
- **SC_HOTKEY** Activate the **CWnd** object associated with the application-specified hot key. The low-order word of *lParam* identifies the **HWND** of the window to activate.
- **SC_HSCROLL** Scroll horizontally.
- **SC_KEYMENU** Retrieve a menu through a keystroke.
- **SC_MAXIMIZE** (or **SC_ZOOM**) Maximize the **CWnd** object.
- **SC_MINIMIZE** (or **SC_ICON**) Minimize the **CWnd** object.
- **SC_MOUSEMENU** Retrieve a menu through a mouse click.
- **SC_MOVE** Move the **CWnd** object.
- **SC_NEXTWINDOW** Move to the next window.
- **SC_PREVWINDOW** Move to the previous window.
- **SC_RESTORE** Restore window to normal position and size.
- **SC_SCREENSAVE** Executes the screen-saver application specified in the [boot] section of the SYSTEM.INI file.
- **SC_SIZE** Size the **CWnd** object.
- **SC_TASKLIST** Execute or activate the Windows Task Manager application.
- **SC_VSCROLL** Scroll vertically.

Windows 3.1 Only

- **SC_HOTKEY** Activate the window associated with the application-specified hot key. The low-order word of *lParam* identifies the window to activate.
- **SC_SCREENSAVE** Execute the screen-save application specified in the Desktop section of Control Panel. ♦

lParam If a Control-menu command is chosen with the mouse contains the cursor coordinates. The low-order word contains the x-coordinate, and the high-order word contains the y-coordinate. Otherwise this parameter is not used.

Remarks

Called when the user selects a command from the Control menu, or when the user selects the Maximize or the Minimize button. By default, **OnSysCommand** carries out the Control-menu request for the predefined actions specified in the preceding table. In **WM_SYSCOMMAND** messages, the four low-order bits of the *nID* parameter are used internally by Windows. When an application tests the value of *nID*, it must combine the value 0xFFF0 with the *nID* value by using the bitwise-AND operator to obtain the correct result.

The menu items in a Control menu can be modified with the **GetSystemMenu**, **AppendMenu**, **InsertMenu**, and **ModifyMenu** member functions. Applications that modify the Control menu must process **WM_SYSCOMMAND** messages, and any **WM_SYSCOMMAND** messages not handled by the application must be passed on to **OnSysCommand**. Any command values added by an application must be processed by the application and cannot be passed to **OnSysCommand**.

An application can carry out any system command at any time by passing a **WM_SYSCOMMAND** message to **OnSysCommand**. Accelerator (shortcut) keystrokes that are defined to select items from the Control menu are translated into **OnSysCommand** calls; all other accelerator keystrokes are translated into **WM_COMMAND** messages.

See Also **WM_SYSCOMMAND**

CWnd::OnSysDeadChar

Protected **afx_msg void OnSysDeadChar(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the dead-key character value.

nRepCnt Specifies the repeat count.

nFlags Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed; otherwise 0).
14	Previous key state (1 if the key is down before the call, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

Remarks

Called if the **CWnd** object has the input focus when the **OnSysKeyUp** or **OnSysKeyDown** member function is called. It specifies the character value of a dead key.

See Also

CWnd::OnSysKeyDown, CWnd::OnSysKeyUp, WM_SYSDEADCHAR, CWnd::OnDeadChar

CWnd::OnSysKeyDown

Protected

afx_msg void OnSysKeyDown(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the virtual-key code of the key being pressed.

nRepCnt Specifies the repeat count.

nFlags Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
14	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For **OnSysKeyDown** calls, the key-transition bit (bit 15) is 0. The context-code bit (bit 13) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Remarks

If the **CWnd** object has the input focus, the **OnSysKeyDown** member function is called when the user holds down the ALT key and then presses another key. If no window currently has the input focus, the active window's **OnSysKeyDown** member function is called. The **CWnd** object that receives the message can distinguish between these two contexts by checking the context code in *nFlags*. When the context code is 0, the **WM_SYSKEYDOWN** message received by **OnSysKeyDown** can be passed to the **TranslateAccelerator** Windows function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

Because of auto-repeat, more than one **OnSysKeyDown** call may occur before the **WM_SYSKEYUP** message is received. The previous key state (bit 14) can be used to determine whether the **OnSysKeyDown** call indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

See Also **::TranslateAccelerator**, **WM_SYSKEYUP**, **WM_SYSKEYDOWN**

CWnd::OnSysKeyUp

Protected **afx_msg void OnSysKeyUp(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);** ♦

nChar Specifies the virtual-key code of the key being pressed.

nRepCnt Specifies the repeat count.

nFlags Specifies the scan code, key-transition code, previous key state, and context code, as shown in the following list:

Value	Meaning
0–7	Scan code (OEM-dependent value). Low byte of high-order word.
8	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key; otherwise 0).
9–10	Not used.
11–12	Used internally by Windows.
13	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
14	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
15	Transition state (1 if the key is being released, 0 if the key is being pressed).

For **OnSysKeyUp** calls, the key-transition bit (bit 15) is 1. The context-code bit (bit 13) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Remarks

If the **CWnd** object has the focus, the **OnSysKeyUp** member function is called when the user releases a key that was pressed while the ALT key was held down. If no window currently has the input focus, the active window's **OnSysKeyUp** member function is called. The **CWnd** object that receives the call can distinguish between these two contexts by checking the context code in *nFlags*. When the context code is 0, the **WM_SYSKEYUP** message received by **OnSysKeyUp** can be passed to the **TranslateAccelerator** Windows function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator (shortcut) keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the slash (/) and ENTER keys in the numeric keypad. Some other keyboards may support the extended-key bit in *nFlags*.

For non-U.S. Enhanced 102-key keyboards, the right ALT key is handled as the CTRL+ALT key combination. The following shows the sequence of messages and calls that result when the user presses and releases this key:

Sequence	Function Accessed	Message Passed
1.	**WM_KEYDOWN**	**VK_CONTROL**
2.	**WM_KEYDOWN**	**VK_MENU**
3.	**WM_KEYUP**	**VK_CONTROL**
4.	**WM_SYSKEYUP**	**VK_MENU**

See Also

::TranslateAccelerator, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**

CWnd::OnTimeChange

Protected

afx_msg void OnTimeChange( ); ♦

Remarks

Called after the system time is changed. Have any application that changes the system time send this message to all top-level windows. To send the **WM_TIMECHANGE** message to all top-level windows, an application can use the **SendMessage** Windows function with its *hwnd* parameter set to **HWND_BROADCAST**.

See Also

::SendMessage, **WM_TIMECHANGE**

CWnd::OnTimer

Protected **afx_msg void OnTimer(UINT** *nIDEvent* **);** ♦

nIDEvent Specifies the identifier of the timer.

Remarks Called after each interval specified in the **SetTimer** member function used to install a timer. The **DispatchMessage** Windows function sends a **WM_TIMER** message when no other messages are in the application's message queue.

See Also **CWnd::SetTimer, WM_TIMER**

CWnd::OnVKeyToItem

Protected **afx_msg int OnVKeyToItem(UINT** *nKey*, **CListBox*** *pListBox*, **UINT** *nIndex* **);** ♦

nKey Specifies the virtual-key code of the key that the user pressed.

pListBox Specifies a pointer to the list box. The pointer may be temporary and should not be stored for later use.

nIndex Specifies the current caret position.

Remarks If the **CWnd** object owns a list box with the **LBS_WANTKEYBOARDINPUT** style, the list box will send the **WM_VKEYTOITEM** message in response to a **WM_KEYDOWN** message. This member function is called only for list boxes that have the **LBS_HASSTRINGS** style.

Return Value Specifies the action that the application performed in response to the message. A return value of –2 indicates that the application handled all aspects of selecting the item and requires no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the keystroke. A return value of 0 or greater specifies the zero-based index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

See Also **WM_KEYDOWN, WM_VKEYTOITEM**

CWnd::OnVScroll

Protected

afx_msg void OnVScroll(UINT *nSBCode*, **UINT** *nPos*,
CScrollBar* *pScrollBar* **);** ♦

nSBCode Specifies a scroll-bar code that indicates the user's scrolling request. This parameter can be one of the following:

- **SB_BOTTOM** Scroll to bottom.
- **SB_ENDSCROLL** End scroll.
- **SB_LINEDOWN** Scroll one line down.
- **SB_LINEUP** Scroll one line up.
- **SB_PAGEDOWN** Scroll one page down.
- **SB_PAGEUP** Scroll one page up.
- **SB_THUMBPOSITION** Scroll to the absolute position. The current position is provided in *nPos*.
- **SB_THUMBTRACK** Drag scroll box to specified position. The current position is provided in *nPos*.
- **SB_TOP** Scroll to top.

nPos Contains the current scroll-box position if the scroll-bar code is **SB_THUMBPOSITION** or **SB_THUMBTRACK**; otherwise not used. Depending on the initial scroll range, *nPos* may be negative and should be cast to an **int** if necessary.

pScrollBar If the scroll message came from a scroll-bar control, contains a pointer to the control. If the user clicked a window's scroll bar, this parameter is **NULL**. The pointer may be temporary and should not be stored for later use.

Remarks

Called when the user clicks the window's vertical scroll bar. **OnVScroll** typically is used by applications that give some feedback while the scroll box is being dragged. If **OnVScroll** scrolls the contents of the **CWnd** object, it must also reset the position of the scroll box with the **SetScrollPos** member function.

See Also

CWnd::SetScrollPos, **CWnd::OnHScroll**, **WM_VSCROLL**

CWnd::OnVScrollClipboard

Protected

afx_msg void OnVScrollClipboard(CWnd* *pClipAppWnd*, **UINT** *nSBCode*, **UINT** *nPos* **);** ♦

pClipAppWnd Specifies a pointer to a Clipboard-viewer window. The pointer may be temporary and should not be stored for later use.

nSBCode Specifies one of the following scroll-bar values:

- **SB_BOTTOM** Scroll to bottom.
- **SB_ENDSCROLL** End scroll.
- **SB_LINEDOWN** Scroll one line down.
- **SB_LINEUP** Scroll one line up.
- **SB_PAGEDOWN** Scroll one page down.
- **SB_PAGEUP** Scroll one page up.
- **SB_THUMBPOSITION** Scroll to the absolute position. The current position is provided in *nPos*.
- **SB_TOP** Scroll to top.

nPos Contains the scroll-box position if the scroll-bar code is **SB_THUMBPOSITION**; otherwise *nPos* is not used.

Remarks

The Clipboard owner's **OnVScrollClipboard** member function is called by the Clipboard viewer when the Clipboard data has the **CF_OWNERDISPLAY** format and there is an event in the Clipboard viewer's vertical scroll bar. The owner should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

See Also

CWnd::Invalidate, **CWnd::OnHScrollClipboard**, **CWnd::InvalidateRect**, **WM_VSCROLLCLIPBOARD**, **CWnd::Default**

CWnd::OnWindowPosChanged

Windows 3.1 Only
Protected

afx_msg void OnWindowPosChanged(WINDOWPOS FAR* *lpwndpos* **);** ♦

lpwndpos Points to a **WINDOWPOS** data structure that contains information about the window's new size and position.

Remarks

Called when the size, position, or Z-order has changed as a result of a call to the **SetWindowPos** member function or another window-management function. The default implementation sends the **WM_SIZE** and **WM_MOVE** messages to the window. These messages are not sent if an application handles the **OnWindowPosChanged** call without calling its base class. It is more efficient to perform any move or size change processing during the call to **OnWindowPosChanged** without calling its base class.

See Also

WM_WINDOWPOSCHANGED

CWnd::OnWindowPosChanging

Windows 3.1 Only
Protected

afx_msg void OnWindowPosChanging(WINDOWPOS FAR* *lpwndpos* **);** ♦

lpwndpos Points to a **WINDOWPOS** data structure that contains information about the window's new size and position.

Remarks

Called when the size, position, or Z-order is about to change as a result of a call to the **SetWindowPos** member function or another window-management function. An application can prevent changes to the window by setting or clearing the appropriate bits in the **flags** member of the **WINDOWPOS** structure. For a window with the **WS_OVERLAPPED** or **WS_THICKFRAME** style, the default implementation sends a **WM_GETMINMAXINFO** message to the window. This is done to validate the new size and position of the window and to enforce the **CS_BYTEALIGNCLIENT** and **CS_BYTEALIGN** client styles. An application can override this functionality by not calling its base class.

WINDOWPOS Structure
Windows 3.1 Only

A **WINDOWPOS** data structure has this form:

```
typedef struct tagWINDOWPOS { /* wp */
    HWND    hwnd;
    HWND    hwndInsertAfter;
    int     x;
    int     y;
    int     cx;
    int     cy;
    UINT    flags;
} WINDOWPOS;
```

The **WINDOWPOS** structure contains information about the size and position of a window. ♦

Members

A **WINDOWPOS** structure has the following members:

hwnd Identifies the window.

hwndInsertAfter Identifies the window behind which this window is placed.

x Specifies the position of the left edge of the window.

y Specifies the position of the right edge of the window.

cx Specifies the window width.

cy Specifies the window height.

flags Specifies window-positioning options. This member can be one of the following values:

- **SWP_DRAWFRAME** Draws a frame (defined in the class description for the window) around the window. The window receives a **WM_NCCALCSIZE** message.
- **SWP_HIDEWINDOW** Hides the window.
- **SWP_NOACTIVATE** Does not activate the window.
- **SWP_NOMOVE** Retains current position (ignores the **x** and **y** members).
- **SWP_NOOWNERZORDER** Does not change the owner window's position in the Z-order.
- **SWP_NOSIZE** Retains current size (ignores the **cx** and **cy** members).
- **SWP_NOREDRAW** Does not redraw changes.
- **SWP_NOREPOSITION** Same as **SWP_NOOWNERZORDER**.
- **SWP_NOZORDER** Retains current ordering (ignores the **hwndInsertAfter** member).
- **SWP_SHOWWINDOW** Displays the window.

See Also

CWnd::OnWindowPosChanged, WM_WINDOWPOSCHANGING

CWnd::OnWinIniChange

Protected

afx_msg void OnWinIniChange(LPCSTR *lpszSection* **);** ♦

lpszSection Points to a string that specifies the name of the section that has changed. (The string does not include the square brackets that enclose the section name.)

Remarks

Called after a change has been made to the Windows initialization file, WIN.INI. The **SystemParametersInfo** Windows function calls **OnWinIniChange** after an application uses the function to change a setting in the WIN.INI file. To send the **WM_WININICHANGE** message to all top-level windows, an application can use the **SendMessage** Windows function with its *hwnd* parameter set to **HWND_BROADCAST**.

If an application changes many different sections in WIN.INI at the same time, the application should send one **WM_WININICHANGE** message with *lpszSection* set to **NULL**. Otherwise, an application should send **WM_WININICHANGE** each time it makes a change to WIN.INI.

If an application receives an **OnWinIniChange** call with *lpszSection* set to **NULL**, the application should check all sections in WIN.INI that affect the application.

See Also

::SendMessage, **::SystemParametersInfo**, **WM_WININICHANGE**

CWnd::OpenClipboard

BOOL OpenClipboard( );

Remarks

Opens the Clipboard. Other applications will not be able to modify the Clipboard until the **CloseClipboard** Windows function is called. The current **CWnd** object will not become the owner of the Clipboard until the **EmptyClipboard** Windows function is called.

Return Value

Nonzero if the Clipboard is opened via **CWnd**, or 0 if another application or window has the Clipboard open.

See Also

::CloseClipboard, **::EmptyClipboard**, **::OpenClipboard**

CWnd::PostMessage

BOOL PostMessage(UINT *message*, **WPARAM** *wParam* **= 0,**
LPARAM *lParam* **= 0);**

message Specifies the message to be posted.

wParam Specifies additional message information. The content of this parameter depends on the message being posted.

lParam Specifies additional message information. The content of this parameter depends on the message being posted.

Remarks Places a message in the window's message queue and then returns without waiting for the corresponding window to process the message. Messages in a message queue are retrieved by calls to the **GetMessage** or **PeekMessage** Windows function. The Windows **PostMessage** function can be used to access another application.

