Class Libraries Reference

Microsoft® C/C++

Microsoft® C/C++

Version 7.0

Class Libraries Reference

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Document No. LN24776-1291

10 9 8 7 6 5 4 3 2 1

Contents Overview

Contents

Introduction

This *Class Libraries Reference* covers the two class libraries that are included with Microsoft® C/C++. The book is divided into three parts:

Part 1	Introduction to the Microsoft Foundation Class Library
Part 2	The Microsoft Foundation Class Reference
Part 3	The Microsoft iostream Class Reference

Part 1 contains overview material for the Windows™ and general-purpose classes in the Microsoft Foundation Class Library followed by an alphabetical listing of all global functions and macros. In addition, it contains reference chapters for Microsoft Foundation Class Library diagnostic services and exception processing. The last two chapters consist of a Windows message map cross-reference and a listing of structures and enumerated values for Windows.

Parts 2 and 3 both begin with class hierarchy diagrams for their respective libraries. These hierarchy diagrams, together with the subset diagrams included with each Foundation 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.

Parts 2 and 3 list classes in alphabetical order. 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
- Manipulators (iostream classes only)

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.

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.

Note The term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.

Document Conventions

This book uses the following typographic conventions:

Example	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 class libraries, macros, flags, data structures and their members, and enumerators.
	Within descriptions of syntax, bold type indicates that the text must be entered exactly as shown.
	Many functions and constants begin with either a single or a double underscore. These are part of the name and are mandatory.
expression	Words in italics indicate placeholders for information you must supply, such as a filename.
[[*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`	Monospace font is used for examples, user input, program output, and error messages in text. It is also used for names of user-derived classes and members.
CL [[*option...*]] *file...*	Three dots (an ellipsis) following an item indicate that more items having the same form may appear.

Example	Description

`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.

Introduction to the Microsoft Foundation Class Library

Part 1

Introduction to the Microsoft Foundation Class Library

This part contains both overview and reference material for the Microsoft Foundation Class Library. Chapters 1 and 2 categorize and summarize the Windows and general-purpose classes. They are useful both as an introduction and as a reference guide for "finding the right class."

Chapter 3 is an alphabetic summary of all Microsoft Foundation Class Library global functions and macros. Chapter 4 lists, in more detail, those global functions and macros pertaining to memory diagnostics. Note that the class reference in Part 2 covers class-specific diagnostic features. Chapter 5 lists global functions and macros used for exception processing.

Chapter 6 is a cross-reference of Windows message-map parameters and corresponding **CWnd** member functions.

Chapter 7 lists Windows structures and enumerated values for the Clipboard and mouse.

Windows Development with the Microsoft Foundation Classes

Chapter 1

This chapter categorizes and describes the classes within the Microsoft Foundation Class Library that specifically support application development for Microsoft Windows, version 3.x.

1.1 Class Summary

The following is a list, in functional order, of the Windows-oriented classes in the Microsoft Foundation Class Library.

Note All classes listed below, except **CPoint**, **CRect**, and **CSize**, are directly or indirectly derived from the **CObject** class described in Chapter 2.

Main Application Class

CWinApp is the class that encapsulates the code for the initialization, running, and termination of the application.

Window Classes

The Microsoft Foundation window classes are the key building blocks in a Windows application. These classes have member functions for processing Windows notification messages as well as messages from other classes. Some member functions communicate directly with Windows itself. An active C++ window object contains a Windows **HWND**.

You will usually derive classes from the frame and child base classes. You can use most of the other window classes directly.

Base Class

CWnd	The base class for all windows.

Frame and Child Windows

CFrameWnd	The main window base class for the single document interface (SDI) frame window.
CMDIFrameWnd	The base class for the multiple document interface (MDI) frame window.
CMDIChildWnd	The base class for MDI child windows.

Dialog Windows

CDialog	The base class for modeless dialog windows.
CModalDialog	The base class for modal dialog windows.

Control Windows

CButton	Button control windows.
CComboBox	Combo-box control windows.
CEdit	Edit control windows.
CListBox	List-box control windows.
CScrollBar	Scroll-bar control windows.
CStatic	Static control windows.

Graphics Device Interface (GDI) Classes

The following classes wrap the Windows device context and drawing tools. They allow the developer to take maximum advantage of C++ syntax.

Device Contexts

CDC	The base class for device contexts, used directly for whole-display and nondisplay contexts.
CClientDC	Display contexts for client areas of windows.
CMetaFileDC	Metafile device contexts.
CPaintDC	Display contexts used in **OnPaint** member functions.
CWindowDC	Display contexts for entire windows.

GDI Drawing Objects

CGdiObject	The base class for GDI drawing tools.
CBitmap	GDI physical bitmaps.
CBrush	GDI physical brushes.
CFont	GDI physical fonts.
CPalette	GDI physical palettes.
CPen	GDI physical pens.
CRgn	GDI physical regions.

Other Classes

CMenu	Menu structures.
CPoint	Coordinate (x, y) pairs.
CRect	Rectangular areas.
CSize	Relative positions or coordinate pairs.

Windows Global Functions and Macros

Chapters 3 through 7 of this manual document the elements of the Microsoft Foundation Class Library that are not directly related to individual classes. A complete summary of macros and global functions, including those for Windows, is provided in Chapter 3. A message-map reference is given in Chapter 6, and Chapter 7 lists Windows structures and enumerated values.

1.2 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.

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

- Execution speed comparable to that of the C-language API
- Minimum code size overhead
- The ability to call any Windows C function directly

- Easy conversion of existing C applications to C++
- The ability to leverage from the existing base of C-language Windows programming experience
- True Windows API for C++ that effectively uses C++ language features
- Solid foundation for future extensions

The single characteristic that sets the Microsoft Foundation classes for Windows apart from other Windows class libraries is their direct access to the C-language Windows API. This direct access does *not*, however, imply that the classes are a *replacement* for that API. Developers must still 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 often need to make native Windows function calls, you should have access to the C-language Windows API documentation. This is included with Microsoft C/C++ as Help. If you require printed documentation, refer to the *Microsoft Windows Programmer's Reference* and the *Microsoft Windows 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 Windows classes.

1.3 C++ and Windows

Many C++ language features are particularly suited to Microsoft Windows. The Windows-oriented classes in the Microsoft Foundation Class Library make Windows programming truly systematic. It's significantly easier to learn Windows through a C++ interface than through the standard C interface.

Message-Based Programming

Windows is a message-based environment. In the familiar MS-DOS programming world, your program calls the operating system. In Windows, the operating system (Windows) sends a message to your program. The "program" is associated with a particular window, and the message might be "destroy yourself," "repaint yourself," "your child button was pushed," or something similar.

Your program might also send a message to Windows. These "outbound" messages are often directed at a child window, such as a button, list box, or edit control, that has inaccessible code. If, for example, you send a "scroll" message to an edit control, Windows itself does the work. Your application program cannot intercept these outbound messages once they are sent.