Return Value Nonzero if the message is posted; otherwise 0.

See Also **::GetMessage, ::PeekMessage, ::PostMessage, ::PostAppMessage, CWnd::SendMessage**

CWnd::PostNcDestroy

Protected **virtual void PostNcDestroy( );** ♦

Remarks Called by the default **OnNcDestroy** member function after the window has been destroyed. Derived classes can use this function for custom cleanup such as the deletion of the **this** pointer.

See Also **CWnd::OnNcDestroy**

CWnd::PreCreateWindow

virtual BOOL PreCreateWindow(CREATESTRUCT& *cs* **);**

cs A **CREATESTRUCT** structure.

Remarks Called by the framework before the creation of the Windows window attached to this **CWnd** object.

Never call this function directly.

The default implementation of this function checks for a NULL window class name and substitutes an appropriate default.

Override this member function to modify the **CREATESTRUCT** structure before the window is created. If you override this member function, you should examine the source code to determine whether or not you need to invoke the base class implementation.

Return Value Nonzero if the window creation should continue; 0 to indicate creation failure.

See Also **CWnd::Create**, **CREATESTRUCT**

CWnd::PreTranslateMessage

virtual BOOL PreTranslateMessage(MSG* *pMsg* **);**

pMsg Points to a **MSG** structure that contains the message to process.

Remarks Used by class **CWinApp** to translate window messages before they are dispatched to the **TranslateMessage** and **DispatchMessage** Windows functions.

Return Value Nonzero if the message was translated and should not be dispatched; 0 if the message was not translated and should be dispatched.

See Also **::TranslateMessage**, **::IsDialogMessage**, **CWinApp::PreTranslateMessage**

CWnd::RedrawWindow

Windows 3.1 Only **BOOL RedrawWindow(LPCRECT** *lpRectUpdate* **= NULL, CRgn*** *prgnUpdate* **= NULL, UINT** *flags* **= RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE);** ♦

lpRectUpdate Points to a **RECT** structure containing the coordinates of the update rectangle. This parameter is ignored if *prgnUpdate* contains a valid region handle.

prgnUpdate Identifies the update region. If both *prgnUpdate* and *lpRectUpdate* are **NULL**, the entire client area is added to the update region.

flags The following flags are used to invalidate the window:

- **RDW_ERASE** Causes the window to receive a **WM_ERASEBKGND** message when the window is repainted. The **RDW_INVALIDATE** flag must also be specified; otherwise **RDW_ERASE** has no effect.
- **RDW_FRAME** Causes any part of the nonclient area of the window that intersects the update region to receive a **WM_NCPAINT** message. The **RDW_INVALIDATE** flag must also be specified; otherwise **RDW_FRAME** has no effect.

- **RDW_INTERNALPAINT** Causes a **WM_PAINT** message to be posted to the window regardless of whether the window contains an invalid region.
- **RDW_INVALIDATE** Invalidate *lpRectUpdate* or *prgnUpdate* (only one may be not **NULL**). If both are **NULL**, the entire window is invalidated.

The following flags are used to validate the window:

- **RDW_NOERASE** Suppresses any pending **WM_ERASEBKGND** messages.
- **RDW_NOFRAME** Suppresses any pending **WM_NCPAINT** messages. This flag must be used with **RDW_VALIDATE** and is typically used with **RDW_NOCHILDREN**. This option should be used with care, as it could prevent parts of a window from painting properly.
- **RDW_NOINTERNALPAINT** Suppresses any pending internal **WM_PAINT** messages. This flag does not affect **WM_PAINT** messages resulting from invalid areas.
- **RDW_VALIDATE** Validates *lpRectUpdate* or *prgnUpdate* (only one may be not **NULL**). If both are **NULL**, the entire window is validated. This flag does not affect internal **WM_PAINT** messages.

The following flags control when repainting occurs. Painting is not performed by the **RedrawWindow** function unless one of these bits is specified.

- **RDW_ERASENOW** Causes the affected windows (as specified by the **RDW_ALLCHILDREN** and **RDW_NOCHILDREN** flags) to receive **WM_NCPAINT** and **WM_ERASEBKGND** messages, if necessary, before the function returns. **WM_PAINT** messages are deferred.
- **RDW_UPDATENOW** Causes the affected windows (as specified by the **RDW_ALLCHILDREN** and **RDW_NOCHILDREN** flags) to receive **WM_NCPAINT**, **WM_ERASEBKGND**, and **WM_PAINT** messages, if necessary, before the function returns.

By default, the windows affected by the **RedrawWindow** function depend on whether the specified window has the **WS_CLIPCHILDREN** style. The child windows of **WS_CLIPCHILDREN** windows are not affected. However, those windows that are not **WS_CLIPCHILDREN** windows are recursively validated or invalidated until a **WS_CLIPCHILDREN** window is encountered. The following flags control which windows are affected by the **RedrawWindow** function:

- **RDW_ALLCHILDREN** Includes child windows, if any, in the repainting operation.
- **RDW_NOCHILDREN** Excludes child windows, if any, from the repainting operation.

Remarks

Updates the specified rectangle or region in the given window's client area.

When the **RedrawWindow** member function is used to invalidate part of the desktop window, that window does not receive a **WM_PAINT** message. To repaint the desktop, an application should use **CWnd::ValidateRgn**, **CWnd::InvalidateRgn**, **CWnd::UpdateWindow**, or **::RedrawWindow**.

CWnd::ReleaseDC

int ReleaseDC(CDC* *pDC* **);**

pDC Identifies the device context to be released.

Remarks

Releases a device context, freeing it for use by other applications. The effect of the **ReleaseDC** member function depends on the device-context type. The application must call the **ReleaseDC** member function for each call to the **GetWindowDC** member function and for each call to the **GetDC** member function.

Return Value

Nonzero if successful; otherwise 0.

See Also

CWnd::GetDC, **CWnd::GetWindowDC**, **::ReleaseDC**

CWnd::RepositionBars

void RepositionBars(UINT *nIDFirst*, **UINT** *nIDLast*, **UINT** *nIDLeftOver* **);**

nIDFirst Specifies ID of first of a range of control bars to reposition and resize.

nIDLast Specifies ID of last of a range of control bars to reposition and resize.

nIDLeftOver Specifies ID of pane that fills the rest of the client area.

Remarks

Called to reposition and resize control bars in the client area of a window. The *nIDFirst* and *nIDLast* parameters define a range of control-bar IDs to be repositioned in the client area. *nIDLeftOver* specifies the ID of the child window (normally the view) which is repositioned and resized to fill the rest of the client area not filled by control bars.

See Also

CFrameWnd::RecalcLayout

CWnd::ScreenToClient

void ScreenToClient(LPPOINT *lpPoint* **) const;**

void ScreenToClient(LPRECT *lpRect* **) const;**

lpPoint Points to a **CPoint** object or **POINT** structure that contains the screen coordinates to be converted.

lpRect Points to a **CRect** object or **RECT** structure that contains the screen coordinates to be converted.

Remarks

Converts the screen coordinates of a given point or rectangle on the display to client coordinates. The **ScreenToClient** member function replaces the screen coordinates given in *lpPoint* or *lpRect* with client coordinates. The new coordinates are relative to the upper-left corner of the **CWnd** client area.

See Also

CWnd::ClientToScreen, ::ScreenToClient

CWnd::ScrollWindow

void ScrollWindow(int *xAmount*, **int** *yAmount*, **LPCRECT** *lpRect* **= NULL, LPCRECT** *lpClipRect* **= NULL);**

xAmount Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

yAmount Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

lpRect Points to a **CRect** object or **RECT** structure that specifies the portion of the client area to be scrolled. If *lpRect* is **NULL**, the entire client area is scrolled. The caret is repositioned if the cursor rectangle intersects the scroll rectangle.

lpClipRect Points to a **CRect** object or **RECT** structure that specifies the clipping rectangle to scroll. Only bits inside this rectangle are scrolled. Bits outside this rectangle are not affected even if they are in the *lpRect* rectangle. If *lpClipRect* is **NULL**, no clipping is performed on the scroll rectangle.

Remarks

Scrolls the contents of the client area of the current **CWnd** object. If the caret is in the **CWnd** being scrolled, **ScrollWindow** automatically hides the caret to prevent it from being erased and then restores the caret after the scroll is finished. The caret position is adjusted accordingly.

The area uncovered by the **ScrollWindow** member function is not repainted but is combined into the current **CWnd** object's update region. The application will eventually receive a **WM_PAINT** message notifying it that the region needs repainting. To repaint the uncovered area at the same time the scrolling is done, call the **UpdateWindow** member function immediately after calling **ScrollWindow**.

If *lpRect* is **NULL**, the positions of any child windows in the window are offset by the amount specified by *xAmount* and *yAmount*, and any invalid (unpainted) areas in the **CWnd** are also offset. **ScrollWindow** is faster when *lpRect* is **NULL**. If *lpRect* is not **NULL**, the positions of child windows are not changed, and invalid areas in **CWnd** are not offset. To prevent updating problems when *lpRect* is not **NULL**, call the **UpdateWindow** member function to repaint **CWnd** before calling **ScrollWindow**.

See Also **CWnd::UpdateWindow**, **::ScrollWindow**

CWnd::ScrollWindowEx

Windows 3.1 Only **int ScrollWindowEx(int** *dx*, **int** *dy*, **LPCRECT** *lpRectScroll*, **LPCRECT** *lpRectClip*, **CRgn*** *prgnUpdate*, **LPRECT** *lpRectUpdate*, **UINT** *flags* **);** ♦

dx Specifies the amount, in device units, of horizontal scrolling. This parameter must have a negative value to scroll to the left.

dy Specifies the amount, in device units, of vertical scrolling. This parameter must have a negative value to scroll up.

lpRectScroll Points to a **RECT** structure that specifies the portion of the client area to be scrolled. If this parameter is **NULL**, the entire client area is scrolled.

lpRectClip Points to a **RECT** structure that specifies the clipping rectangle to scroll. This structure takes precedence over the rectangle pointed to by *lpRectScroll*. Only bits inside this rectangle are scrolled. Bits outside this rectangle are not affected even if they are in the *lpRectScroll* rectangle. If this parameter is **NULL**, no clipping is performed on the scroll rectangle.

prgnUpdate Identifies the region that is modified to hold the region invalidated by scrolling. This parameter may be **NULL**.

lpRectUpdate Points to a **RECT** structure that will receive the boundaries of the rectangle invalidated by scrolling. This parameter may be **NULL**.

flags Can have one of the following values:

- **SW_ERASE** When specified with **SW_INVALIDATE**, erases the newly invalidated region by sending a **WM_ERASEBKGND** message to the window.
- **SW_INVALIDATE** Invalidates the region identified by *prgnUpdate* after scrolling.
- **SW_SCROLLCHILDREN** Scrolls all child windows that intersect the rectangle pointed to by *lpRectScroll* by the number of pixels specified in *dx* and *dy*. Windows sends a **WM_MOVE** message to all child windows that intersect *lpRectScroll*, even if they do not move. The caret is repositioned when a child window is scrolled and the cursor rectangle intersects the scroll rectangle.

Remarks

Scrolls the contents of a window's client area. This function is similar to the **ScrollWindow** function, with some additional features. If **SW_INVALIDATE** and **SW_ERASE** are not specified, the **ScrollWindowEx** member function does not invalidate the area that is scrolled away from. If either of these flags is set, **ScrollWindowEx** invalidates this area. The area is not updated until the application calls the **UpdateWindow** member function, calls the **RedrawWindow** member function (specifying **RDW_UPDATENOW** or **RDW_ERASENOW**), or retrieves the **WM_PAINT** message from the application queue.

If the window has the **WS_CLIPCHILDREN** style, the returned areas specified by *prgnUpdate* and *lpRectUpdate* represent the total area of the scrolled window that must be updated, including any areas in child windows that need updating. If the **SW_SCROLLCHILDREN** flag is specified, Windows will not properly update the screen if part of a child window is scrolled. The part of the scrolled child window that lies outside the source rectangle will not be erased and will not be redrawn properly in its new destination. Use the **DeferWindowPos** Windows function to move child windows that do not lie completely within the *lpRectScroll* rectangle. The cursor is repositioned if the **SW_SCROLLCHILDREN** flag is set and the caret rectangle intersects the scroll rectangle.

All input and output coordinates (for *lpRectScroll*, *lpRectClip*, *lpRectUpdate*, and *prgnUpdate*) are assumed to be in client coordinates, regardless of whether the window has the **CS_OWNDC** or **CS_CLASSDC** class style. Use the **LPtoDP** and **DPtoLP** Windows functions to convert to and from logical coordinates, if needed.

Return Value

The return value is **SIMPLEREGION** (rectangular invalidated region), **COMPLEXREGION** (nonrectangular invalidated region; overlapping rectangles), or **NULLREGION** (no invalidated region), if the function is successful; otherwise the return value is **ERROR**.

See Also

CWnd::RedrawWindow, **CDC::ScrollDC**, **CWnd::ScrollWindow**, **CWnd::UpdateWindow**, **::DeferWindowPos**, **::ScrollWindowEx**

CWnd::SendDlgItemMessage

LRESULT SendDlgItemMessage(int *nID*, **UINT** *message*,
WPARAM *wParam* = **0, LPARAM** *lParam* = **0);**

nID Specifies the identifier of the dialog control that will receive the message.

message Specifies the message to be sent.

wParam Specifies additional message-dependent information.

lParam Specifies additional message-dependent information.

Remarks

Sends a message to a control. The **SendDlgItemMessage** member function does not return until the message has been processed. Using **SendDlgItemMessage** is identical to obtaining a **CWnd*** to the given control and calling the **SendMessage** member function.

Return Value

Specifies the value returned by the control's window procedure, or 0 if the control was not found.

See Also

CWnd::SendMessage, **::SendDlgItemMessage**

CWnd::SendMessage

LRESULT SendMessage(UINT *message*, **WPARAM** *wParam* = **0,**
LPARAM *lParam* = **0);**

message Specifies the message to be sent.

wParam Specifies additional message-dependent information.

lParam Specifies additional message-dependent information.

Remarks

Sends the specified message to this window. The **SendMessage** member function calls the window procedure directly and does not return until that window procedure has processed the message. This is in contrast to the **PostMessage** member function, which places the message into the window's message queue and returns immediately.

Return Value

The result of the message processing; its value depends on the message sent.

See Also

::InSendMessage, **CWnd::PostMessage**, **CWnd::SendDlgItemMessage**, **::SendMessage**

CWnd::SendMessageToDescendants

void SendMessageToDescendants(UINT *message*, **WPARAM** *wParam* = **0, LPARAM** *lParam* = **0, BOOL** *bDeep* = **TRUE);**

message Specifies the message to be sent.

wParam Specifies additional message-dependent information.

lParam Specifies additional message-dependent information.

bDeep Specifies the level to which to search. If **TRUE**, search all children; if **FALSE**, search only immediate children.

Remarks Call this member function to send the specified Windows message to all descendant windows. If the *bDeep* parameter is **FALSE**, the message is sent just to the immediate children of the window; otherwise the message is sent to all descendant windows.

CWnd::SetActiveWindow

CWnd* SetActiveWindow( );

Remarks Makes **CWnd** the active window. The **SetActiveWindow** member function should be used with care since it allows an application to arbitrarily take over the active window and input focus. Normally, Windows takes care of all activation.

Return Value The window that was previously active. The returned pointer may be temporary and should not be stored for later use.

See Also **::SetActiveWindow, CWnd::GetActiveWindow**

CWnd::SetCapture

CWnd* SetCapture( );

Remarks Causes all subsequent mouse input to be sent to the current **CWnd** object regardless of the position of the cursor. When **CWnd** no longer requires all mouse input, the application should call the **ReleaseCapture** function so that other windows can receive mouse input.

Return Value A pointer to the window object that previously received all mouse input. It is **NULL** if there is no such window. The returned pointer may be temporary and should not be stored for later use.

See Also **::ReleaseCapture**, **::SetCapture**, **CWnd::GetCapture**

CWnd::SetCaretPos

static void PASCAL SetCaretPos(POINT *point* **);**

point Specifies the new x- and y-coordinates (in client coordinates) of the caret.

Remarks Sets the position of the caret. The **SetCaretPos** member function moves the caret only if it is owned by a window in the current task. **SetCaretPos** moves the caret whether or not the caret is hidden. The caret is a shared resource. A window should not move the caret if it does not own the caret.

See Also **CWnd::GetCaretPos**, **::SetCaretPos**

CWnd::SetClipboardViewer

HWND SetClipboardViewer( );

Remarks Adds this window to the chain of windows that are notified (by means of the **WM_DRAWCLIPBOARD** message) whenever the content of the Clipboard is changed. A window that is part of the Clipboard-viewer chain must respond to **WM_DRAWCLIPBOARD**, **WM_CHANGECBCHAIN**, and **WM_DESTROY** messages and pass the message to the next window in the chain. This member function sends a **WM_DRAWCLIPBOARD** message to the window. Since the handle to the next window in the Clipboard-viewer chain has not yet been returned, the application should not pass on the **WM_DRAWCLIPBOARD** message that it receives during the call to **SetClipboardViewer**. To remove itself from the Clipboard-viewer chain, an application must call the **ChangeClipboardChain** member function.

Return Value A handle to the next window in the Clipboard-viewer chain if successful. Applications should save this handle (it can be stored as a member variable) and use it when responding to Clipboard-viewer chain messages.

See Also **CWnd::ChangeClipboardChain**, **::SetClipboardViewer**

CWnd::SetDlgItemInt

void SetDlgItemInt(int *nID*, **UINT** *nValue*, **BOOL** *bSigned* = **TRUE);**

nID Specifies the integer ID of the control to be changed.

nValue Specifies the integer value used to generate the item text.

bSigned Specifies whether the integer value is signed or unsigned. If this parameter is **TRUE**, *nValue* is signed. If this parameter is **TRUE** and *nValue* is less than 0, a minus sign is placed before the first digit in the string. If this parameter is **FALSE**, *nValue* is unsigned.

Remarks

Sets the text of a given control in a dialog box to the string representation of a specified integer value. **SetDlgItemInt** sends a **WM_SETTEXT** message to the given control.

See Also

CWnd::GetDlgItemInt, **::SetDlgItemInt**, **WM_SETTEXT**

CWnd::SetDlgItemText

void SetDlgItemText(int *nID*, **LPCSTR** *lpszString* **);**

nID Identifies the control whose text is to be set.

lpszString Points to a **CString** object or null-terminated string that contains the text to be copied to the control.

Remarks

Sets the caption or text of a control owned by a window or dialog box. **SetDlgItemText** sends a **WM_SETTEXT** message to the given control.

See Also

::SetDlgItemText, **WM_SETTEXT**, **CWnd::GetDlgItemText**

CWnd::SetFocus

CWnd* SetFocus( );

Remarks

Claims the input focus. The input focus directs all subsequent keyboard input to this window. The window, if any, that previously had the input focus loses it. The **SetFocus** member function sends a **WM_KILLFOCUS** message to the window that loses the input focus and a **WM_SETFOCUS** message to the window that receives the input focus. It also activates either the window or its parent. If the current window is active but doesn't have the focus (that is, no window has the focus), any key pressed will produce the messages **WM_SYSCHAR**, **WM_SYSKEYDOWN**, or **WM_SYSKEYUP**.

Return Value A pointer to the window object that previously had the input focus. It is **NULL** if there is no such window. The returned pointer may be temporary and should not be stored.

See Also **::SetFocus**, **CWnd::GetFocus**

CWnd::SetFont

void SetFont(CFont* *pFont*, **BOOL** *bRedraw* = **TRUE);**

pFont Specifies the new font.

bRedraw If **TRUE**, redraw the **CWnd** object.

Remarks Sets the window's current font to the specified font. If *bRedraw* is **TRUE**, the window will also be redrawn.

See Also **CWnd::GetFont**, **WM_SETFONT**

CWnd::SetMenu

BOOL SetMenu(CMenu* *pMenu* **);**

pMenu Identifies the new menu. If this parameter is **NULL**, the current menu is removed.

Remarks Sets the current menu to the specified menu. Causes the window to be redrawn to reflect the menu change. **SetMenu** will not destroy a previous menu. An application should call the **CMenu::DestroyMenu** member function to accomplish this task.

Return Value Nonzero if the menu is changed; otherwise 0.

See Also **CMenu::DestroyMenu**, **CMenu::LoadMenu**, **::SetMenu**, **CWnd::GetMenu**

CWnd::SetParent

CWnd* SetParent(CWnd* *pWndNewParent* **);**

pWndNewParent Identifies the new parent window.

Remarks Changes the parent window of a child window. If the child window is visible, Windows performs the appropriate redrawing and repainting.

Return Value A pointer to the previous parent window object if successful. The returned pointer may be temporary and should not be stored for later use.

See Also **::SetParent**, **CWnd::GetParent**

CWnd::SetRedraw

void SetRedraw(BOOL *bRedraw* **= TRUE);**

bRedraw Specifies the state of the redraw flag. If this parameter is **TRUE**, the redraw flag is set; if **FALSE**, the flag is cleared.

Remarks An application calls **SetRedraw** to allow changes to be redrawn or to prevent changes from being redrawn. This member function sets or clears the redraw flag. While the redraw flag is cleared, the contents will not be updated after each change and will not be repainted until the redraw flag is set. For example, an application that needs to add several items to a list box can clear the redraw flag, add the items, and then set the redraw flag. Finally, the application can call the **Invalidate** or **InvalidateRect** member function to cause the list box to be repainted.

See Also **WM_SETREDRAW**

CWnd::SetScrollPos

int SetScrollPos(int *nBar*, **int** *nPos*, **BOOL** *bRedraw* **= TRUE);**

nBar Specifies the scroll bar to be set, using one of the following values:

- **SB_HORZ** Sets the position of the scroll box in the horizontal scroll bar of the window.
- **SB_VERT** Sets the position of the scroll box in the vertical scroll bar of the window.

nPos Specifies the new position of the scroll box. It must be within the scrolling range.

bRedraw Specifies whether the scroll bar should be repainted to reflect the new scroll-box position. If this parameter is **TRUE**, the scroll bar is repainted; if **FALSE**, the scroll bar is not repainted.

Remarks Sets the current position of a scroll box and, if requested, redraws the scroll bar to reflect the new position of the scroll box. Setting *bRedraw* to **FALSE** is useful whenever the scroll bar will be redrawn by a subsequent call to another function.

Return Value The previous position of the scroll box.

See Also **::SetScrollPos**, **CWnd::GetScrollPos**, **CScrollBar::SetScrollPos**

CWnd::SetScrollRange

void SetScrollRange(int *nBar*, **int** *nMinPos*, **int** *nMaxPos*,
BOOL *bRedraw* = **TRUE);**

nBar Specifies the scroll bar to be set. This parameter can be one of the following values:

- **SB_HORZ** Sets the range of the horizontal scroll bar of the window.
- **SB_VERT** Sets the range of the vertical scroll bar of the window.

nMinPos Specifies the minimum scrolling position.

nMaxPos Specifies the maximum scrolling position.

bRedraw Specifies whether the scroll bar should be redrawn to reflect the change. If *bRedraw* is **TRUE**, the scroll bar is redrawn; if **FALSE**, the scroll bar is not redrawn.

Remarks Sets minimum and maximum position values for the given scroll bar. It can also be used to hide or show standard scroll bars. An application should not call this function to hide a scroll bar while processing a scroll-bar notification message. If the call to **SetScrollRange** immediately follows a call to the **SetScrollPos** member function, the *bRedraw* parameter in the **SetScrollPos** member function should be 0 to prevent the scroll bar from being drawn twice. The default range for a standard scroll bar is 0 through 100. The default range for a scroll bar control is empty (both the *nMinPos* and *nMaxPos* values are 0). The difference between the values specified by *nMinPos* and *nMaxPos* must not be greater than **INT_MAX**.

See Also **CWnd::SetScrollPos**, **::SetScrollRange**, **CWnd::GetScrollRange**

CWnd::SetTimer

UINT SetTimer(UINT *nIDEvent*, **UINT** *nElapse*, **void (CALLBACK EXPORT*** *lpfnTimer*)**(HWND, UINT, UINT, DWORD));**

nIDEvent Specifies a nonzero timer identifier.

nElapse Specifies the time-out value, in milliseconds.

lpfnTimer Specifies the address of the application-supplied `TimerProc` callback function that processes the **WM_TIMER** messages. If this parameter is **NULL**, the **WM_TIMER** messages are placed in the application's message queue and handled by the **CWnd** object.

Remarks

Installs a system timer. A time-out value is specified, and every time a time-out occurs, the system posts a **WM_TIMER** message to the installing application's message queue or passes the message to an application-defined **TimerProc** callback function. The *lpfnTimer* callback function need not be named `TimerProc`, but it must be defined as follows and return 0.

```
void CALLBACK EXPORT TimerProc(
    HWND hWnd,      //handle of CWnd that called SetTimer
    UINT nMsg,      //WM_TIMER
    UINT nIDEvent   //timer identification
    DWORD dwTime    //system time
);
```

Timers are a limited global resource; therefore it is important that an application check the value returned by the **SetTimer** member function to verify that a timer is actually available.

Return Value

The timer identifier of the new timer if the function is successful. An application passes this value to the **KillTimer** member function to kill the timer. Nonzero if successful; otherwise 0.

See Also

WM_TIMER, **CWnd::KillTimer**, **::SetTimer**, **CWnd::FromHandle**

CWnd::SetWindowPlacement

Windows 3.1 Only

BOOL SetWindowPlacement(const WINDOWPLACEMENT FAR* *lpwndpl* **);** ♦

lpwndpl Points to a **WINDOWPLACEMENT** structure that specifies the new show state and positions.

Remarks

Sets the show state and the normal (restored), minimized, and maximized positions for a window.

Return Value

Nonzero if the function is successful; otherwise 0.

WINDOWPLACEMENT Structure Windows 3.1 Only

A **WINDOWPLACEMENT** data structure has this form:

```
typedef struct tagWINDOWPLACEMENT {     /* wndpl */
    UINT  length;
    UINT  flags;
    UINT  showCmd;
    POINT ptMinPosition;
    POINT ptMaxPosition;
    RECT  rcNormalPosition;
} WINDOWPLACEMENT;
```

The **WINDOWPLACEMENT** structure contains information about the placement of a window on the screen. ♦

Members

The **WINDOWPLACEMENT** structure has the following members:

length Specifies the length, in bytes, of the structure.

flags Specifies flags that control the position of the minimized window and the method by which the window is restored. This member can be one or both of the following flags:

- **WPF_SETMINPOSITION** Specifies that the x- and y-positions of the minimized window may be specified. This flag must be specified if the coordinates are set in the **ptMinPosition** member.
- **WPF_RESTORETOMAXIMIZED** Specifies that the restored window will be maximized, regardless of whether it was maximized before it was minimized. This setting is valid only the next time the window is restored. It does not change the default restoration behavior. This flag is valid only when the **SW_SHOWMINIMIZED** value is specified for the **showCmd** member.

showCmd Specifies the current show state of the window. This member may be one of the following values:

- **SW_HIDE** Hides the window and passes activation to another window.
- **SW_MINIMIZE** Minimizes the specified window and activates the top-level window in the system's list.
- **SW_RESTORE** Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as **SW_SHOWNORMAL**).

- **SW_SHOW** Activates a window and displays it in its current size and position.
- **SW_SHOWMAXIMIZED** Activates a window and displays it as a maximized window.
- **SW_SHOWMINIMIZED** Activates a window and displays it as an icon.
- **SW_SHOWMINNOACTIVE** Displays a window as an icon. The window that is currently active remains active.
- **SW_SHOWNA** Displays a window in its current state. The window that is currently active remains active.
- **SW_SHOWNOACTIVATE** Displays a window in its most recent size and position. The window that is currently active remains active.
- **SW_SHOWNORMAL** Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as **SW_RESTORE**).

ptMinPosition Specifies the position of the window's top-left corner when the window is minimized.

ptMaxPosition Specifies the position of the window's top-left corner when the window is maximized.

rcNormalPosition Specifies the window's coordinates when the window is in the normal (restored) position.

See Also **CWnd::GetWindowPlacement, ::SetWindowPlacement**

CWnd::SetWindowPos

BOOL SetWindowPos(const CWnd* *pWndInsertAfter*, **int** *x*, **int** *y*, **int** *cx*, **int** *cy*, **UINT** *nFlags* **);**

pWndInsertAfter Identifies the **CWnd** object that will precede this **CWnd** object in the Z-order. This parameter can be a pointer to a **CWnd** or one of the following values:

- **wndBottom** Places the window at the bottom of the Z-order. If this **CWnd** is a topmost window, the window loses its topmost status; the system places the window at the bottom of all other windows.
- **wndTop** Places the window at the top of the Z-order.

Windows 3.1 Only

- **wndTopMost** Places the window above all nontopmost windows. The window maintains its topmost position even when it is deactivated.
- **wndNoTopMost** Repositions the window to the top of all nontopmost windows (that is, behind all topmost windows). This flag has no effect if the window is already a nontopmost window. ♦

See the following "Remarks" section for rules about how this parameter is used.

x Specifies the new position of the left side of the window.

y Specifies the new position of the top of the window.

cx Specifies the new width of the window.

cy Specifies the new height of the window.

nFlags Specifies sizing and positioning options. This parameter can be a combination of the following:

- **SWP_DRAWFRAME** Draws a frame (defined when the window was created) around the window.
- **SWP_HIDEWINDOW** Hides the window.
- **SWP_NOACTIVATE** Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the topmost or the nontopmost group (depending on the setting of the *pWndInsertAfter* parameter).
- **SWP_NOMOVE** Retains current position (ignores the *x* and *y* parameters).
- **SWP_NOREDRAW** Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title and scroll bars), and any part of the parent window uncovered as a result of the moved window. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that must be redrawn.
- **SWP_NOSIZE** Retains current size (ignores the *cx* and *cy* parameters).
- **SWP_NOZORDER** Retains current ordering (ignores *pWndInsertAfter*).
- **SWP_SHOWWINDOW** Displays the window.

Remarks

Call this member function to change the size, position, and Z-order of child, pop-up, and top-level windows.

Windows are ordered on the screen according to their Z-order; the window at the top of the Z-order appears on top of all other windows in the order.

All coordinates for child windows are client coordinates (relative to the upper-left corner of the parent window's client area).

A window can be moved to the top of the Z-order either by setting the *pWndInsertAfter* parameter to **&wndTopMost** and ensuring that the **SWP_NOZORDER** flag is not set or by setting a window's Z-order so that it is above any existing topmost windows. When a nontopmost window is made topmost, its owned windows are also made topmost. Its owners are not changed. A topmost window is no longer topmost if it is repositioned to the bottom (**&wndBottom**) of the Z-order or after any nontopmost window. When a topmost window is made nontopmost, all of its owners and its owned windows are also made nontopmost windows.

If neither **SWP_NOACTIVATE** nor **SWP_NOZORDER** is specified (that is, when the application requests that a window be simultaneously activated and placed in the specified Z-order), the value specified in *pWndInsertAfter* is used only in the following circumstances:

- Neither **&wndTopMost** nor **&wndNoTopMost** is specified in the *pWndInsertAfter* parameter.
- This window is not the active window.

An application cannot activate an inactive window without also bringing it to the top of the Z-order. Applications can change the Z-order of an activated window without restrictions.

A nontopmost window may own a topmost window, but not vice versa. Any window (for example, a dialog box) owned by a topmost window is itself made a topmost window to ensure that all owned windows stay above their owner.

Windows 3.1 Only

With Windows version 3.1, windows can be moved to the top of the Z-order and locked there by setting their **WS_EX_TOPMOST** styles. Such a topmost window maintains its topmost position even when deactivated. For example, selecting the WinHelp Always On Top command makes the Help window topmost, and it then remains visible when you return to your application.