The C++ language naturally accommodates the messaging behavior of Windows. Objects receive messages through the "member functions" of their class, and they send messages by calling a member function for another object. A C++ object represents a Windows window, and the member functions of the class process individual messages. The **OnPaint** member function of a derived window class, for example, *receives* and processes a Windows **WM_PAINT** message. The **CListBox** member function **AddString** *sends* an **LB_ADDSTRING** message to Windows.

The real magic of the Windows Foundation classes begins here. There is no longer any "program logic flow" as in conventional procedural programming. Each window object is self-sufficient and is responsible for (1) acting on the messages that are important to it and (2) sending messages to other window objects. It can create and delete other windows along the way. The interaction among window objects, and thus the flow of the program, is governed by the actions of the end user rather than by complex code and data structures.

Message Processing–The Microsoft Foundation Classes vs. Native Windows

In native Windows, the **WndProc** function processes a particular window's incoming messages. The message ID is a **WndProc** function parameter that is decoded with a **case** statement. The ID is compared to a list of expected codes defined as constants in WINDOWS.H. Each message has two "message parameters," **wParam** (two bytes) and **lParam** (four bytes), that are also **WndProc** parameters. The meaning of **wParam** and **lParam** depends on the message type. These message parameters can be pointers to structures or functions or they can be composites of flags and fields. The **WndProc** program must decode the messages appropriately.

Outbound messages are sent to Windows and to other windows through several "send message" functions. The **WndProc** program must encode the message parameters for these outbound messages.

The Microsoft Foundation classes replace the **case** statement and parameter decoding for incoming messages with class member functions. These member functions are linked to a structure called a "message map" that governs translation of the message parameters. The message map is described later in this chapter, in "Notification Messages and the Message Map," on page 13.

Class Derivation

Programmers often try to exploit existing code to solve new problems. In the C programming environment, the programmer can "clone" useful code by copying it and making modifications. In the C++ programming environment, you add functionality through "class derivation". The functionality of the base class remains unmodified, but that of the derived class may be added to or changed. Derivation works well with Windows because you can extend useful window base classes with new member functions and new data members.

Suppose a frame window base class includes a caption, menu bar, scroll bars, and so forth. Also suppose that this base window has the ability to get the input focus in response to activation by the mouse. If you need to add the capability of displaying a dialog box in response to an access key, then you can derive a new class from the frame window base class. You get all the base class functionality without having to modify its code or worry about its internals.

There are three or more levels of window class derivation in a typical application built with the Microsoft Foundation classes for Windows. **CWnd**, the base class for all windows, contains many member functions that apply to all window types. Some second-level derived window classes, such as **CFrameWnd**, **CMDIFrameWnd**, and **CMDIChildWnd**, are designed for further derivation. Others, such as **CDialog**, can be used directly or as a base for further derivation. Finally, the classes that you derive for your own windows provide the third level of derivation.

Note The **CWnd** class is useful as a base class for SDI child windows. You do not need to derive from one of the second-level classes listed above.

Polymorphism

In the C-language Windows API, the programming interface of a dialog box is different from that of a frame window, even though both are defined as windows and identified by an **HWND** parameter. The Microsoft Foundation classes make all window types look similar because all are derived from the **CWnd** base class. Microsoft Foundation window classes are truly polymorphic because they give a common programming interface to dissimilar window types.

Reduced Programming "Surface Area"

C programming with Windows is intimidating because of the complex interrelationship between functions and the proliferation of messages. The progammer must write a **WndProc** function for each type of window and a main program called **WinMain**. Each **WndProc** function processes the window's messages by means of a **case** statement and must be linked to windows and to the application through an elaborate data structure.

The Microsoft Foundation classes for Windows encapsulate most of this complexity while allowing the same flexibility found in the C programming environment. You don't need to write the **WinMain** and **WndProc** functions because they are provided for you. However, you can override them if necessary.

1.4 Windows Class Categories

Like any C++ class library, the Microsoft Foundation Class Library encapsulates its functionality in classes. The important Windows class categories are:

- The main application class, **CWinApp**
- The window classes–**CWnd** and its derived classes
- The graphics device interface (GDI) classes, which support device contexts and drawing tools
- The miscellaneous classes, which support menus, points, and rectangles

The Main Application Class, CWinApp

The main application class encapsulates the initialization, running, and termination of a Windows application. A Microsoft Foundation Class Library Windows application must contain one (and only one) object of a class derived from **CWinApp**. This class has several important member functions that you can override:

- **InitInstance**

  Windows allows you to run more than one copy, or "instance," of the same application. **InitInstance** is called every time a new instance of the program starts. It *must* be overridden in order to create a main window and thus start the application. It is the most important member function of the class.

- **InitApplication**

  This function is called when the *first* instance of a program starts. The default version does nothing, but you can override it if you need special processing for the first instance only.

- **ExitInstance**

  This function is called each time an application instance terminates, usually as a result of the user quitting the application. You can override **ExitInstance** if you need special cleanup processing, such as closing of disk files or deallocating memory used during program execution.

- **OnIdle**

  The default version of **OnIdle** does nothing, but your overridden function can perform background tasks when no messages are being processed.

For a typical Windows application, you need only override **InitInstance** in a class derived from **CWinApp**. Then you construct a static object of the **CWinApp**-derived class.

Program Initialization–The Foundation Classes vs. Native Windows

Native Windows always starts your application by calling the **WinMain** function, which, in turn, creates a main window. Like other windows, this main window has an associated **WndProc** function that processes the window's messages.

WndProc functions are associated with windows by means of a "window class registration" procedure. The main window is registered in the **WinMain** function, but other window classes 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 in a call to the function **RegisterClass**, which returns a string that is used in the window creation process. Thus a registration class can be shared by multiple windows.

The Microsoft Foundation classes include a special version of **WinMain** that calls member functions of class **CWinApp**. An object of a **CWinApp**-derived class is constructed prior to the execution of **WinMain**. The **CWinApp** constructor ensures that there is only one object of the class. The **CWinApp** member function **Run** intercepts Windows messages in order to do key code translation and other special processing. The **Run** function also calls **OnIdle** and **ExitInstance**.

In a Microsoft Foundation Class Library Windows application, there is only one **WndProc** function. This built-in function processes all incoming Windows messages through the message map. Most of the time, you won't have to worry about the registration of Windows classes because the Microsoft Foundation Class Library registers objects for you with default parameters. If you do need a special cursor, background brush, or other feature not supported by the window **Create** function, you can register your own Windows registration class with the global **AfxRegisterWndClass** function.

The Window Classes—CWnd and Its Derived Classes

You will normally override the **InitInstance** member function of the **CWinApp** class to create your application's main window, an object of a class derived from **CWnd**. This window class, together with all the other window classes, have member functions for receiving and sending messages.

There are different types of Windows messages. Each type is handled somewhat differently by the Microsoft Foundation classes.

Notification Messages and the Message Map

A "notification message" is a message sent to a window by Windows itself in response to a keystroke, mouse click, window move, control window activity, or other event. If necessary, your application can force Windows to send a notification message. The Microsoft Foundation Class Library has a special mechanism, called a "message map," that links Windows notification messages with the member functions you have written.

A message map is a table that you include with your window class code. It contains an entry for each Windows notification message that you intend to process with a custom-written member function. The result is a message-processing system that provides all the advantages of virtual functions without the storage overhead.

Many of the member functions that process notification messages are predefined. For example, if your class needs to process the Windows **WM_CREATE** message, you must put the following entry in the class's message map:

```
ON_WM_CREATE()
```

and you must declare and implement this exact member function:

```
afx_msg int OnCreate( LPCREATESTRUCT lpcs );
```

Note The **afx_msg** keyword denotes a notification message declaration or implementation. It is defined as a no-operation in AFXWIN.H and thus documents the fact that the function behaves like a **CWnd** virtual function. It must be emphasized that message maps depend solely on standard preprocessor macros and not on any extensions to the C++ language.

A table that shows all permitted message-map entries and the corresponding member function prototypes is presented in Chapter 6, "Message Map Cross-Reference." For a complete example of message-map usage, see Chapter 6 of the *Class Libraries User's Guide.*

Other notification messages allow you to define your own functions. For example, if you need to call a function in response to a mouse click on a button whose ID number is `IDD_BUTN1`, your message-map entry is:

```
ON_BN_CLICKED( IDD_BUTN1, OnTopButton )
```

and your member function declaration looks like this:

```
afx_msg void OnTopButton();
```

When you define a message map, you specify the base class in addition to the messages. This allows the base class to handle messages not handled in the derived classes.

Windows Control Messages

A Windows control, such as an edit window or list box, is represented by a window object (of a class derived from **CWnd**), but the processing is controlled by Windows rather than by the Microsoft Foundation classes. If you need to update a control, your application must send a "control message" to Windows. A frame window object, for example, can send an **LB_ADDSTRING** message to a list-box window that is its child. The Microsoft Foundation classes wrap this message in the **CListBox** member function **AddString**. The call looks like this:

```
CListBox* listBox1; // Object initialized elsewhere
listBox1->AddString( "list-box line item" );
```

In this case the phrase `list-box line item` is sent directly to Windows for display. The **CListBox** member functions cannot access the list-box line items unless they retrieve them directly from Windows.

Other Windows Messages

Other Windows messages do not relate to controls. If, for example, you want to select a font for future text drawing, you must send the **WM_SETFONT** message to the appropriate window. The Microsoft Foundation classes wrap messages like this with **CWnd** member functions. In this example, you call the **SetFont** member function.

C++ Window Objects and the Windows They Represent

A C++ window object is distinct from its corresponding Windows window, but the two are tightly linked. A good understanding of this relationship is crucial for effective Microsoft Foundation class programming.

The window *object* is an instance of the C++ **CWnd** class (or a derived class). It comes and goes in response to your program's constructor and destructor calls. The Windows *window*, on the other hand, is an internal Windows data structure that corresponds to a visible (or invisible) window. A Windows window is identified by a "window handle" (**HWND**) and is created when the **CWnd** object is created, but the window may be destroyed by either a program call or by a user's action.

All the window classes provided by the Microsoft Foundation Class Library employ "two-phase construction". The C++ constructor makes an object but does not create a corresponding Windows window. The **Create** member function makes the Windows window (usually by calling the native Windows **CreateWindow** function) and stores its **HWND** value in the C++ object's public data member **m_hWnd**.

If you derive a window class that will, in turn, be used for derivation, you *must* use two-phase construction. Suppose, instead, that your higher-level constructor called **Create**. In that case, the lower-level constructor creates the Windows window before the object (and its message-map linkage) is completely constructed. The ensuing **WM_CREATE** and **WM_PAINT** messages are then handled by the higher-level class's message-handling functions rather than by those of the lower-level class.

The **CWnd** virtual destructor destroys the Windows window. If you need to circumvent the object-**HWND** relationship, the Microsoft Foundation Class Library provides several options. The function **DestroyWindow**, one of the few public virtual member functions in **CWnd**, destroys the Windows window without destroying the object. Another **CWnd** member function, **Detach**, prevents the destructor from destroying the Windows window.

You must be careful, particularly with child windows, to destroy the C++ window object when the user closes a Windows window. If you do not destroy these objects, you will not recover their memory. Main window object destruction is not so critical because the program generally terminates immediately after the main window is closed.

Some messages are not wrapped by individual member functions. Suppose you derive a class from **CWnd** and you need a member function that activates the window's nonclient area. The following call accomplishes the task:

```
SendMessage( WM_NCACTIVATE, TRUE, 0 );
```

SendMessage is a **CWnd** member function, and thus it communicates directly with your C++ window object that underlies the Windows window. The **SendMessage** function sends its message immediately. Another function, **PostMessage**, posts the message to the Windows message queue for delayed processing.

Direct Calls to Windows

In many cases your program interacts with Windows through a direct call rather than through a sent message. Many of these direct calls are mapped to **CWnd** member functions. If you need to set a window's caption, for example, you simply call **CWnd::SetWindowText**.

Control Window Classes

The Microsoft Foundation Class Library includes classes for standard Windows controls, which are actually special-purpose windows. These controls include:

- Buttons
- Combo boxes
- Edit windows
- List boxes
- Scroll bars
- Static controls

You will seldom need to derive from these classes because most of their functionality is determined by Windows itself. If you need a button, for example, you construct an object and then specify one of several predefined styles to the **Create** member function.

Buttons, like other controls, are designed to be child windows. When the user clicks a button, for example, the button object sends a **BN_CLICKED** message to its parent window object. The parent window class must define a message map and have an appropriate member function to handle the message from the button.

Note Do not confuse the scroll-bar control window with the frame window's built-in scroll bars. Any frame window can have horizontal and vertical scroll bars if it is created with the proper parameters. The scroll bars created in a frame window this way are not actually separate child windows. A true scroll-bar control is a separate child window that can be sized and placed as required.

Dialog Boxes

A "dialog box" is a special kind of frame window that contains a number of child window controls, such as buttons and edit fields. It is generally used to collect data from the user. A "modal" dialog forces the user to complete the requested action prior to returning to the application's main window. The "modeless" dialog allows the user to continue work in the parent window.

Dialog boxes are frequently defined, along with the constituent child windows, in "resource files" where the child windows have assigned ID numbers. Your program must construct an object of class **CDialog** or **CModalDialog** (or a derived class) in order to use a resource-based dialog box. If your dialog box requires only routine operations, such as detecting button hits and reading input strings from edit controls, then you do not have to construct child control window objects; instead, you call member functions of the dialog base class that use child window IDs as arguments.