To create a topmost window, call **SetWindowPos** with the *pWndInsertAfter* parameter equal to **&wndTopMost**, or set the **WS_EX_TOPMOST** style when you create the window.

If the Z-order contains any windows with the **WS_EX_TOPMOST** style, a window moved with the **&wndTopMost** value is placed at the top of all nontopmost windows, but below any topmost windows. When an application activates an inactive window without the **WS_EX_TOPMOST** bit, the window is moved above all nontopmost windows but below any topmost windows.

If **SetWindowPos** is called when the *pWndInsertAfter* parameter is **&wndBottom** and **CWnd** is a topmost window, the window loses its topmost status (**WS_EX_TOPMOST** is cleared), and the system places the window at the bottom of the Z-order. ♦

Return Value Nonzero if the function is successful; otherwise 0.

See Also **::DeferWindowPos**, **::SetWindowPos**

CWnd::SetWindowText

void SetWindowText(LPCSTR *lpszString* **);**

lpszString Points to a **CString** object or null-terminated string to be used as the new title or control text.

Remarks Sets the window's title to the specified text. If the window is a control, the text within the control is set. This function causes a **WM_SETTEXT** message to be sent to this window.

See Also **CWnd::GetWindowText**, **::SetWindowText**

CWnd::ShowCaret

void ShowCaret( );

Remarks Shows the caret on the screen at the caret's current position. Once shown, the caret begins flashing automatically. The **ShowCaret** member function shows the caret only if it has a current shape and has not been hidden two or more times consecutively. If the caret is not owned by this window, the caret is not shown.

Hiding the caret is cumulative. If the **HideCaret** member function has been called five times consecutively, **ShowCaret** must be called five times to show the caret. The caret is a shared resource. The window should show the caret only when it has the input focus or is active.

See Also **CWnd::HideCaret**, **::ShowCaret**

CWnd::ShowOwnedPopups

void ShowOwnedPopups(BOOL *bShow* **= TRUE);**

bShow Specifies whether pop-up windows are to be shown or hidden. If this parameter is **TRUE**, all hidden pop-up windows are shown. If this parameter is **FALSE**, all visible pop-up windows are hidden.

Remarks Shows or hides all pop-up windows owned by this window.

See Also **::ShowOwnedPopups**

CWnd::ShowScrollBar

void ShowScrollBar(UINT *nBar*, **BOOL** *bShow* **= TRUE);**

nBar Specifies whether the scroll bar is a control or part of a window's nonclient area. If it is part of the nonclient area, *nBar* also indicates whether the scroll bar is positioned horizontally, vertically, or both. It must be one of the following:

- **SB_BOTH** Specifies the horizontal and vertical scroll bars of the window.
- **SB_HORZ** Specifies that the window is a horizontal scroll bar.
- **SB_VERT** Specifies that the window is a vertical scroll bar.

bShow Specifies whether Windows shows or hides the scroll bar. If this parameter is **TRUE**, the scroll bar is shown; otherwise the scroll bar is hidden.

Remarks Shows or hides a scroll bar. An application should not call **ShowScrollBar** to hide a scroll bar while processing a scroll-bar notification message.

See Also **::ShowScrollBar**, **CScrollBar::ShowScrollBar**

CWnd::ShowWindow

BOOL ShowWindow(int *nCmdShow* **);**

nCmdShow Specifies how the **CWnd** is to be shown. It must be one of the following values:

- **SW_HIDE** Hides this window and passes activation to another window.
- **SW_MINIMIZE** Minimizes the window and activates the top-level window in the system's list.
- **SW_RESTORE** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- **SW_SHOW** Activates the window and displays it in its current size and position.
- **SW_SHOWMAXIMIZED** Activates the window and displays it as a maximized window.
- **SW_SHOWMINIMIZED** Activates the window and displays it as an icon.
- **SW_SHOWMINNOACTIVE** Displays the window as an icon. The window that is currently active remains active.
- **SW_SHOWNA** Displays the window in its current state. The window that is currently active remains active.
- **SW_SHOWNOACTIVATE** Displays the window in its most recent size and position. The window that is currently active remains active.
- **SW_SHOWNORMAL** Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.

Remarks

Sets the visibility state of the window. **ShowWindow** must be called only once per application for the main window with **CWinApp::m_nCmdShow**. Subsequent calls to **ShowWindow** must use one of the values listed above instead of the one specified by **m_nCmdShow**.

Return Value

Nonzero if the window was previously visible; 0 if the **CWnd** was previously hidden.

See Also

::ShowWindow, CWnd::OnShowWindow, CWnd::ShowOwnedPopups, WM_SHOWWINDOW

CWnd::SubclassDlgItem

BOOL SubclassDlgItem(UINT *nID*, **CWnd*** *pParent* **);**

nID The control's ID.

pParent The control's parent (usually a dialog box).

Remarks

Call this member function to "dynamically subclass" a control created from a dialog template and attach it to this **CWnd** object. When a control is dynamically subclassed, windows messages will route through the **CWnd**'s message map and call message handlers in the **CWnd**'s class first. Messages that are passed to the base class will be passed to the default message handler in the control.

This member function attaches the Windows control to a **CWnd** object and replaces the control's **WndProc** and **AfxWndProc** functions. The function stores the old **WndProc** in the location returned by the **GetSuperWndProcAddr** member function. You must override the **GetSuperWndProcAddr** member function for every unique window class to provide a place to store the old **WndProc**.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CWnd::GetSuperWndProcAddr, **CWnd::DefWindowProc**, **::WndProc**, **CWnd::SubclassWindow**, **CWnd::Attach**

CWnd::SubclassWindow

BOOL SubclassWindow(HWND *hWnd* **);**

hWnd A handle to the window.

Remarks

Call this member function to "dynamically subclass" a window and attach it to this **CWnd** object. When a window is dynamically subclassed, windows messages will route through the **CWnd**'s message map and call message handlers in the **CWnd**'s class first. Messages that are passed to the base class will be passed to the default message handler in the window.

This member function attaches the Windows control to a **CWnd** object and replaces the window's **WndProc** and **AfxWndProc** functions. The function stores the old **WndProc** in the location returned by the **GetSuperWndProcAddr** member function. You must override the **GetSuperWndProcAddr** member function for every unique window class to provide a place to store the old **WndProc**.

Return Value

Nonzero if the function is successful; otherwise 0.

See Also

CWnd::GetSuperWndProcAddr, **CWnd::DefWindowProc**, **::WndProc**, **CWnd::SubclassDlgItem**, **CWnd::Attach**

CWnd::UpdateData

BOOL UpdateData(BOOL *bSaveAndValidate* **= TRUE);**

bSaveAndValidate Flag that indicates whether dialog box is being initialized (**FALSE**) or data is being retrieved (**TRUE**).

Remarks

Call this member function to initialize data in a dialog box, or to retrieve and validate dialog data.

The framework automatically calls **UpdateData** with *bSaveAndValidate* set to **FALSE** when a modal dialog box is created in the default implementation of **CDialog::OnInitDialog**. The call occurs before the dialog box is visible. The default implementation of **CDialog::OnOK** calls this member function with *bSaveAndValidate* set to **TRUE** to retrieve the data, and if successful, will close the dialog box. (If the Cancel button is clicked in the dialog box, the dialog box is closed without the data being retrieved.)

Return Value

Nonzero if the operation is successful; otherwise 0. If *bSaveAndValidat*e is **TRUE**, then a return value of nonzero means that the data is successfully validated.

See Also

CWnd::DoDataExchange

CWnd::UpdateDialogControls

void UpdateDialogControls(CCmdTarget* *pTarget***,**
  BOOL *bDisableIfNoHndler* **);**

pTarget Points to the main frame window of the application, and used for routing update messages.

bDisableIfNoHndler Flag that indicates whether a control that has no update handler should be automatically displayed as disabled.

Remarks

Call this member function to update the state of dialog buttons and other controls in a dialog box or window that uses the **ON_UPDATE_COMMAND_UI** callback mechanism.

If a child control does not have a handler and *bDisableIfNoHndler* is **TRUE**, then the child control will be disabled.

The framework calls this member function for controls in dialog bars or toolbars as part of the application's idle processing.

See Also **CFrameWnd::m_bAutoMenuEnable**

CWnd::UpdateWindow

void UpdateWindow( );

Remarks Updates the client area by sending a **WM_PAINT** message if the update region is not empty. The **UpdateWindow** member function sends a **WM_PAINT** message directly, bypassing the application queue. If the update region is empty, **WM_PAINT** is not sent.

See Also **::UpdateWindow**, **CWnd::RedrawWindow**

CWnd::ValidateRect

void ValidateRect(LPCRECT *lpRect* **);**

lpRect Points to a **CRect** object or **RECT** structure that contains client coordinates of the rectangle to be removed from the update region. If *lpRect* is **NULL**, the entire window is validated.

Remarks Validates the client area within the given rectangle by removing the rectangle from the update region of the window. The **BeginPaint** member function automatically validates the entire client area. Neither the **ValidateRect** nor **ValidateRgn** member function should be called if a portion of the update region needs to be validated before **WM_PAINT** is next generated. Windows continues to generate **WM_PAINT** messages until the current update region is validated.

See Also **CWnd::BeginPaint**, **::ValidateRect**, **CWnd::ValidateRgn**

CWnd::ValidateRgn

void ValidateRgn(CRgn* *pRgn* **);**

pRgn Identifies a region that defines the area to be removed from the update region. If this parameter is **NULL**, the entire client area is removed.

Remarks

Validates the client area within the given region by removing the region from the current update region of the window. The given region must have been created previously by a region function. The region coordinates are assumed to be client coordinates. The **BeginPaint** member function automatically validates the entire client area. Neither the **ValidateRect** nor the **ValidateRgn** member function should be called if a portion of the update region must be validated before the next **WM_PAINT** message is generated.

See Also

::ValidateRgn, **CWnd::ValidateRect**

CWnd::WindowFromPoint

static CWnd* PASCAL WindowFromPoint(POINT *point* **);**

point Specifies a **CPoint** object or **POINT** data structure that defines the point to be checked.

Remarks

Retrieves the window that contains the specified point; *point* must specify the screen coordinates of a point on the screen. **WindowFromPoint** does not retrieve a hidden, disabled, or transparent window, even if the point is within the window. An application should use the **ChildWindowFromPoint** member function for a nonrestrictive search.

Return Value

A pointer to the window object in which the point lies. It is **NULL** if no window exists at the given point. The returned pointer may be temporary and should not be stored for later use.

See Also

::WindowFromPoint, **CWnd::ChildWindowFromPoint**

CWnd::WindowProc

Protected **virtual LRESULT WindowProc(UINT** *message*, **WPARAM** *wParam*, **LPARAM** *lParam* **);**

message Specifies the Windows message to be processed.

wParam Provides additional information used in processing the message. The parameter value depends on the message.

lParam Provides additional information used in processing the message. The parameter value depends on the message.

Remarks Provides a Windows procedure (**WindowProc**) for a **CWnd** object. It dispatches messages through the window's message map.

Return Value The return value depends on the message.

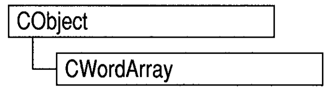
Data Members

CWnd::m_hWnd

Remarks The handle of the Windows window attached to this **CWnd**. The **m_hWnd** data member is a public variable of type **HWND**.

See Also **CWnd::Attach**, **CWnd::Detach**, **CWnd::FromHandle**

class CWordArray : public CObject



The **CWordArray** class supports arrays of 16-bit words. The member functions of **CWordArray** are similar to the member functions of class **CObArray**. Because of this similarity, you can use the **CObArray** reference documentation for member function specifics. Wherever you see a **CObject** pointer as a function parameter or return value, substitute a **WORD**.

```
CObject* CObArray::GetAt( int <nIndex> ) const;
```

for example, translates to

```
WORD CWordArray::GetAt( int <nIndex> ) const;
```

CWordArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of words is stored to an archive, either with an overloaded insertion operator or with the **Serialize** member function, each element is, in turn, serialized. If you need a dump of individual elements in the array, you must set the depth of the dump context to 1 or greater.

#include <afxcoll.h>

Construction/Destruction—Public Members

CWordArray	Constructs an empty array for words.
~CWordArray	Destroys a **CWordArray** object.

Bounds—Public Members

GetSize	Gets number of elements in this array.
GetUpperBound	Returns the largest valid index.
SetSize	Sets the number of elements to be contained in this array.

Operations—Public Members

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access—Public Members

GetAt	Returns the value at a given index.
SetAt	Sets the value for a given index; array is not allowed to grow.
ElementAt	Returns a temporary reference to the element pointer within the array.

Growing the Array—Public Members

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

Insertion/Removal—Public Members

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators—Public Members

operator [ ]	Sets or gets the element at the specified index.

Macros and Globals

The Microsoft Foundation Class Library can be divided into two major sections: 1) the Foundation classes and 2) macros and globals. If a function or variable is not a member of a class, it is a global function or variable.

The Microsoft Foundation macros and globals, which are designed to assist both MS-DOS and Windows programmers, offer functionality in the following categories:

- Data types
- Run-time object-model services
- Diagnostic services
- Exception processing
- **CString** formatting and message-box display
- Message maps
- Application information and management
- Support for Object Linking and Embedding (OLE)
- Standard commands and window IDs

The first part of this section briefly discusses each of the above categories and lists each global and macro in the category, along with a short description of what it does. Following this is a complete alphabetical listing of all the global functions, global variables, and macros in the Microsoft Foundation classes.

The main supporting reference for the "Macros and Globals" section is the *Class Library User's Guide*. This is usually the first place you will look to find more information on macros and globals. When necessary, the appropriate chapter of the *Class Library User's Guide* is mentioned with the function or macro description.

Note All global functions start with the prefix "Afx." All global variables start with the prefix "afx." Macros do not start with any particular prefix, but they are all uppercase.

Data Types

This section lists the data types most commonly used in the Microsoft Foundation Class Library. Most of these data types are exactly the same as those in the *Windows Software Development Kit* (SDK) version 3.1, while others are unique to the Microsoft Foundation Class Library.

Commonly used Windows SDK and Microsoft Foundation class data types are as follows:

- **BOOL** A Boolean value.
- **BYTE** An 8-bit unsigned integer.
- **COLORREF** A 32-bit value used as a color value.
- **DWORD** A 32-bit unsigned integer or the address of a segment and its associated offset.
- **LONG** A 32-bit signed integer.
- **LPARAM** A 32-bit value passed as a parameter to a window procedure or callback function.
- **LPCSTR** A 32-bit pointer to a constant character string.
- **LPSTR** A 32-bit pointer to a character string.
- **LPVOID** A 32-bit pointer to an unspecified type.
- **LRESULT** A 32-bit value returned from a window procedure or callback function.
- **UINT** A 16-bit unsigned integer in Windows version 3.0 and later; a 32-bit unsigned integer in Win32.
- **WNDPROC** A 32-bit pointer to a window procedure.
- **WORD** A 16-bit unsigned integer.
- **WPARAM** A value passed as a parameter to a window procedure or callback function; 16 bits in Windows version 3.0 and later; 32-bits in Win32.

Data types unique to the Microsoft Foundation Class Library include:

- **POSITION** A value used to denote the position of an element in a collection; used by Microsoft Foundation collection classes.
- **LPCRECT** A 32-bit pointer to a constant (nonmodifiable) **RECT** structure.

For a list of the less common data types, see the Windows SDK reference.

Run-Time Object Model Services

The classes **CObject** and **CRuntimeClass** encapsulate several object services, including access to run-time class information, serialization, and dynamic object creation. All classes derived from **CObject** inherit this functionality.

Access to run-time class information enables you to determine information about an object's class at run time. The ability to determine the class of an object at run time is useful when extra type-checking of function arguments is needed and when you must write special-purpose code based on the class of an object. Run-time class information is not supported directly by the C++ language.