If, for example, you need to read the string from an edit window identified as `IDM_DATA`, then use the **CWnd** member function **GetDlgItemText** as follows:

```
GetDlgItemText( IDM_DATA, string, 128 );
```

where `string` is the address of a character buffer and `128` is the maximum buffer size. You do not need to reference an edit window object.

If you do need to access resource-based dialog child windows as C++ objects, the Microsoft Foundation classes provide a way. The **GetDlgItem** dialog class member function returns a **CWnd** pointer that corresponds to a dialog child window ID number that is defined by the resource. This **CWnd** pointer refers to an internally allocated window object that is stored in a temporary table. It allows you to use the appropriate window class member functions. The **IsKindOf** and **GetRuntimeClass** member functions of **CObject** can help identify the specific window class of the object.

If, for example, you need the line count from the edit control introduced previously, then use **CWnd::GetDlgItemText** as follows:

```
CEdit* pEdit = (CEdit*)GetDlgItem( IDM_DATA );
int count = pEdit->GetLineCount();
```

Graphics Device Interface (GDI) Classes

The **CWinApp** class and the **CWnd** derivatives are by far the most important Windows-oriented classes in the library. Most are intended for derivation. The GDI Windows classes are included as a convenience for the C++ programmer. Each corresponds almost exactly to a Windows data structure and is generally not used for derivation.

The Device Context Classes

CDC, **CPaintDC**, **CWindowDC**, **CClientDC**, and **CMetaFileDC** are "device context" classes because they are C++ wrappings of the Windows device context. A device context is a Windows data structure associated with a physical device. It is the Windows method of rendering graphics in a hardware-independent manner. In order to draw or print in a window, you must first get access to a display device context object.

The base class, **CDC**, can be used directly to access the entire display or a nondisplay context. A "nondisplay context" is a hardware device, such as a printer or plotter, that has a Windows driver.

A **CWindowDC** context is a "display" context that is "clipped" (by Windows) to include only the area of its associated window. A **CClientDC** context includes only the window's "client" area (exclusive of title bar and scroll bars). A **CPaintDC** context is like a **CClientDC** context except that it is enhanced (by the Microsoft Foundation Class Library) to work in an **OnPaint** member function without the need for the **BeginPaint** and **EndPaint** function calls.

The most frequently used device context is **CPaintDC**. A typical **OnPaint** member function obtains and uses a device context as shown:

```
void CMainWindow::OnPaint()
{ // BeginPaint function call not required
    CPaintDC dc( this ); // The device context for this window
    dc.TextOut( 0, 0, "hello", 5 ); // Top left of the client rectangle
} // EndPaint function call not required
```

The GDI Object Classes

The Windows GDI employs various drawing tools, including pens, brushes, palettes, fonts, bit maps, and regions. Many device context operations, such as drawing and painting, depend on a specific drawing tool being linked to the device context. In the native Windows environment, this operation is known as "selecting a GDI object into a device context." In the Microsoft Foundation classes, a tool type is represented by a class derived from **CGdiObject**.

The base class of the device context classes, **CDC**, has a **SelectObject** member function overloaded for each GDI-object-derived class. This function selects a GDI object to a device context (and returns the previously selected object of that type). Thus you can attach a brush to a paint display context (and use it) as follows:

```
// Create a device context for this window
CPaintDC dc( this );
// Construct a crosshatch filling brush
CBrush brush( HS_DIAGCROSS, 0L );
// Select the brush into the device context
CBrush* pOldBrush = dc.SelectObject( &brush );
// Paint the ellipse with crosshatching
dc.Ellipse( 0, 20, 40, 60 );
// Restore the original brush
dc.SelectObject( pOldBrush );
```

The last statement disconnects `brush` from `dc` at the Windows level. This permits the Windows brush to be deleted by the **CBrush** destructor (when the `brush` object goes out of scope). It is very important to delete Windows GDI objects; otherwise their memory will not be reclaimed, even after the Windows application terminates. Windows GDI objects cannot be deleted as long as they are selected in a valid device context.

Other Windows Classes

There are several other classes in the Microsoft Foundation Class Library that bring C++ syntax to Windows. These classes include **CMenu**, **CPoint**, **CRect**, and **CSize**.

The CMenu Class

A Windows menu is a data structure that associates user actions with **WM_COMMAND** messages. The **CMenu** class wraps this menu structure and provides a constructor for an empty menu. A menu's list of choices can be altered dynamically through member functions such as **AppendMenu**, **InsertMenu**, and **DeleteMenu**. The **LoadMenu** member function loads a menu object with a menu definition from a resource file.

You can attach a resource-based menu, identified by a string or resource ID, directly to a window (through the frame window's **Create** member function) without defining a **CMenu** object. Alternatively, the window **SetMenu** member function associates a **CMenu** object with the window.

The **WM_COMMAND** messages that result from menu activity must be processed by window class member functions that are declared through message-map entries.

The CPoint, CSize, and CRect Classes

CPoint and **CSize** are simple classes that define absolute and relative (x, y) points and provide some useful overloaded operators. The **CRect** class defines rectangular regions specified by the (left, top) and (right, bottom) coordinates. **CPoint** and **CRect** inherit from the Windows **POINT** and **RECT** structures.

Many Microsoft Foundation class functions take **POINT** structures or pointers to **RECT** structures as parameters. Because **CPoint** and **CRect** are derived from these structures, the compiler can accept objects in place of structure instances.

General-Purpose Foundation Classes

Chapter 2

This chapter categorizes and describes the general-purpose classes within the Microsoft Foundation Class Library. These classes can be used alone in an MS-DOS application, or they can be combined with the Microsoft Windows classes described in Chapter 1.

2.1 Class Summary

The following is a list of the Microsoft Foundation Class Library's general-purpose classes categorized by function. **CObject** is the root class in the Microsoft Foundation class hierarchy.

File Classes

CFile	Binary disk files.
CMemFile	In-memory files.
CStdioFile	Buffered stream disk files, usually text mode.

Object Input and Output

CArchive	Persistent storage for objects.
CDumpContext	Destinations for diagnostic dumps.

Exceptions

CException	Base class for exceptions.
CArchiveException	Archive exceptions.
CFileException	File-oriented exceptions.
CMemoryException	Out-of-memory exceptions.

CNotSupportedException	Exceptions resulting from the invocation of an unsupported feature.
CResourceException	Exceptions resulting from a failure to load a Windows resource (Windows only).

Collections

CByteArray	Arrays of bytes.
CDWordArray	Arrays of double words.
CObArray	Arrays of **CObject** pointers.
CPtrArray	Arrays of **void** (generic) pointers.
CStringArray	Arrays of **CString** objects.
CWordArray	Arrays of words.
CObList	Lists of **CObject** pointers.
CPtrList	Lists of **void** (generic) pointers.
CStringList	Lists of **CString** objects.
CMapPtrToWord	Maps that associate **void** pointers to words.
CMapPtrToPtr	Maps that associate **void** pointers to **void** pointers.
CMapStringToOb	Maps that associate **CString** objects to **CObject** pointers.
CMapStringToPtr	Maps that associate **CString** objects to **void** pointers.
CMapStringToString	Maps that associate **CString** objects to **CString** objects.
CMapWordToOb	Maps that associate words to **CObject** pointers.
CMapWordToPtr	Maps that associate words to **void** pointers.

Miscellaneous Support Classes

CString	Character strings.
CTime	Absolute time and date values.
CTimeSpan	Relative time and date values.

Global Functions and Macros

Chapters 3 through 6 of this manual document the elements of the Microsoft Foundation Class Library that are not directly related to individual classes. A complete summary of macros and global functions is provided in Chapter 3, while diagnostic services, including memory diagnostics and object dump functions, are discussed in Chapter 4. Exception processing, which uses **TRY**, **CATCH**, **THROW**, and other macros, is covered in Chapter 5.

2.2 CObject Services

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

- Object persistence
- Object diagnostics
- Run-time class information
- Compatibility with selected collection classes

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.

Even though **CObject** is not a true "abstract" base class, you are advised not to construct objects of this class.

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.

Serialization is not random access, but rather sequential. A group of objects is written to an archive, which is associated with an individual **CFile** object. If the objects to be serialized are contained in a collection, then a single **Serialize** call for the collection object results in the serialization of the whole collection, even if it contains nested objects or heterogeneous object collections. For a good example of collection serialization, see the tutorial in the *Class Libraries User's Guide*.

When you create your own serializable **CObject**-derived class, you must use the **DECLARE_SERIAL** macro in the class declaration, and you must use the **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 and load object data from it.

Like the iostream classes, **CArchive** provides insertion (<<) and extraction (>>) operators.

Object Diagnostics

The Microsoft Foundation classes provide many diagnostic features, but diagnostic object printing and validity checking are specific services of the **CObject** class. For diagnostic features that are not class oriented, see "Memory Diagnostics" later in this chapter, on page 29.

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**, like the **ostream** class (in the iostream library), provides an insertion (<<) operator that accepts not only **CObject** pointers but also standard types 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). With MS-DOS, the output from **afxDump** goes to **stderr**. With Windows, the output goes to the CodeView® debugger if it is present; otherwise it goes to device AUX.

Without any programming on your part, the **Dump** member function of the **CObject** class provides a hexadecimal printout of the contents of your derived object. If you override the base class **Dump** member function in your derived class, you can get a formatted dump of your object's contents. If you have used the **DECLARE_DYNAMIC** or **DECLARE_SERIAL** macros in your derived class declaration and if you have used the **IMPLEMENT_DYNAMIC** or **IMPLEMENT_SERIAL** macros in your derived class implementation, then **Dump** prints your object's class name even if you supply a generic **CObject** pointer.

Object Validity Checking

The **AssertValid** member function of **CObject** always returns **TRUE**. If you override the base class **AssertValid** member function in your derived class, you can perform a specific test of your object's internal consistency.

Run-Time Class Information

The C++ language was designed for speed and efficiency; therefore, binding among functions and data elements is done at *compile and link time*. Even the implementation of virtual functions depends on a data structure (known as the v-table) that is set up during compilation. Other object-oriented languages, such as Smalltalk, are designed for flexibility. Their binding is done at *run time*; objects send and receive standard-format messages that are processed by an interpreted language.

The Microsoft Foundation classes offer the developer some optional features usually associated with a run-time–bound system. If you derive a class from **CObject**, you can use member functions to access, at run time, (1) the class name and (2) the classes above it in the derivation hierarchy. You can also retrieve class information for any **CObject**-derived class declared in your program. This information allows you to 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 (1) to detect incorrect casts and (2) to produce object dumps with class names included.

Run-time class information is, of course, available in the Release environment. If in Windows, for example, you need to process the children of a frame window, you can use the frame's **GetWindow** member function to return a generic **CWnd** pointer for each child window. If you want to know the child's specific class, then you can use the **CObject** member functions **IsKindOf** or **GetRuntimeClass**. During serialization, the runtime class information is stored to the archive along with object data.

Run-time class testing is not meant to be a substitute for using virtual functions. Use the run-time type information only when virtual functions are not appropriate, as in the **GetWindow** example described above.

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

Compatibility with Selected Collection Classes

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 processed. For more about collection classes, see Section 2.4 later in this chapter.

2.3 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.

2.4 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 objects or standard types. C++ makes a collection appear as a single object, so collection member functions can operate on all elements of the collection.

All 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.

If you need a list, array, or map that is not included among the 16 standard collections provided with the Microsoft Foundation classes, then you can use the Templdef template tool that is included in the sample directory. The disk file

MFC\DOC\TN004.TXT contains a guide to the use of this tool. The hierarchy chart of the Microsoft Foundation classes shown at the beginning of Part 2 indicates these three collection templates: "CArray<TYPE>," "CList<TYPE>," and "CMap<KEY, VALUE>."

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 removed during the traversal. Elements can be found by zero-based index or by value, but the find operation requires a sequential scan of the list.

Arrays

The Microsoft Foundation Class Library contains "array" classes for bytes, words, double words, **CString** objects, **CObject** pointers, and **void** pointers. An array is a dynamically sized grouping of elements that are directly accessible through a zero-based integer subscript. If a new element is inserted into an array, then the elements above the insertion point are moved up. If an element above the current array bound is to be set, then the programmer can specify whether the array is to grow automatically. The subscript ([ ]) operator can be used to set or retrieve array elements.

When growing is not required, array collection access is just as fast as standard C array access. The added storage overhead is insignificant.

Maps

A "map" is a dictionary that maps *keys* to *values.* Seven 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. A map can be traversed, but the retrieval sequence is indeterminate. It makes sense, then, to iterate over *all* the elements in a map.

2.5 Miscellaneous 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 in order 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 1900 to 2036, 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. If two **CTime** objects are subtracted, 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.

2.6 Diagnostic Services

Several non-class-related functions and macros that provide diagnostic services are available. Most of these require the Debug version of the Microsoft Foundation Class Library and thus should not be used in released applications. For a detailed description of the functions and macros available, see Chapter 4, "Diagnostic Services."

Memory Diagnostics

Most 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

Most programmers want diagnostic output statements in their programs, particularly during the early stages of development. The **TRACE** statement acts like **printf** except that it has no effect with the Release version of the library. In the Windows environment, debugging output goes to the CodeView debugger if it is present; otherwise it goes to device AUX.

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 prints the source filename and the line number, then it terminates the program . In the Release environment, the 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.

With Windows, **ASSERT** and **VERIFY** display their messages in a pop-up dialog box.

2.7 Exception Handling

The Microsoft Foundation Class Library includes an exception-handling mechanism, similar to 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 end of file.

For exception-processing examples and a more detailed explanation of error categories, see Chapter 12, "Exceptions," in the *Class Libraries User's Guide*. For a detailed description of the functions and macros available, see Chapter 5 of this book, "Exception Processing."