Serialization is the process of reading or writing an object's contents to and from a file. You can use serialization to store an object's contents even after the application exits. The object can then be read from the file when the application is restarted. Such data objects are said to be "persistent."

Dynamic object creation enables you to create an object of a specified class at run time. For example, document, view, and frame objects must support dynamic creation because the framework needs to create them dynamically.

The following table lists the Microsoft Foundation Class Library macros that support run-time class information, serialization, and dynamic creation. For more information on these run-time object services, see Chapter 12 of the *Class Library User's Guide*. For more information on serialization, see Chapter 14 of the *Class Library User's Guide*.

Run-Time Object Model Services

DECLARE_DYNAMIC	Enables access to run-time class information (must be used in the class declaration).
DECLARE_DYNCREATE	Enables dynamic creation and access to run-time class information (must be used in the class declaration).
DECLARE_SERIAL	Enables serialization and access to run-time class information (must be used in the class declaration).
IMPLEMENT_DYNAMIC	Enables access to run-time class information (must be used in the class implementation).
IMPLEMENT_DYNCREATE	Enables dynamic creation and access to run-time information (must be used in the class implementation).

IMPLEMENT_SERIAL	Permits serialization and access to run-time class information (must be used in the class implementation).
RUNTIME_CLASS	Returns the **CRuntimeClass** structure that corresponds to the named class.

Diagnostic Services

The Microsoft Foundation Class Library provides a range of diagnostic services that make debugging your programs easier. These diagnostic services include macros and global functions that allow you to track your program's memory allocations, dump the contents of objects during run time, and print debugging messages during run time. The macros and global functions for diagnostic services are grouped into the following categories:

- General diagnostic macros
- General diagnostic functions and variables
- Object diagnostic functions

These macros and functions are available for all classes derived from **CObject** in the Debug and Release versions of the Microsoft Foundation Class Library. However, all except **DEBUG_NEW** and **VERIFY** do nothing in the Release version.

In the Debug library, all allocated memory blocks are bracketed with a series of "guard bytes." If these bytes are disturbed by an errant memory write, then the diagnostic routines can report a problem. If you include the line

```
#define new DEBUG_NEW
```

in your implementation file, all calls to **new** will store the filename and line number where the memory allocation took place. The function **CMemoryState::DumpAllObjectsSince** will display this extra information, allowing you to identify memory leaks. Refer also to the class **CDumpContext** for additional information on diagnostic output.

For a general discussion of diagnostic facilities, see Chapter 15, "Diagnostics," in the *Class Library User's Guide*. For more information on the use of some of the key memory diagnostic functions, see the section "Detecting Memory Leaks" in Chapter 15 of the *Class Library User's Guide*.

General Diagnostic Macros

ASSERT	Prints a message and then aborts the program if the specified expression evaluates to **FALSE** in the Debug version of the library.
ASSERT_VALID	Tests the internal validity of an object by calling its **AssertValid** member function; typically overridden from **CObject**.
DEBUG_NEW	Provides a filename and line number for all object allocations in Debug mode to help find memory leaks.
TRACE	Provides **printf**-like capability in the Debug version of the library.
TRACE0	Similar to **TRACE** but takes a format string with no arguments.
TRACE1	Similar to **TRACE** but takes a format string with a single argument.
TRACE2	Similar to **TRACE** but takes a format string with two arguments.
TRACE3	Similar to **TRACE** but takes a format string with three arguments.
VERIFY	Similar to **ASSERT** but evaluates the expression in the Release version of the library as well as in the Debug version.

General Diagnostic Functions and Variables

afxDump	Global variable that sends **CDumpContext** information to the debugger output window or to the debug terminal.
afxMemDF	Global variable that controls the behavior of the debugging memory allocator.
afxTraceEnabled	Global variable used to enable or disable output from the **TRACE** macro
afxTraceFlags	Global variable used to turn on the built-in reporting features of the Microsoft Foundation Class Library.
AfxCheckMemory	Checks the integrity of all currently allocated memory.
AfxDump	Call this function while in the debugger to dump the state of an object while debugging.

AfxEnableMemoryTracking	Turns memory tracking on and off.
AfxIsMemoryBlock	Verifies that a memory block has been properly allocated.
AfxIsValidAddress	Verifies that a memory address range is within the program's bounds.
AfxIsValidString	Determines whether a pointer to a string is valid.
AfxSetAllocHook	Enables the calling of a function on each memory allocation.

Object Diagnostic Functions

AfxDoForAllClasses	Performs a specified function on all **CObject**-derived classes that support run-time type checking.
AfxDoForAllObjects	Performs a specified function on all **CObject**-derived objects that were allocated with **new**.

Exception Processing

When a program executes, a number of abnormal conditions and errors called "exceptions" can occur. These may include running out of memory, resource allocation errors, and failure to find files.

The Microsoft Foundation Class Library uses an exception-handling scheme that is modeled closely after the one proposed by the ANSI standards committee for C++. This involves setting up an exception handler before calling a function that may encounter an abnormal situation. If the function encounters an abnormal condition, it throws an exception and control is passed to the exception handler.

Several macros included with the Microsoft Foundation Class Library set up exception handlers. A number of other global functions help to throw specialized exceptions and terminate programs, if necessary. These macros and global functions fall into the following categories:

- Exception macros, which structure your exception handler
- Exception-throwing functions, which generate exceptions of specific types
- Termination functions, which cause program termination

For examples and more details, see Chapter 16, "Exceptions," in the *Class Library User's Guide*. You can also refer to class **CException**.

See Also **CException**

Exception Macros

TRY	Designates a block of code for exception processing.
CATCH	Designates a block of code for catching an exception from the preceding **TRY** block.
AND_CATCH	Designates a block of code for catching additional exception types from the preceding **TRY** block.
END_CATCH	Ends the last **CATCH** or **AND_CATCH** code block.
THROW	Throws a specified exception.
THROW_LAST	Throws the currently handled exception to the next outer handler.

Exception-Throwing Functions

AfxThrowArchiveException	Throws an archive exception.
AfxThrowFileException	Throws a file exception.
AfxThrowMemoryException	Throws a memory exception.
AfxThrowNotSupportedException	Throws a not-supported exception.
AfxThrowOleException	Throws an OLE exception.
AfxThrowResourceException	Throws a Windows resource-not-found exception.
AfxThrowUserException	Throws an exception in a user-initiated program action.

Termination Functions

AfxAbort	Called to terminate an application when a fatal error occurs.

CString Formatting and Message-Box Display

A number of functions are provided to format and parse **CString** objects. You can use these functions in any situation where you have to manipulate **CString** objects, but they are particularly useful for formatting strings that will appear in message-box text.

This group of functions also includes a global routine for displaying a message box.

Refer to class **CString** for more information about **CString** objects.

See Also **CString**

CString Functions

AfxFormatString1	Substitutes a given string for the format characters "%1" in a string contained in the string table.
AfxFormatString2	Substitutes two strings for the format characters "%1" and "%2" in a string contained in the string table.
AfxMessageBox	Displays a message box.

Message Maps

Since Windows is a message-oriented operating system, a large portion of programming for the Windows environment involves message handling. Each time an event such as a keystroke or mouse click occurs, a message is sent to the application, which must then handle the event.

The Microsoft Foundation Class Library offers a programming model optimized for message-based programming. In this model, "message maps" are used to designate which functions handle which messages for a particular class. Message maps contain one or more macros that specify which messages are handled by which functions. For example, a message-map containing an **ON_COMMAND** macro might look something like the following:

```
BEGIN_MESSAGE_MAP( CMyDoc, CDocument )
    //{{AFX_MSG_MAP( CMyDoc )
    ON_COMMAND( ID_MYCMD, OnMyCommand )
    // ... More entries to handle additional commands
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

The **ON_COMMAND** macro is used to handle command messages generated by menus, buttons, and accelerator keys. Macros are available to map the following:

Windows Messages

- Control notifications
- User-defined messages

Command Messages

- Registered user-defined messages
- User-interface update messages
- VBX event messages

Although message-map macros are important, you generally won't have to use them directly. This is because ClassWizard automatically creates message-map entries in your source files when you use it to associate message-handling functions with messages. Any time you want to edit or add a message-map entry, you can use ClassWizard.

However, since message maps are such an important part of the Microsoft Foundation Class Library, you should understand what they do, and documentation is therefore provided for them.

To support message maps, the Microsoft Foundation Class Library provides the following macros:

Message-Map Declaration and Demarcation

DECLARE_MESSAGE_MAP	Declares that a message map will be used in a class to map messages to functions (must be used in the class declaration).
BEGIN_MESSAGE_MAP	Begins the definition of a message map (must be used in the class implementation).
END_MESSAGE_MAP	Ends the definition of a message map (must be used in the class implementation).

Message-Mapping Macros

ON_COMMAND	Indicates which function will handle a specified command message.
ON_CONTROL	Indicates which function will handle a specified control-notification message.
ON_MESSAGE	Indicates which function will handle a user-defined message.
ON_REGISTERED_MESSAGE	Indicates which function will handle a registered user-defined message.
ON_UPDATE_COMMAND_UI	Indicates which function will handle a specified user-interface update command message.
ON_VBXEVENT	Indicates which function will handle a specified VBX control event message.

For more information on message maps and the above message-map macros, see Chapter 6 of the *Class Library User's Guide*. For more information on how to use ClassWizard, see Chapter 9 of the *App Studio User's Guide*.

Application Information and Management

When you write an application, you create a single **CWinApp**-derived object. At times, you may wish to get information about this object outside the **CWinApp**-derived object.

See Also **CWinApp**

The Microsoft Foundation Class Library provides the following global functions to help you accomplish these tasks:

Application Information and Management

AfxGetApp	Returns a pointer to the application's single **CWinApp** object.
AfxGetAppName	Returns a string containing the application's name.
AfxGetInstanceHandle	Returns an **HINSTANCE** representing this instance of the application.
AfxGetResourceHandle	Returns an **HINSTANCE** where the application loads its default resources; use this to access the application's resources directly.
AfxRegisterWndClass	Registers a Windows window class to supplement those registered automatically by the library.
AfxRegisterVBEvent	Registers a VB event of a specified name and returns an atom identifying the event.
AfxSetResourceHandle	Sets the **HINSTANCE** handle where the default resources of the application are loaded.

OLE Support

A number of functions are provided to help you write programs that use the Windows Object Linking and Embedding (OLE) mechanism. You can use these functions to provide the standard OLE user interface for client applications as well as a helper for automatic server registration.

In addition to these global functions, the Microsoft Foundation Class Library contains several classes that help you implement OLE functionality in your program. See Chapter 18 of the *Class Library User's Guide* and Techical Note 8 (which can be found in MSVC\HELP\MFCNOTES.HLP) for more information on using the OLE classes.

To use these macros and global functions, add the following directive at the top of your program or in your STDAFX.H header file:

```
#include <afxole.h>
```

See Also **CWinApp**

OLE Client Functions

AfxOleInsertDialog	Allows the user to choose an item type from a list of registered server applications.
AfxOleLinksDialog	Allows the user to update the client's OLE links.
AfxOleSetEditMenu	Implements the user interface for the *typename* Object command, allowing users to invoke verbs on OLE items.

OLE Server Functions

AfxOleRegisterServerName	Registers an application as an OLE server.

Standard Commands and Window IDs

The Microsoft Foundation Class Library defines a number of standard command and window IDs in AFXRES.H. These IDs are most commonly used within App Studio and ClassWizard to map messages to your handler functions. All standard commands have an **ID_** prefix. For example, when you use App Studio's menu editor, you normally bind the File Open menu item to the standard **ID_FILE_OPEN** command ID.

For most standard commands, application code does not need to refer to the command ID because the framework itself handles the commands through message-maps in its primary framework classes (**CWinApp**, **CView**, **CDocument**, and so forth).

In addition to standard command IDs, a number of other standard IDs are defined which have a prefix of **AFX_ID**. These IDs include standard window IDs (prefix **AFX_IDW_**), string IDs (prefix **AFX_IDS_**), and several other types.

IDs that begin with the **AFX_ID** prefix are rarely used by programmers, but you might, however, need to refer to these IDs when overriding framework functions which themselves refer to the **AFX_ID**s.

IDs are not individually documented in this reference. However, you can find more information on them in Technical Notes 20, 21, and 22, which can be found in MSVC\HELP\MFCNOTES.HLP.

Note The header file AFXRES.H is indirectly included in AFXWIN.H. You must explicitly include the statement

```
#include afxres.h
```

in your application's resource script (.RC) file.

Macros, Global Functions, and Global Variables

AfxAbort

void AfxAbort( );

Remarks

The default termination function supplied by the Microsoft Foundation Class Library. **AfxAbort** is called internally by Microsoft Foundation Class Library member functions when there is a fatal error, such as an uncaught exception that cannot be handled. You can call **AfxAbort** in the rare case when you encounter a catastrophic error from which you cannot recover.

AfxCheckMemory

BOOL AfxCheckMemory( );

Remarks

This function validates the free memory pool and prints error messages as required. If the function detects no memory corruption, it prints nothing.

All memory blocks currently allocated on the heap are checked, including those allocated by **new** but not those allocated by direct calls to underlying memory allocators such as the **malloc** function or the **GlobalAlloc** Windows function. If any block is found to be corrupted, a message is printed to the debugger output.

If you include the line

```
#define new DEBUG_NEW
```

in a program module, then subsequent calls to **AfxCheckMemory** show the filename and line number where the memory was allocated.

Note If your module contains one or more implementations of serializable classes, then you must put the `#define` line after the last **IMPLEMENT_SERIAL** macro invocation.

Return Value Nonzero if no memory errors; otherwise 0.

Example

```
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject.
Age* page = new Age( 22 );     // Age is NOT derived from CObject.
*(((char*) pcage) - 1) = 99;  // Corrupt preceding guard byte
*(((char*) page) - 1) = 99;   // Corrupt preceding guard byte
AfxCheckMemory();
```

The results from the program are as follows:

```
memory check error at $0067495F = $63, should be $FD
DAMAGE: before Non-Object block at $00674960
Non-Object allocated at file test02.cxx(48)
Non-Object located at $00674960 is 2 bytes long
memory check error at $00674905 = $63, should be $FD
DAMAGE: before Object block at $00674906
Object allocated at file test02.cxx(47)
Object located at $00674906 is 6 bytes long
```

Note This function only works in the Debug version of the Foundation library.

AfxDoForAllClasses

void AfxDoForAllClasses(void (*pfn**)(const CRuntimeClass*** *pClass*, **void*** *pContext*)**, void*** *pContext* **);**

pfn Points to an iteration function to be called for each class. The function arguments are a pointer to a **CRuntimeClass** object and a void pointer to extra data that the caller supplies to the function.

pContext Points to optional data that the caller can supply to the iteration function. This pointer can be **NULL**.

Remarks

Calls the specified iteration function for all **CObject**-derived classes in the application's memory space that support run-time type checking using the macros **DECLARE_DYNAMIC**, **DECLARE_DYNCREATE**, or **DECLARE_SERIAL**. The pointer that is passed to **AfxDoForAllClasses** in *pContext* is passed to the specified iteration function each time it is called.

Note This function only works in the Debug version of the Microsoft Foundation Class Library.

AfxDoForAllObjects

void AfxDoForAllObjects(void (*pfn***)(CObject*** *pObject*, **void*** *pContext*), **void*** *pContext* **);**

pfn Points to an iteration function to execute for each object. The function arguments are a pointer to a **CObject** and a void pointer to extra data that the caller supplies to the function.

pContext Points to optional data that the caller can supply to the iteration function. This pointer can be **NULL**.

Remarks

Executes the specified iteration function for all objects derived from **CObject** that have been allocated with **new**. Stack, global, or embedded objects are not enumerated. The pointer passed to **AfxDoForAllObjects** in *pContext* is passed to the specified iteration function each time it is called.

Note This function only works in the Debug version of the Foundation library.

afxDump

CDumpContext afxDump;

Remarks

Use this variable to provide basic object-dumping capability in your application. **afxDump** is a predefined **CDumpContext** object that allows you to send **CDumpContext** information to the debugger output window or to a debug terminal. Typically, you supply **afxDump** as a parameter to the **CObject::Dump** member function. You can also use the DBWin program (in the Windows SDK) to view the output of **afxDump**.

In Windows version 3.0 and later, **afxDump** output is sent to the debugger, if present. In MS-DOS, **afxDump** output is sent to **stderr**.

This variable is defined only in the Debug version of the Microsoft Foundation Class Library. For more information on **afxDump**, see Chapter 15 of the *Class Library User's Guide* and Technical Notes 7 and 12, which can be found in MSVC\HELP\MFCNOTES.HLP.

Note This function only works in the Debug version of the Foundation library.

See Also **CObject::Dump**

Example