Exception Classes and Macros

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 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**. You can use the **AfxSetTerminate** function to change the effect of **AfxTerminate**.

When to Use Exception Handling

Out-of-memory and disk-full conditions could occur any time during program execution. A **TRY/CATCH** sequence at the top level of your application can provide a warning message to the user, followed by a graceful exit.

Routine file exceptions can occur at a lower level in the application. If your program attempts to open a nonexistent file, local **CATCH** logic can inform the user or take other corrective action. A better alternative, however, might be an explicit test for the file's presence.

If you want your program to keep running after the exception, be very careful to clean up memory by deleting unused objects. Don't forget **CString** objects that have been allocated on the stack.

Macros and Global Functions

Chapter 3

This chapter briefly describes the macros and global functions available to simplify your programming with the Microsoft Foundation Class Library. Most programmers will find that the macros presented here meet most of their needs. Advanced programmers may wish to use some of the global functions provided for special needs.

All macros are listed in alphabetical order, followed by all global functions in alphabetical order. A few items not documented elsewhere are documented following the alphabetical listings.

For easy reference, the following table shows where to find related discussion and examples in other parts of the *Class Libraries Reference* and in the *Class Libraries User's Guide*:

Category	Reference Chapter	User's Guide Chapter
Diagnostics	Chapter 4	Chapter 2
Exceptions	Chapter 5	Chapters 2 and 12
Message Map	Chapter 6	Chapters 3 and 14
Run-Time Class Information	**CObject**	Chapter 8
Serialization	**CObject**	Chapter 10

3.1 Alphabetical Listing of Macros

To find additional discussion and examples for a macro, see the table above, using the category specified in the macro description below. Macros documented in this section, in addition to the locations shown in the table, are indicated by the phrase "Details follow."

AND_CATCH

Designates a block of code for catching the second or subsequent exception from the preceding **TRY** block.

For additional information, see the Exceptions category in the table.

ASSERT

Prints a message and aborts the application if the specified expression evaluates to **FALSE** in the Debug version of the library.

For additional information, see the Diagnostics category in the table.

ASSERT_VALID

Tests the internal validity of an object by calling its **AssertValid** member function, typically overridden from **CObject**.

For additional information, see the Diagnostics category in the table.

BEGIN_MESSAGE_MAP

Sets up the message map for a window class.

For additional information, see the Message Map category in the table.

CATCH

Designates a block of code for catching the first exception from the preceding **TRY** block.

For additional information, see the Exceptions category in the table.

DEBUG_NEW

Helps find memory leaks by providing a filename and line number for all object allocations in Debug mode. Details follow.

For additional information, see the Diagnostics category in the table.

DECLARE_DYNAMIC

Prepares a class so that you can determine its name, the name of its base class, and other information at run time. Details follow.

For additional information, see the Run-Time Information category in the table.

DECLARE_MESSAGE_MAP

Associates a message map with a window class declaration.

For additional information, see the Message Map category in the table.

DECLARE_SERIAL
Prepares a class to serialize its data to and from persistent storage. Details follow.

For additional information, see the Serialization category in the table.

END_CATCH
Ends the last **CATCH** or **AND_CATCH** block in an exception frame.

For additional information, see the Exceptions category in the table.

END_MESSAGE_MAP
Completes a message-map definition for a window class.

For additional information, see the Message Map category in the table.

IMPLEMENT_DYNAMIC
Enables a class so that you can determine its run-time information. Details follow.

For additional information, see the Run-Time Information category in the table.

IMPLEMENT_SERIAL
Enables the ability of a class to serialize its data to and from persistent storage. Details follow.

For additional information, see the Serialization category in the table.

RUNTIME_CLASS
Returns a **CRuntimeClass** object from which you can extract run-time information about a specified class. Details follow.

For additional information, see the Run-Time Information category in the table.

THROW
Throws a specified exception.

For additional information, see the Exceptions category in the table.

THROW_LAST
Invokes the exception handler in the next outer frame.

For additional information, see the Exceptions category in the table.

TRACE
Provides a **printf**-like capability in the Debug version of the library.

For additional information, see the Diagnostics category in the table.

TRY
Designates a block of code for exception processing.

For additional information, see the Exceptions category in the table.

VERIFY
Similar to **ASSERT** but evaluates the expression in the Release version of the library as well as in the Debug version.

For additional information, see the Diagnostics category in the table.

3.2 Alphabetical Listing of Global Functions

To find additional discussion and examples for a global function, see the preceding table, using the category specified in the function description below. Functions that are documented in this section are indicated by the phrase "Details follow."

AfxAbort
The default function called by **AfxTerminate**.

For additional information, see the Exceptions category in the table.

AfxCheckMemory
Checks all currently allocated memory for corrupted guard bytes.

For additional information, see the Diagnostics category in the table.

AfxDoForAllClasses
Performs a specified function on all classes derived from **CObject** that support run-time type checking and are used by the running program.

For additional information, see the Diagnostics category in the table.

AfxDoForAllObjects
Performs a specified function on all objects derived from **CObject** that support run-time type checking and are used by the running program.

For additional information, see the Diagnostics category in the table.

AfxEnableMemoryTracking
Turns memory tracking on and off.

For additional information, see the Diagnostics category in the table.

AfxGetApp
Returns a pointer to the application's one **CWinApp** object. Details follow.

AfxGetAppName
Returns a string containing the application's name. Details follow.

AfxGetInstanceHandle
Returns a **HANDLE** to the current instance of the application. Details follow.

AfxGetResourceHandle
Returns a **HANDLE** to the current instance of the application. Use this handle to access the application's resources directly. Details follow.

AfxIsMemoryBlock
Verifies that a memory block has been properly allocated.

For additional information, see the Diagnostics category in the table.

AfxIsValidAddress
Verifies that a memory block is within the program's bounds.

For additional information, see the Diagnostics category in the table.

AfxRegisterWndClass
Registers a Windows window class to supplement those registered automatically by the library. Details follow.

For additional information, see the *Class Libraries User's Guide*, Chapter 14, "Window Management."

AfxSetAllocHook
Enables the calling of a function on each memory allocation.

For additional information, see the Diagnostics category in the table.

AfxSetAllocStop
Enables the calling of a function on the nth memory allocation.

For additional information, see the Diagnostics catgeory in the table.

AfxSetTerminate
Sets the final destination of calls to **AfxTerminate**.

For additional information, see the Exceptions category in the table.

AfxTerminate
Called internally if there is no applicable **TRY/CATCH** frame in effect.

For additional information, see the Exceptions category in the table.

AfxThrowArchiveException
Throws an archive exception.

For additional information, see the Exceptions category in the table.

AfxThrowFileException
Throws a file exception.

For additional information, see the Exceptions category in the table.

AfxThrowMemoryException
Throws a memory exception.

For additional information, see the Exceptions category in the table.

AfxThrowNotSupportedException
Throws a not-supported exception.

For additional information, see the Exceptions category in the table.

AfxThrowResourceException
Throws a Windows resource-not-found exception.

For additional information, see the Exceptions category in the table.

3.3 Macros and Global Functions Not Documented Elsewhere

AfxGetApp

Syntax **CWinApp* AfxGetApp( );**

Remarks Returns a pointer to the one and only **CWinApp** object for the Windows application. This pointer is useful for getting access to the main message dispatch code or the topmost window.

Return Value A pointer to a **CWinApp** object.

AfxGetAppName

Syntax **const char* AfxGetAppName( );**

Remarks Returns a null-terminated string containing the Windows application's name. This string is useful for diagnostic messages or as a root for temporary string names.

Return Value A null-terminated string containing the application's name.

AfxGetInstanceHandle

Syntax **HANDLE AfxGetInstanceHandle( );**

Remarks Returns a **HANDLE** to the current instance of the Windows application.

Return Value A **HANDLE** to the current instance of the application.

AfxGetResourceHandle

Syntax

HANDLE AfxGetResourceHandle( );

Remarks

Returns a **HANDLE** to the current instance of the Windows application. Use this handle to access the application's resources directly, for example in calls to the Windows function **FindResource**.

Note Override and reimplement this function if you wish to load your resources from a DLL.

Return Value

A **HANDLE** to the current instance of the application.

AfxRegisterWndClass

Syntax

const char* AfxRegisterWndClass(UINT *nClasstyle*, **HCURSOR** *hCursor* **= 0, HBRUSH** *hbrBackground* **= 0, HICON** *hIcon* **= 0);**

Parameters

nClasstyle
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.

hCursor
A handle to the cursor resource to be installed in each window created from the window class.

hbrBackground
A handle to the brush resource to be installed in each window created from the window class.

hIcon
A handle to the icon resource to be installed in each window created from the window class.

Remarks

Although the Microsoft Foundation Class Library automatically registers several standard window classes for you, you can call this function to register your own window classes. You may also use the function to change the application's icon, although a simpler way is discussed in the *Class Libraries User's Guide*. For additional information, see Chapter 14, "Window Management." See that chapter as well for more information about using **AfxRegisterWndClass**, and see Technical Note 1, in the file TN001.TXT on the distribution disks.

Return Value

A null-terminated string containing the class name. You can pass this class name to the **CWnd::Create** member function to create a window. The name is generated by the Microsoft Foundation Class Library.

Note The return value is stored in a static buffer. To save this string, assign it to a **CString** variable.

DEBUG_NEW Macro

Syntax

#define new DEBUG_NEW

Remarks

Use to assist 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 **DumpAllObjectsince** member function of class **CMemoryState**, each object allocated with **DEBUG_NEW** is shown with the filename and line number where it was allocated.

To use **DEBUG_NEW**, insert the **define** directive shown in the syntax line above into your source files. Then wherever you use **new**, the preprocessor will insert **DEBUG_NEW**, and the 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.

Note In Release mode, **DEBUG_NEW** is defined to be the standard operator **new**, so you can leave **DEBUG_NEW** in your code.

DECLARE_DYNAMIC Macro

Syntax

DECLARE_DYNAMIC(*className* **);**

Parameters

className
The name of the class that you want to be compliant with the ability of class **CObject** to supply dynamic run-time class information.

Remarks

Any class derived from class **CObject** can supply run-time information about itself and its base class, provided you invoke the **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC** macros. This means you can determine the exact class of an object at run time and also determine the base class from which it was derived.

Put the **DECLARE_DYNAMIC** macro in your class declaration. Put the **IMPLEMENT_DYNAMIC** macro in your .CPP file. These macros add code to your class to enable dynamic run-time information.

You can access the dynamic information about a class with the **IsKindOf** member function of class **CObject** and with the **RUNTIME_CLASS** macro. This run-time information is available and valid only for classes that have a single base class. For more information and examples, see the *Class Libraries User's Guide*, Chapter 8, "The CObject Class."

See Also

IMPLEMENT_DYNAMIC, RUNTIME_CLASS, CObject

DECLARE_SERIAL Macro

Syntax

DECLARE_SERIAL(*className* **)**

Parameters

className
The name of the class that is to have serialization capability.

Remarks

Classes that are derived from **CObject** can take advantage of the ability of **CObject** ability to write its members to a persistent storage medium, such as a disk file, and to read its persistent data back in.

DECLARE_SERIAL includes dynamic type information as well, so you don't need **DECLARE_DYNAMIC** if you use **DECLARE_SERIAL**.

To take advantage of this ability, a derived class must use the **DECLARE_SERIAL** macro in its class declaration and the corresponding **IMPLEMENT_SERIAL** macro in its .CPP file. The class must also override the **Serialize** member function of class **CObject**.

Put the **DECLARE_SERIAL** macro at the beginning of your derived class declaration. For more discussion and examples, see the *Class Libraries User's Guide*, Chapter 2, "Creating a Data Model with the Microsoft Foundation Classes," and Chapter 10, "Files and Serialization."

See Also **IMPLEMENT_SERIAL**

IMPLEMENT_DYNAMIC Macro

Syntax **IMPLEMENT_DYNAMIC(** *className*, *baseClassName* **)**

Parameters

className
The name of the class that you want to be compliant with the ability of class **CObject** to supply dynamic run-time information.

baseClassName
The name of the base class of your compliant class.

Remarks Use in your .CPP file in conjunction with the **DECLARE_DYNAMIC** macro in your .H file to allow your class to supply dynamic run-time information. This allows you to query an object with the **IsKindOf** member function in class **CObject** to determine its class and base class names. For discussion and examples, see the *Class Libraries User's Guide*, Chapter 8, "The CObject Class."

See Also **DECLARE_DYNAMIC, CObject**

IMPLEMENT_SERIAL Macro

Syntax **IMPLEMENT_SERIAL(** *className*, *baseClassName*, *schemaNumber* **)**

Parameters

className
The name of the class that is to have the ability to serialize its members to persistent storage.

baseClassName
The name of the base class of the serializable class.

schemaNumber
The version number for objects of this class. If you modify a class, you can assign it a higher schema. Then, during serialization from storage to memory, if the schema number of the object on disk does not match that of the class in memory, an exception is thrown. This prevents you from reading an incorrect version of an object. The schema number is an integer greater than or equal to 0.

Remarks

Use in your .CPP file to correspond to the **DECLARE_SERIAL** macro in your .H file. This macro adds the necessary code to permit a class to serialize its members. You must also override the **Serialize** member function of class **CObject**.

See Also

DECLARE_SERIAL, **CObject::Serialize**, **CObject**

RUNTIME_CLASS Macro

Syntax

RUNTIME_CLASS(*className* **)**

Parameters

className
The name of the class that you want run-time class information.