```
CPerson myPerson = new CPerson;
// set some fields of the CPerson object...
//..
// now dump the contents
#ifdef _DEBUG
afxDump << "Dumping myPerson:\n";
myPerson->Dump( afxDump );
afxDump << "\n";
#endif
```

AfxDump

void AfxDump(const CObject* *pOb* **);**

pOb A pointer to an object of a class derived from **CObject**.

Remarks

Call this function while in the debugger to dump the state of an object while debugging. **AfxDump** calls an object's **Dump** member function and sends the information to the location specified by the **afxDump** variable. **AfxDump** is available only in the Debug version of the Microsoft Foundation Class Library.

Your program code should not call **AfxDump**, but should instead call the **Dump** member function of the appropriate object.

For example, the following command prints the state of the current object when you enter it at the > prompt in the CodeView® command window:

```
? AfxDump(this)
```

See Also **CObject::Dump**, **afxDump**

AfxEnableMemoryTracking

BOOL AfxEnableMemoryTracking(BOOL *bTrack* **);**

bTrack Setting this value to **TRUE** turns on memory tracking; **FALSE** turns it off.

Remarks

Diagnostic memory tracking is normally enabled in the Debug version of the Microsoft Foundation Class Library. Use this function to disable tracking on sections of your code that you know are allocating blocks correctly.

For more information on **AfxEnableMemoryTracking**, see Chapter 15 of the *Class Library User's Guide.*

Note This function only works in the Debug version of the Microsoft Foundation Class Library.

Return Value

The previous setting of the tracking-enable flag.

AfxFormatString1

void AfxFormatString1(CString& *rString*, **UINT** *nIDS*, **LPCSTR** *lpsz1* **);**

rString A reference to a **CString** object that will contain the resultant string after the substitution is performed.

nIDS The resource ID of the template string on which the substitution will be performed.

lpsz1 A string that will replace the format characters "%1" in the template string.

Remarks

Loads the specified string resource and substitutes the characters "%1" for the string pointed to by *lpsz1*. The newly formed string is stored in *rString*. For example, if the string in the string table is "File %1 not found", and *lpsz1* is equal to "C:\MYFILE.TXT", then *rString* will contain the string "File C:\MYFILE.TXT not found". This function is useful for formatting strings sent to message boxes and other windows.

If the format characters "%1" appear in the string more than once, multiple substitutions will be made.

See Also

AfxFormatString2

AfxFormatString2

void AfxFormatString2(CString& *rString*, **UINT** *nIDS*, **LPCSTR** *lpsz1*, **LPCSTR** *lpsz2* **);**

rString A reference to the **CString** that will contain the resultant string after the substitution is performed.

nIDS The string table ID of the template string on which the substitution will be performed.

lpsz1 A string that will replace the format characters "%1" in the template string.

lpsz2 A string that will replace the format characters "%2" in the template string.

Remarks

Loads the specified string resource and substitutes the characters "%1" and "%2" for the strings pointed to by *lpsz1* and *lpsz2*. The newly formed string is stored in *rString*. For example, if the string in the string table is "File %1 not found in directory %2", *lpsz1* points to "MYFILE.TXT", and *lpsz2* points to "C:\MYDIR", then rString will contain the string "File MYFILE.TXT not found in directory C:\MYDIR".

If the format characters "%1" or "%2" appear in the string more than once, multiple substitutions will be made. They do not have to be in numerical order.

See Also

AfxFormatString1

AfxGetApp

CWinApp* AfxGetApp( );

Remarks

The pointer returned by this function can be used to access application information such as the main message-dispatch code or the topmost window.

Return Value

A pointer to the single **CWinApp** object for the application.

AfxGetAppName

const char* AfxGetAppName( );

Remarks The string returned by this function can be used for diagnostic messages or as a root for temporary string names.

Return Value A null-terminated string containing the application's name.

AfxGetInstanceHandle

HINSTANCE AfxGetInstanceHandle( );

Remarks This function allows you to retrieve the instance handle of the current application. Unlike **AfxGetResourceHandle**, **AfxGetInstanceHandle** always returns the **HINSTANCE** of your executable (.EXE). **AfxGetResourceHandle** can return an instance handle to either your application's .EXE or a resource dynamic-link library (DLL).

Return Value An **HINSTANCE** to the current instance of the application.

See Also **AfxGetResourceHandle**, **AfxSetResourceHandle**

AfxGetResourceHandle

HINSTANCE AfxGetResourceHandle( );

Remarks Use the **HINSTANCE** handle returned by this function to access the application's resources directly, for example, in calls to the Windows function **FindResource**.

Return Value An **HINSTANCE** handle where the default resources of the application are loaded.

See Also **AfxGetInstanceHandle**, **AfxSetResourceHandle**

AfxIsMemoryBlock

BOOL AfxIsMemoryBlock(const void* *p*, **UINT** *nBytes*, **LONG*** *plRequestNumber* = **NULL);**

p Points to the block of memory to be tested.

nBytes Contains the length of the memory block in bytes.

plRequestNumber Points to a **long** integer that will be filled in with the memory block's allocation sequence number. The variable pointed to by *plRequestNumber* will only be filled in if **AfxIsMemoryBlock** returns nonzero.

Remarks

Tests a memory address to make sure it represents a currently active memory block that was allocated by the diagnostic version of **new**. It also checks the specified size against the original allocated size. If the function returns nonzero, the allocation sequence number is returned in *plRequestNumber*. This number represents the order in which the block was allocated relative to all other **new** allocations.

Return Value

Nonzero if the memory block is currently allocated and the length is correct; otherwise 0.

See Also

AfxIsValidAddress

Example

```
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject.
ASSERT( AfxIsMemoryBlock( pcage, sizeof( CAge ) ) )
```

AfxIsValidAddress

BOOL AfxIsValidAddress(const void FAR* *lp*, **UINT** *nBytes*, **BOOL** *bReadWrite* = **TRUE);**

lp Points to the memory address to be tested.

nBytes Contains the number of bytes of memory to be tested.

bReadWrite Specifies whether the memory is both for reading and writing (**TRUE**) or just reading (**FALSE**).

Remarks

Tests any memory address to ensure that it is contained entirely within the program's memory space. The address is not restricted to blocks allocated by **new**.

Return Value Nonzero if the specified memory block is contained entirely within the program's memory space; otherwise 0.

See Also **AfxIsMemoryBlock**, **AfxIsValidString**

AfxIsValidString

BOOL AfxIsValidString(LPCSTR *lpsz*, **int** *nLength* = –1 **);**

lpsz The pointer to test.

nLength Specifies the length of the string to be tested, in bytes. A value of –1 indicates that the string will be null-terminated.

Remarks Use this function to determine whether a pointer to a string is valid.

Return Value Nonzero if the specified pointer does not point to a string of the specified size; otherwise 0.

See Also **AfxIsMemoryBlock**, **AfxIsValidAddress**

afxMemDF

int afxMemDF;

Remarks This variable is accessible from a debugger or your program and allows you to tune allocation diagnostics. It can have the following values as specified by the enumeration **afxMemDF**:

- **allocMemDF** Turns on debugging allocator (default setting in Debug library).
- **delayFreeMemDF** Delays freeing memory. While your program frees a memory block, the allocator does not return that memory to the underlying operating system. This will place maximum memory stress on your program.
- **checkAlwaysMemDF** Calls **AfxCheckMemory** every time memory is allocated or freed. This will significantly slow memory allocations and deallocations.

Example

```
afxMemDF = allocMemDF | checkAlwaysMemDF;
```

AfxMessageBox

int AfxMessageBox(LPCSTR *lpszText*, **UINT** *nType* = **MB_OK,**
UINT *nIDHelp* = **0);**

int AFXAPI AfxMessageBox(UINT *nIDPrompt*, **UINT** *nType* = **MB_OK,**
UINT *nIDHelp* = **(UINT) –1);**

lpszText Points to a **CString** object or null-terminated string containing the message to be displayed in the message box.

nType The style of the message box (see the list of message-box styles below).

nIDHelp The Help-context ID for the message; 0 indicates no Help context.

nIDPrompt A unique ID used to reference a string in the string table.

Remarks

Displays a message box on the screen. The first form of this overloaded function displays a text string pointed to by *lpszText* in the message box and uses *nIDHelp* to describe a Help context. The Help context is used to jump to an associated Help topic when the user presses the Help key (typically F1).

The second form of the function uses the string resource with the ID *nIDPrompt* to display a message in the message box. The associated Help page is found through the value of *nIDHelp*. If *nIDHelp* is not specified, the string resource ID, *nIDPrompt*, is used for the Help context. For more information about defining Help contexts, see Chapter 10 of the *Class Library User's Guide* and Technical Note 28, which can be found in MSVC\HELP\MFCNOTES.HLP.

Return Value

Zero if there is not enough memory to display the message box; otherwise one of the following values is returned:

- **IDABORT** The Abort button was selected.
- **IDCANCEL** The Cancel button was selected.
- **IDIGNORE** The Ignore button was selected.
- **IDNO** The No button was selected.
- **IDOK** The OK button was selected.
- **IDRETRY** The Retry button was selected.
- **IDYES** The Yes button was selected.

If a message box has a Cancel button, the **IDCANCEL** value will be returned if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing the ESC key has no effect.

The message-box style given in the *nType* parameter can be any one of the following predefined constants:

Message-Box Styles

Message-Box Types

- **MB_ABORTRETRYIGNORE** The message box contains three pushbuttons: Abort, Retry, and Ignore.
- **MB_OK** The message box contains one pushbutton: OK.
- **MB_OKCANCEL** The message box contains two pushbuttons: OK and Cancel.
- **MB_RETRYCANCEL** The message box contains two pushbuttons: Retry and Cancel.
- **MB_YESNO** The message box contains two pushbuttons: Yes and No.
- **MB_YESNOCANCEL** The message box contains three pushbuttons: Yes, No, and Cancel.

Message-Box Modality

- **MB_APPLMODAL** The user must respond to the message box before continuing work in the current window. However, the user can move to the windows of other applications and work in those windows. **MB_APPLMODAL** is the default if neither **MB_SYSTEMMODAL** nor **MB_TASKMODAL** is specified.
- **MB_SYSTEMMODAL** All applications are suspended until the user responds to the message box. System-modal message boxes are used to notify the user of serious, potentially damaging errors that require immediate attention. They should be used sparingly.
- **MB_TASKMODAL** Similar to **MB_APPLMODAL**, but not useful within a Microsoft Foundation class application. This flag is reserved for a calling application or library that does not have a window handle available.

Message-Box Icons

- **MB_ICONEXCLAMATION** An exclamation-point icon appears in the message box.
- **MB_ICONINFORMATION** An icon consisting of an "i" in a circle appears in the message box.
- **MB_ICONQUESTION** A question-mark icon appears in the message box.
- **MB_ICONSTOP** A stop-sign icon appears in the message box.

Message-Box Default Buttons

- **MB_DEFBUTTON1** The first button is the default. Note that the first button is always the default unless **MB_DEFBUTTON2** or **MB_DEFBUTTON3** is specified.
- **MB_DEFBUTTON2** The second button is the default.
- **MB_DEFBUTTON3** The third button is the default.

The functions **AfxFormatString1** and **AfxFormatString2** can be useful to format text that appears in a message box.

See Also **CWnd::MessageBox**

AfxOleInsertDialog

BOOL AfxOleInsertDialog(CString& *name* **);**

name A reference to a **CString** object that will store the type name chosen by the user.

Remarks

Displays the Insert Object dialog box, which allows the user to insert a new embedded OLE item in a document. The dialog prompts the user to choose an OLE object or item type from a list of registered server applications and then invokes the specified application for the user to create the item. When the user exits the server application, an embedded item is inserted into the document. Call this function to implement the Insert Object command.

You must have the following statement in your client's application resource script (.RC) file:

```
#include <afxolecl.rc>
```

To add this include file to your .RC file, you should choose the Set Include item on App Studio's File menu and add "#include <afxolecl.rc>" to the list of compile-time directives.

Return Value

Nonzero if the user selected an item type; otherwise 0.

AfxOleLinksDialog

BOOL AfxOleLinksDialog(COleClientDoc* *pDoc* **);**

pDoc A pointer to the OLE client document that contains the links.

Remarks

Displays the Links dialog box, which displays all the OLE linked objects in the document and allows the user to update, cancel, or modify linked items. Call this function to implement the edit links command. Allows the user to update this client's OLE links.

You must have the following statement in your client's application resource script (.RC) file:

```
#include <afxolecl.rc>
```

To add this include file to your .RC file, you should choose the Set Include item on App Studio's File menu and add "#include <afxolecl.rc>" to the list of compile-time directives.

Return Value

Nonzero if successful; otherwise 0.

AfxOleRegisterServerName

BOOL AfxOleRegisterServerName(LPCSTR *lpszTypeName*, **LPCSTR** *lpszLocalTypeName* **);**

lpszTypeName The internal name of the document type supported by the OLE server. This name is used internally by the OLE system DLLs and the Windows registration database. This name cannot contain spaces.

lpszLocalTypeName A user-visible name of the document type supported by the OLE server. This name may be displayed by applications using the registration database. This name can contain spaces.

Remarks

Registers the application as an OLE server with the Windows registration database and allows the server to be launched if a client application requests it. This function updates the registration database with the current location of the application's executable file and, if the server has no registered verbs, specifies Edit as the primary verb.

You typically call this function only if you are writing a miniserver; if you are writing a full server, use the **COleTemplateServer** class to perform the registration for you. Call this function from the `InitInstance` member function of your **CWinApp**-derived class.

Return Value Nonzero if successful; otherwise 0.

See Also **COleTemplateServer::RunEmbedded**

AfxOleSetEditMenu

void AfxOleSetEditMenu(COleClientItem* *pClient*, **CMenu*** *pMenu*, **UINT** *iMenuItem*, **UINT** *nIDVerbMin* **);**

pClient A pointer to the client OLE item.

pMenu A pointer to the menu object that is to be updated.

iMenuItem The index of the menu item that is to be updated.

nIDVerbMin The command ID that corresponds to the primary verb.

Remarks Implements the user interface for the *typename* Object command. If the server recognizes only a primary verb, the menu item becomes "*verb typename* Object" and the *nIDVerbMin* command is sent when the user chooses the command. If the server recognizes several verbs, then the menu item becomes "*typename* Object" and a submenu listing all the verbs appears when the user chooses the command. When the user chooses a verb from the submenu, *nIDVerbMin* is sent if the first verb is chosen, *nIDVerbMin* + 1 is sent if the second verb is chosen, and so forth.

The default **COleClientDoc** implementation automatically handles this feature.

You must have the following statement in your client's application resource script (.RC) file:

```
#include <afxolecl.rc>
```

To add this include file to your .RC file, you should choose the Set Include item on App Studio's File menu and add "#include <afxolecl.rc>" to the list of compile-time directives.

AfxRegisterVBEvent

UINT AfxRegisterVBEvent(LPCSTR *lpszEventName* **);**

lpszEventName The name of the VB event.

Remarks

Registers a VB event of a specified name and returns an atom identifying the event. This function is usually used to define VB events for message mapping using a global initializer. For example:

```
UINT NEAR VBN_MYEVENT = AfxRegisterVBEvent("MyEvent");
```

Return Value

An atom identifying the event.

See Also

ON_VBXEVENT

AfxRegisterWndClass

const char* AfxRegisterWndClass(UINT *nClassStyle***,**
HCURSOR *hCursor* **= 0, HBRUSH** *hbrBackground* **= 0, HICON** *hIcon* **= 0);**

nClassStyle Specifies the Windows class style or combination of styles for the window class. This parameter can be any valid window style or control style, or a combination of styles created by using the bitwise-OR (**|**) operator. For a list of class styles, see the **WNDCLASS** structure in the Windows SDK documentation.

hCursor Specifies a handle to the cursor resource to be installed in each window created from the window class.

hbrBackground Specifies a handle to the brush resource to be installed in each window created from the window class.

hIcon Specifies a handle to the icon resource to be installed in each window created from the window class.

Remarks

The Microsoft Foundation Class Library automatically registers several standard window classes for you. Call this function if you want to register your own window classes.

Return Value

A null-terminated string containing the class name. You can pass this class name to the **Create** member function in **CWnd** or other **CWnd**-derived classes to create a window. The name is generated by the Microsoft Foundation Class Library.

Note The return value is a pointer to a static buffer. To save this string, assign it to a **CString** variable.

See Also

CWnd::Create, CWnd::PreCreateWindow

AfxSetAllocHook

AFX_ALLOC_HOOK AfxSetAllocHook(**AFX_ALLOC_HOOK** *pfnAllocHook* **);**

pfnAllocHook Specifies the name of the function to call. See below for the prototype of an allocation function.

Remarks

Sets a hook that enables calling of the specified function before each memory block is allocated. The hook function is described below.

Hook Function

The Microsoft Foundation Class Library debug-memory allocator can call a user-defined hook function to allow the user to monitor a memory allocation and to control whether the allocation is permitted. Allocation hook functions are prototyped as:

BOOL AllocHook(size_t *nSize*, **BOOL** *bObject*, **LONG** *lRequestNumber* **);**

nSize The size of the proposed memory allocation.

bObject **TRUE** if the allocation is for a **CObject**-derived object.

lRequestNumber The memory allocation's sequence number.

Return Value

Nonzero if you want to permit the allocation; otherwise 0.

AfxSetResourceHandle

void AfxSetResourceHandle(HINSTANCE *hInstResource* **);**

hInstResource The instance or module handle to a .EXE or DLL file from which the application's resources are loaded.

Remarks Use this function to set the **HINSTANCE** handle that determines where the default resources of the application are loaded.

See Also **AfxGetInstanceHandle**, **AfxGetResourceHandle**

AfxThrowArchiveException

void AfxThrowArchiveException(int *cause* **);**

cause Specifies an integer that indicates the reason for the exception. For a list of the possible values, see **CArchiveException::m_cause**.

Remarks Throws an archive exception.

See Also **CArchiveException**, **THROW**

AfxThrowFileException

void AfxThrowFileException(int *cause*, **LONG** *lOsError* = **–1);**

cause Specifies an integer that indicates the reason for the exception. For a list of the possible values, see **CFileException::m_cause**.

lOsError Contains the operating-system error number (if available) that states the reason for the exception. See your operating-system manual for a listing of error codes.

Remarks Throws a file exception. You are responsible for determining the cause based on the operating-system error code.

See Also **CFileException::ThrowOsError**, **THROW**

AfxThrowMemoryException

void AfxThrowMemoryException( );

Remarks Throws a memory exception. Call this function if calls to underlying system memory allocators (such as **malloc** and the **GlobalAlloc** Windows function) fail. You do not need to call it for **new** because **new** will throw a memory exception automatically if the memory allocation fails.

See Also **CMemoryException**, **THROW**

AfxThrowNotSupportedException

void AfxThrowNotSupportedException( );

Remarks Throws an exception that is the result of a request for an unsupported feature.

See Also **CNotSupportedException**, **THROW**

AfxThrowOleException

void AfxThrowOleException(OLESTATUS *status* **);**

status Indicates the reason for the exception. For a list of the possible values, see **COleException::m_status**.

Remarks Throws an OLE exception.

See Also **COleException**, **THROW**

AfxThrowResourceException

void AfxThrowResourceException( );

Remarks Throws a resource exception. This function is normally called when a Windows resource cannot be loaded.

See Also **CResourceException**, **THROW**

AfxThrowUserException

void AfxThrowUserException( );

Remarks

Throws an exception to stop an end-user operation. This function is normally called immediately after **AfxMessageBox** has reported an error to the user.

See Also

CUserException, **THROW**, **AfxMessageBox**

afxTraceEnabled

BOOL afxTraceEnabled;

Remarks

A global variable used to enable or disable output from the **TRACE** macro.

By default, output from the **TRACE** macro is disabled. Set **afxTraceEnabled** to a nonzero value if you want **TRACE** macros in your program to produce output. Set it to 0 if you don't want **TRACE** macros in your program to produce output.

Usually, the value of **afxTraceEnabled** is set in your AFX.INI file. Alternately, you can set the value of **afxTraceEnabled** with the TRACER.EXE utility. For more information on **afxTraceEnabled**, see Technical Note 7, which can be found in MSVC\HELP\MFCNOTES.HLP.

See Also

afxTraceFlags, **TRACE**

afxTraceFlags

int afxTraceFlags;

Remarks

Used to turn on the built-in reporting features of the Microsoft Foundation Class Library.

This variable can be set under program control or while using the debugger. Each bit of **afxTraceFlags** selects a trace reporting option. You can turn any one of these bits on or off as desired using TRACER.EXE. There is never a need to set these flags manually.

The following is a list of the bit patterns and the resulting trace report option:

- **0x01** Multiapplication debugging. This will prefix each **TRACE** output with the name of the application and affects both the explicit **TRACE** output of your program as well as the additional report options described below.
- **0x02** Main message pump. Reports each message received in the main **CWinApp** message-handling mechanism. Lists the window handle, the message name or number, **wParam**, and **lParam**.

  The report is made after the Windows **GetMessage** call but before any message translation or dispatch occurs.

  Dynamic data exchange (DDE) messages will display additional data that can be used for some debugging scenarios in OLE.

  This flag only displays messages that are posted—not those that are sent.
- **0x04** Main message dispatch. Like option **0x02** above but applies to messages dispatched in **CWnd::WindowProc**, and therefore handles both posted and sent messages that are about to be dispatched.
- **0x08** **WM_COMMAND** dispatch. A special case used for extended **WM_COMMAND/OnCommand** handling to report progress of the command-routing mechanism.

  Also reports which class receives the command (when there is a matching message-map entry), and when classes don't receive a command (when there is no matching message map entry). This report is especially useful to track the flow of command messages in multiple document interface (MDI) applications.
- **0x10** OLE tracing. Reports significant OLE notifications or requests.

  Turn this option on for an OLE client or server to track communication between the OLE DLLs and an OLE application.

For more information, see Technical Note 7, which can be found in MSVC\HELP\MFCNOTES.HLP.

See Also **afxTraceEnabled**, **TRACE**

AND_CATCH

AND_CATCH(*exception_class*, *exception_object_pointer_name* **)**

exception_class Specifies the exception type to test for. For a list of standard exception classes, see class **CException**.

exception_object_pointer_name A name for an exception-object pointer that will be created by the macro. You can use the pointer name to access the exception object within the **AND_CATCH** block. This variable is declared for you.

Remarks

Defines a block of code for catching additional exception types thrown in a preceding **TRY** block. Use the **CATCH** macro to catch one exception type, then the **AND_CATCH** macro to catch each subsequent type.

The exception-processing code can interrogate the exception object, if appropriate, to get more information about the specific cause of the exception. Invoke the **THROW_LAST** macro within the **AND_CATCH** block to shift processing to the next outer exception frame. **AND_CATCH** marks the end of the preceding **CATCH** or **AND_CATCH** block.

Note The **AND_CATCH** block is defined as a C++ scope (delineated by curly braces). If you declare variables in this scope, remember that they are accessible only within that scope. This also applies to the *exception_object_pointer_name* variable.

See Also

TRY, **CATCH**, **THROW**, **END_CATCH**, **THROW_LAST**, **CException**

ASSERT

ASSERT(*booleanExpression* **)**

booleanExpression Specifies an expression (including pointer values) that evaluates to nonzero or 0.

Remarks

Evaluates its argument. If the result is 0, the macro prints a diagnostic message and aborts the program. If the condition is nonzero, it does nothing.

The diagnostic message has the form:

```
assertion failed in file <name> in line <num>
```

where *name* is the name of the source file, and *num* is the line number of the assertion that failed in the source file.

In the Release version of the Microsoft Foundation Class Library, **ASSERT** does not evaluate the expression and thus will not interrupt the program. If the expression must be evaluated regardless of environment, use the **VERIFY** macro in place of **ASSERT**.

Note This function is available only in the Debug version of the Microsoft Foundation Class Library.

See Also **VERIFY**

Example

```
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject.
ASSERT( pcage!= NULL )
ASSERT( pcage->IsKindOf( RUNTIME_CLASS( CAge ) ) )
// Terminates program only if pcage is NOT a CAge*.
```

ASSERT_VALID

ASSERT_VALID(*pObject* **)**

pObject Specifies an object of a class derived from **CObject** that has an overriding version of the **AssertValid** member function.

Remarks

Use to test your assumptions about the validity of an object's internal state. **ASSERT_VALID** calls the **AssertValid** member function of the object passed as its argument.

In the Release version of the Microsoft Foundation Class Library, **ASSERT_VALID** does nothing. In the Debug version, it validates the pointer, checks against **NULL**, and calls the object's own **AssertValid** member functions. If any of these tests fails, this displays an alert message in the same manner as **ASSERT**.

Note This function is available only in the Debug version of the Microsoft Foundation Class Library.

For more information and examples, see Chapter 15 of the *Class Library User's Guide*.

See Also **ASSERT**, **VERIFY**, **CObject**, **CObject::AssertValid**

BEGIN_MESSAGE_MAP

BEGIN_MESSAGE_MAP(*theClass*, *baseClass* **)**

theClass Specifies the name of the class whose message map this is.

baseClass Specifies the name of the base class of *theClass*.

Remarks

Use the **BEGIN_MESSAGE_MAP** macro to begin the definition of your message map.

In the implementation (.CPP) file that defines the member functions for your class, start the message map with the **BEGIN_MESSAGE_MAP** macro, then add macro entries for each of your message-handler functions (see the listing under "Message Maps" on page 1053), and complete the message map with the **END_MESSAGE_MAP** macro.

For more information on message maps and the **BEGIN_MESSAGE_MAP** macro, see Chapter 6 of the *Class Library User's Guide*.

See Also

DECLARE_MESSAGE_MAP, **END_MESSAGE_MAP**

Example

```
BEGIN_MESSAGE_MAP( CMyWindow, CFrameWnd )
    //{{AFX_MSG_MAP( CMyWindow )
    ON_WM_PAINT()
    ON_COMMAND( IDM_ABOUT, OnAbout )
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

CATCH

CATCH(*exception_class*, *exception_object_pointer_name* **)**

exception_class Specifies the exception type to test for. For a list of standard exception classes, see class **CException**.

exception_object_pointer_name Specifies a name for an exception-object pointer that will be created by the macro. You can use the pointer name to access the exception object within the **CATCH** block. This variable is declared for you.

Remarks

Use this macro to define a block of code that catches the first exception type thrown in the preceding **TRY** block. The exception-processing code can interrogate the exception object, if appropriate, to get more information about the specific cause of the exception. Invoke the **THROW_LAST** macro to shift processing to the next outer exception frame.

If *exception_class* is the class **CException**, then all exception types will be caught. You can use the **CObject::IsKindOf** member function to determine which specific exception was thrown. A better way to catch several kinds of exceptions is to use sequential **AND_CATCH** statements, each with a different exception type.

The exception object pointer is created by the macro. You do not need to declare it yourself.

Note The **CATCH** block is defined as a C++ scope (delineated by curly braces). If you declare variables in this scope, remember that they are accessible only within that scope. This also applies to *exception_object_pointer_name*.

For more information on exceptions and the **CATCH** macro, see Chapter 16 of the *Class Library User's Guide*.

See Also **TRY**, **AND_CATCH**, **END_CATCH**, **THROW**, **THROW_LAST**, **CException**

DEBUG_NEW

#define new DEBUG_NEW

Remarks Assists in finding memory leaks. You can use **DEBUG_NEW** everywhere in your program that you would ordinarily use the **new** operator to allocate heap storage.

In Debug mode (when the **_DEBUG** symbol is defined), **DEBUG_NEW** keeps track of the filename and line number for each object that it allocates. Then, when you use the **CMemoryState::DumpAllObjectsSince** member function, each object allocated with **DEBUG_NEW** is shown with the filename and line number where it was allocated.

To use **DEBUG_NEW**, insert the following directive into your source files:

```
#define new DEBUG_NEW
```

Once you insert this directive, the preprocessor will insert **DEBUG_NEW** wherever you use **new**, and the Microsoft Foundation Class Library does the rest. When you compile a release version of your program, **DEBUG_NEW** resolves to a simple **new** operation, and the filename and line number information is not generated.

For more information on the **DEBUG_NEW** macro, see Chapter 15 of the *Class Library User's Guide.*

DECLARE_DYNAMIC

DECLARE_DYNAMIC(*class_name* **)**

class_name The actual name of the class (not enclosed in quotation marks).

Remarks

When deriving a class from **CObject**, this macro adds the ability to access run-time information about an object's class.

Add the **DECLARE_DYNAMIC** macro to the header (.H) module for the class, then include that module in all .CPP modules that need access to objects of this class.

If you use the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros as described, you can then use the **RUNTIME_CLASS** macro and the **CObject::IsKindOf** function to determine the class of your objects at run time.

If **DECLARE_DYNAMIC** is included in the class declaration, then **IMPLEMENT_DYNAMIC** must be included in the class implementation.

For more information on the **DECLARE_DYNAMIC** macro, see Chapter 12 of the *Class Library User's Guide.*

See Also

IMPLEMENT_DYNAMIC, DECLARE_DYNCREATE, DECLARE_SERIAL, RUNTIME_CLASS, CObject::IsKindOf

DECLARE_DYNCREATE

DECLARE_DYNCREATE(*class_name* **)**

class_name The actual name of the class (not enclosed in quotation marks).

Remarks

Use the **DECLARE_DYNCREATE** macro to enable objects of **CObject**-derived classes to be created dynamically at run time. The framework uses this ability to create new objects dynamically, for example, when it reads an object from disk during serialization. Document, view, and frame classes should support dynamic creation because the framework needs to create them dynamically.

Add the **DECLARE_DYNCREATE** macro in the .H module for the class, then include that module in all .CPP modules that need access to objects of this class.

If **DECLARE_DYNCREATE** is included in the class declaration, then **IMPLEMENT_DYNCREATE** must be included in the class implementation.

For more information on the **DECLARE_DYNCREATE** macro, see Chapter 12 of the *Class Library User's Guide*.

See Also **DECLARE_DYNAMIC**, **IMPLEMENT_DYNAMIC**, **IMPLEMENT_DYNCREATE**, **RUNTIME_CLASS**, **CObject::IsKindOf**

DECLARE_MESSAGE_MAP

DECLARE_MESSAGE_MAP( )

Remarks

Each **CCmdTarget**-derived class in your program must provide a message map to handle messages. Use the **DECLARE_MESSAGE_MAP** macro at the end of your class declaration. Then, in the .CPP file that defines the member functions for the class, use the **BEGIN_MESSAGE_MAP** macro, macro entries for each of your message-handler functions (see the listing under "Message Maps" on page 1053), and the **END_MESSAGE_MAP** macro.

For more information on message maps and the **DECLARE_MESSAGE_MAP** macro, see Chapter 6 of the *Class Library User's Guide*.

See Also **BEGIN_MESSAGE_MAP**, **END_MESSAGE_MAP**

Example

```
class CMyWnd : public CFrameWnd
{
    // Member declarations

    DECLARE_MESSAGE_MAP( )
};
```

Note If you declare any member after **DECLARE_MESSAGE_MAP**, you must specify a new access type (public, private, protected) for them.

DECLARE_SERIAL

DECLARE_SERIAL(*class_name* **)**

class_name The actual name of the class (not enclosed in quotation marks).

Remarks

DECLARE_SERIAL generates the C++ header code necessary for a **CObject**-derived class that can be serialized. Serialization is the process of writing or reading the contents of an object to and from a file.

Use the **DECLARE_SERIAL** macro in a .H module, then include that module in all .CPP modules that need access to objects of this class. For more information, see Chapter 12 of the *Class Library User's Guide*.

If **DECLARE_SERIAL** is included in the class declaration, then **IMPLEMENT_SERIAL** must be included in the class implementation. The **DECLARE_SERIAL** macro includes all the functionality of **DECLARE_DYNAMIC** and **DECLARE_DYNCREATE**.

For more information on the **DECLARE_SERIAL** macro, see Chapter 12 of the *Class Library User's Guide*.

See Also

DECLARE_DYNAMIC, **IMPLEMENT_SERIAL**, **RUNTIME_CLASS**, **CObject::IsKindOf**

END_CATCH

END_CATCH

Remarks

Marks the end of the last **CATCH** or **AND_CATCH** block.

For more information on the **END_CATCH** macro, see Chapter 16 of the *Class Library User's Guide*.

See Also

TRY, **CATCH**, **THROW**, **AND_CATCH**, **THROW_LAST**

END_MESSAGE_MAP

END_MESSAGE_MAP( )

Remarks

Use the **END_MESSAGE_MAP** macro to end the definition of your message map.

For more information on message maps and the **END_MESSAGE_MAP** macro, see Chapter 6 of the *Class Library User's Guide*.

See Also

DECLARE_MESSAGE_MAP, **BEGIN_MESSAGE_MAP**, **Message Map Function Categories**

IMPLEMENT_DYNAMIC

IMPLEMENT_DYNAMIC(*class_name*, *base_class_name* **)**

class_name The actual name of the class (not enclosed in quotation marks).

base_class_name The name of the base class (not enclosed in quotation marks).

Remarks

Generates the C++ code necessary for a dynamic **CObject**-derived class with run-time access to the class name and position within the hierarchy. Use the **IMPLEMENT_DYNAMIC** macro in a .CPP module, then link the resulting object code only once.

For more information, see Chapter 12 of the *Class Library User's Guide*.

See Also

DECLARE_DYNAMIC, **RUNTIME_CLASS**, **CObject::IsKindOf**

IMPLEMENT_DYNCREATE

IMPLEMENT_DYNCREATE(*class_name, base_class_name* **)**

class_name The actual name of the class (not enclosed in quotation marks).

base_class_name The actual name of the base class (not enclosed in quotation marks).

Remarks

Use the **IMPLEMENT_DYNCREATE** macro with the **DECLARE_DYNCREATE** macro to enable objects of **CObject**-derived classes to be created dynamically at run time. The framework uses this ability to create new objects dynamically, for example, when it reads an object from disk during serialization. Add the **IMPLEMENT_DYNCREATE** macro in the class implementation file. For more information, see Chapter 12 of the *Class Library User's Guide*.

If you use the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** macros, you can then use the **RUNTIME_CLASS** macro and the **CObject::IsKindOf** member function to determine the class of your objects at run time.

If **DECLARE_DYNCREATE** is included in the class declaration, then **IMPLEMENT_DYNCREATE** must be included in the class implementation.

See Also

DECLARE_DYNCREATE, **RUNTIME_CLASS**, **CObject::IsKindOf**

IMPLEMENT_SERIAL

IMPLEMENT_SERIAL(*class_name*, *base_class_name*, *wSchema* **)**

class_name The actual name of the class (not enclosed in quotation marks).

base_class_name The name of the base class (not enclosed in quotation marks).

wSchema A **UINT** "version number" that will be encoded in the archive to enable a deserializing program to identify and handle data created by earlier program versions. The class schema number must not be –1.

Remarks

Generates the C++ code necessary for a dynamic **CObject**-derived class with run-time access to the class name and position within the hierarchy. Use the **IMPLEMENT_SERIAL** macro in a .CPP module; then link the resulting object code only once.

For more information, see Chapter 12 of the *Class Library User's Guide*.

See Also

DECLARE_SERIAL, **RUNTIME_CLASS**, **CObject::IsKindOf**

ON_COMMAND

ON_COMMAND(*id*, *memberFxn* **)**

id The command ID.

memberFxn The name of the message-handler function to which the command is mapped.

Remarks

This macro is usually inserted in a message map by ClassWizard or manually. It indicates which function will handle a command message from a command user-interface object such as a menu item or toolbar button.

When a command-target object receives a Windows **WM_COMMAND** message with the specified ID, **ON_COMMAND** will call the member function *memberFxn* to handle the message.

There should be exactly one **ON_COMMAND** macro statement in your message map for every menu or accelerator command that must be mapped to a message-handler function.

For more information and examples, see Chapter 6 of the *Class Library User's Guide*.

See Also

ON_UPDATE_COMMAND_UI

Example

```
BEGIN_MESSAGE_MAP( CMyDoc, CDocument )
    //{{AFX_MSG_MAP( CMyDoc )
    ON_COMMAND( ID_MYCMD, OnMyCommand )
    // ... More entries to handle additional commands
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

ON_CONTROL

ON_CONTROL(*wNotifyCode*, *id*, *memberFxn* **)**

wNotifyCode The notification code of the control.

id The command ID.

memberFxn The name of the message-handler function to which the command is mapped.

Remarks

Indicates which function will handle a custom-control notification message. Control notification messages are those sent from a control to its parent window.

There should be exactly one **ON_CONTROL** macro statement in your message map for every control notification message that must be mapped to a message-handler function.

For more information and examples, see Chapter 6 of the *Class Library User's Guide*.

See Also

ON_MESSAGE, ON_REGISTERED_MESSAGE, ON_VBXEVENT

ON_MESSAGE

ON_MESSAGE(*message*, *memberFxn* **)**

message The message ID.

memberFxn The name of the message-handler function to which the message is mapped.

Remarks

Indicates which function will handle a user-defined message. User-defined messages are usually defined in the range **WM_USER** to 0x7FFF. User-defined messages are any messages that are not standard Windows **WM_MESSAGE** messages. There should be exactly one **ON_MESSAGE** macro statement in your message map for every user-defined message that must be mapped to a message-handler function.

For more information and examples, see Chapter 6 of the *Class Library User's Guide*.

See Also

ON_UPDATE_COMMAND_UI, ON_CONTROL, ON_REGISTERED_MESSAGE, ON_VBXEVENT, ON_COMMAND

Example

```
#define WM_MYMESSAGE (WM_USER + 1)
BEGIN_MESSAGE_MAP( CMyWnd, CMyParentWndClass )
    //{{AFX_MSG_MAP( CMyWnd
    ON_MESSAGE( WM_MYMESSAGE, OnMyMessage )
    // ... Possibly more entries to handle additional messages
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

ON_REGISTERED_MESSAGE

ON_REGISTERED_MESSAGE(*nMessageVariable*, *memberFxn* **)**

nMessageVariable The registered window-message ID variable.

memberFxn The name of the message-handler function to which the message is mapped.

Remarks

The Windows **RegisterWindowMessage** function is used to define a new window message that is guaranteed to be unique throughout the system. This macro indicates which function will handle the registered message.

The variable *nMessageVariable* should be declared with the **NEAR** modifier.

For more information and examples, see Chapter 6 of the *Class Library User's Guide*.

See Also

ON_MESSAGE, **ON_UPDATE_COMMAND_UI**, **ON_CONTROL**, **ON_VBXEVENT**, **ON_COMMAND**, **::RegisterWindowMessage**

Example

```
const UINT NEAR  wm_Find = RegisterWindowMessage( FINDMSGSTRING )
BEGIN_MESSAGE_MAP( CMyWnd, CMyParentWndClass )
    //{{AFX_MSG_MAP( CMyWnd )
    ON_REGISTERED_MESSAGE( wm_Find, OnFind )
    // ... Possibly more entries to handle additional messages
    //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

ON_UPDATE_COMMAND_UI

ON_UPDATE_COMMAND_UI(*id*, *memberFxn* **)**

id The message ID.

memberFxn The name of the message-handler function to which the message is mapped.

Remarks

This macro is usually inserted in a message map by ClassWizard to indicate which function will handle a user-interface update command message.

There should be exactly one **ON_UPDATE_COMMAND_UI** macro statement in your message map for every user-interface update command that must be mapped to a message-handler function.

For more information and examples, see Chapter 6 of the *Class Library User's Guide*.

See Also **ON_MESSAGE**, **ON_REGISTERED_MESSAGE**, **ON_CONTROL**, **ON_VBXEVENT**, **ON_COMMAND**, **CCmdUI**

ON_VBXEVENT

ON_VBXEVENT(*wNotifyCode*, *id*, *memberFxn* **)**

wNotifyCode The notification code of the VBX event.

id The message ID.

memberFxn The name of the message-handler function to which the message is mapped.

Remarks This macro is usually inserted in a message map by ClassWizard. It indicates which function will handle a message from a VBX control. There should be exactly one macro statement in your message map for every VBX-control message mapped to a message-handler function.

For more information and examples, see Chapter 6 of the *Class Library User's Guide*.

See Also **ON_MESSAGE**, **ON_UPDATE_COMMAND_UI**, **ON_CONTROL**, **ON_COMMAND**, **ON_REGISTERED_MESSAGE**, **AfxRegisterVBEvent**

RUNTIME_CLASS

RUNTIME_CLASS(*class_name* **)**

class_name The actual name of the class (not enclosed in quotation marks).

Remarks Use this macro to get the run-time class structure from the name of a C++ class.

RUNTIME_CLASS returns a pointer to a **CRuntimeClass** structure for the class specified by *class_name*. Only **CObject**-derived classes declared with **DECLARE_DYNAMIC**, **DECLARE_DYNCREATE**, or **DECLARE_SERIAL** will return pointers to a **CRuntimeClass** structure.

For more information, see Chapter 12 of the *Class Library User's Guide.*

See Also **DECLARE_DYNAMIC, DECLARE_DYNCREATE, DECLARE_SERIAL, CObject::GetRuntimeClass, CRuntimeClass**

Example

```
CRuntimeClass* prt = RUNTIME_CLASS( CAge );
ASSERT( lstrcmp( prt->m_lpszClassName, "CAge" )  == 0 );
```

THROW

THROW(*exception_object_pointer* **)**

exception_object_pointer Points to an exception object derived from **CException**.

Remarks Throws the specified exception. **THROW** interrupts program execution, passing control to the associated **CATCH** block in your program. If you have not provided the **CATCH** block, then control is passed to a Microsoft Foundation Class Library module that prints an error message and exits.

For more information, see Chapter 16 of the *Class Library User's Guide.*

See Also **TRY, CATCH, THROW, THROW_LAST, AND_CATCH, END_CATCH, AfxThrowArchiveException, AfxThrowFileException, AfxThrowMemoryException, AfxThrowNotSupportedException, AfxThrowOleException, AfxThrowResourceException, AfxThrowUserException**

THROW_LAST

THROW_LAST()

Remarks Throws the exception back to the next outer **CATCH** block.

This macro allows you to throw a locally created exception. If you try to throw an exception that you have just caught, it will normally go out of scope and be deleted. With **THROW_LAST**, the exception is passed correctly to the next **CATCH** handler.

For more information, see Chapter 16 of the *Class Library User's Guide.*

See Also **TRY, CATCH, THROW, AND_CATCH, END_CATCH**

TRACE

TRACE(*exp* **)**

exp Specifies a variable number of arguments that are used in exactly the same way that a variable number of arguments are used in the run-time function **printf.**

Remarks

Provides similar functionality to the **printf** function by sending a formatted string to a dump device such as a file or debug monitor. Like **printf** for C programs under MS-DOS, the **TRACE** macro is a convenient way to track the value of variables as your program executes. In the Debug environment, the **TRACE** macro output goes to **afxDump**. In the Release environment, it does nothing.

Note This macro is available only in the Debug version of the Microsoft Foundation Class Library.

For more information, see Chapter 15 of the *Class Library User's Guide.*

See Also

TRACE0, **TRACE1**, **TRACE2**, **TRACE3**, **AfxDump**, **afxTraceEnabled**

Example

```
int i = 1;
char sz[] = "one";
TRACE( "Integer = %d, String = %s\n", i, sz );
// Output: 'Integer = 1, String = one'
```

TRACE0

TRACE0(*exp* **)**

exp A format string as used in the run-time function **printf**.

Remarks

Similar to **TRACE**, but places the trace string in a code segment rather than DGROUP, thus using less DGROUP space. **TRACE0** is one variant of a group of trace macros that you can use for debug output. This group includes **TRACE0**, **TRACE1**, **TRACE2**, and **TRACE3**. The difference between these macros is the number of parameters taken. **TRACE0** only takes a format string and can be used for simple text messages. **TRACE1** takes a format string plus one argument—a variable to be dumped. Likewise, **TRACE2** and **TRACE3** take two and three parameters after the format string, respectively.

TRACE0 does nothing if you have compiled a release version of your application. As with **TRACE**, it only dumps data to **afxDump** if you have compiled a debug version of your application.

Note This macro is available only in the Debug version of the Microsoft Foundation Class Library.

Example

```
TRACE0( "Start Dump of MyClass members:" );
```

See Also **TRACE, TRACE1, TRACE2, TRACE3**

TRACE1

TRACE1(*exp*, *param1* **)**

exp A format string as used in the run-time function **printf**.

param1 The name of the variable whose value should be dumped.

Remarks See **TRACE0** for a description of the **TRACE1** macro.

Example

```
int i = 1;
TRACE1( "Integer = %d\n", i );
// Output: 'Integer = 1'
```

TRACE2

TRACE2(*exp*, *param1*, *param2* **)**

exp A format string as used in the run-time function **printf**.

param1 The name of the variable whose value should be dumped.

param2 The name of the variable whose value should be dumped.

Remarks See **TRACE0** for a description of the **TRACE2** macro.

Example

```
int i = 1;
char sz[] = "one";
TRACE2( "Integer = %d, String = %s\n", i, sz );
// Output: 'Integer = 1, String = one'
```

TRACE3

TRACE3(*exp*, *param1*, *param2*, *param3* **)**

exp A format string as used in the run-time function **printf**.

param1 The name of the variable whose value should be dumped.

param2 The name of the variable whose value should be dumped.

param3 The name of the variable whose value should be dumped.

Remarks

See **TRACE0** for a description of the **TRACE3** macro.

TRY

TRY

Remarks

Use this macro to set up a **TRY** block. A **TRY** block identifies a block of code that might throw exceptions. Those exceptions are handled in the following **CATCH** and **AND_CATCH** blocks. Recursion is allowed: exceptions may be passed to an outer **TRY** block, either by ignoring them or by using the **THROW_LAST** macro.

For more information, see Chapter 16 of the *Class Library User's Guide*.

See Also

THROW, **CATCH**, **AND_CATCH**, **END_CATCH**

VERIFY

VERIFY(*booleanExpression* **)**

booleanExpression Specifies an expression (including pointer values) that evaluates to nonzero or 0.

Remarks

In the Debug version of the Microsoft Foundation Class Library, the **VERIFY** macro evaluates its argument. If the result is 0, the macro prints a diagnostic message and halts the program. If the condition is nonzero, it does nothing.

The diagnostic message has the form:

```
assertion failed in file <name> in line <num>
```

where *name* is the name of the source file and *num* is the line number of the assertion that failed in the source file.

In the Release version of the Microsoft Foundation Class Library, **VERIFY** evaluates the expression but does not print or interrupt the program. For example, if the expression is a function call, the call will be made.

See Also **ASSERT**

Index

B

C

D

E

F

G

H

I

J

K

N

O

P

Q

R

S

T

U

V

W

Y

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-63

Microsoft®

Recyclable

* 3 5 7 4 3 *