Remarks

Use to extract the run-time class information for a specified class derived from **CObject**. The macro returns an object of class **CRuntimeClass**. A **CRuntimeClass** structure has member variables containing the class name, object size, schema number, base class, and other information, which you can access directly. **CRuntimeClass** is defined in the file AFX.H. You can also use the **IsKindOf** member function of class **CObject** to query whether an object belongs to a specified class. For more information and examples, see the *Class Libraries User's Guide*, Chapter 8, "The CObject Class."

See Also

CObject::IsKindOf, **CRuntimeClass**, **CObject**

Example

```
CRuntimeClass* pCls;
pCls = RUNTIME_CLASS( CObject );
...
```

Diagnostic Services

Chapter 4

This chapter describes a group of macros and global functions that provide diagnostic services. All these functions, except as noted, require the Debug version of the Microsoft Foundation Class Library.

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 the problem.

If you include the line

```
#define new DEBUG_NEW
```

in your implementation file, then all calls to **new** will store the filename and line number where the memory allocation took place. The **DumpAllObjectsSince** function of the **CMemoryState** class will display this extra information, thus greatly simplifying the identification of memory leaks.

Since many of these diagnostic functions are designed for tracking memory errors, you should refer to the "Memory Management" section in Chapter 7 of the *Class Libraries User's Guide* for a discussion of memory allocation for both MS-DOS and Windows while using the Microsoft Foundation Class Library. Refer also to the class **CDumpContext** for additional information on diagnostic output. The cookbook section of the *Class Libraries User's Guide*, "Detecting Memory Leaks" (page 290), illustrates the use of several key memory-diagnostic functions.

For a general discussion of diagnostic facilities, see Chapter 11, "Diagnostics," in the *Class Libraries User's Guide*.

To use these macros and global functions, add the following directives to the top of your program:

#define _DEBUG

#include <afx.h>

4.1 General Diagnostic Macros

The following list describes general diagnostic macros:

ASSERT
Prints a message if the specified expression evaluates to **FALSE** in the Debug version of the library, and then aborts the program.

ASSERT_VALID
Tests the internal validity of an object by calling its **AssertValid** member function, typically overridden from **CObject**.

TRACE
Provides **printf**-like capability in the Debug version of the library.

VERIFY
Similar to **ASSERT** but evaluates the expression in the Release version of the library as well as in the Debug version.

4.2 General Diagnostic Functions

The following list describes general diagnostic functions:

afxMemDF
Global variable that controls the behavior of the debugging memory allocator.

AfxCheckMemory
Checks all currently allocated memory for corrupted guard bytes.

AfxEnableMemoryTracking
Turns memory tracking on and off.

AfxIsMemoryBlock
Verifies that a memory block has been properly allocated.

AfxIsValidAddress
Verifies that any memory block is within the program's bounds.

AfxSetAllocHook
Enables the calling of a function on each memory allocation.

AfxSetAllocStop
Enables the calling of a function on the nth memory allocation.

Checkpoint
A **CMemoryState** member function that checkpoints a memory state.

CMemoryState
Constructor for a class-like structure that controls memory checkpointing.

Difference
A **CMemoryState** member function that computes the difference between two checkpointed memory states.

DumpAllObjectsSince
A **CMemoryState** member function that dumps all currently allocated objects since the last checkpoint.

DumpStatistics
A **CMemoryState** member function that prints memory allocation statistics.

4.3 Object Diagnostic Functions

The following list describes object diagnostic functions:

AfxDoForAllClasses
Performs a specified function on all **CObject**-derived classes that support run-time type checking by using the **DECLARE_DYNAMIC** or **DECLARE_SERIAL** macros.

AfxDoForAllObjects
Performs a specified function on all **CObject**-derived objects that support run-time type checking.

4.4 Global Variables

afxMemDF

Syntax

int afxMemDF;

Remarks

An integer variable, easily accessible from a debugger, that tunes the 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.
checkAlwaysMemDF	Calls **AfxCheckMemory** every time memory is allocated or freed. This will significantly slow memory allocations and deallocations.

Example

```
afxMemDF = delayFreeMemDF | checkAlwaysMemDF;
```

4.5 Functions and Macros

AfxCheckMemory

Syntax

BOOL PASCAL AfxCheckMemory();

Remarks

Iterates through all memory blocks currently allocated on the heap. These blocks include those allocated by **new**, but not those allocated by direct calls to underlying memory allocators such as **malloc** or **::GlobalAlloc**. If any block is found to have corrupt guard bytes, a message is printed on **stderr**.

If the block contains an object of a class derived from **CObject**, then the function reports an `Object`, otherwise it reports a `Non-Object`. It always reports an address that corresponds to the address printed by **DumpAllObjectsSince**.

Additionally, the function validates the free memory pool, printing error messages as required.

If the function detects no memory corruption, it prints nothing.

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 **new** redefinition statement after the last **IMPLEMENT_SERIAL** macro invocation.

Return Value

TRUE if no memory errors; otherwise **FALSE**.

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();

/*  T Y P I C A L   R E S U L T S
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 4 bytes long
*/
```

AfxDoForAllClasses

Syntax

void PASCAL AfxDoForAllClasses(void (**pfn*)**(const CRuntimeClass*** *pClass*, **void*** *pContext*)**, void*** *pContext* **);**

Parameters

pfn
A pointer to an iteration function to execute for each class. The function arguments are a pointer to a **CRuntimeClass** object and an optional void pointer to extra data that the caller supplies to the function.

pClass
A pointer to a **CRuntimeClass** object. **AfxDoForAllClasses** uses this parameter to pass each eligible class in turn to the iteration function.

pContext
A pointer to optional data that the caller can supply to the iteration function.

Remarks

Executes the specified iteration function for all **CObject**-derived classes in the application's memory space that support run-time type checking using the **DECLARE_DYNAMIC** or **DECLARE_SERIAL** macros. The pointer 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 library.

AfxDoForAllObjects

Syntax

void PASCAL AfxDoForAllObjects(void (**pfn***)(CObject*** *pObject*,
void* *pContext*)**, void*** *pContext* **);**

Parameters

pfn
A pointer to an iteration function to execute for each object. The function arguments are a pointer to a **CObject** and an optional **void** pointer to extra data that the caller supplies to the function.

pObject
A pointer to an object of class **CObject** or a class derived from it. **AfxDoForAllObjects** uses this parameter to pass each eligible object in turn to the iteration function.

pContext
A pointer to optional data that the caller can supply to the iteration function.

Remarks

Executes the specified iteration function for all objects derived from **CObject** in the application's memory space. The objects must have been allocated with **new**; stack objects are not enumerated. The pointer 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 library.

AfxEnableMemoryTracking

Syntax

BOOL PASCAL AfxEnableMemoryTracking(BOOL *bTrack* **);**

Parameters

bTrack
TRUE turns on memory tracking; **FALSE** turns it off.

Remarks

Diagnostic memory tracking is normally enabled in the Debug version of the Microsoft Foundation classes. Use this function to disable tracking on sections of your code that you know are allocating blocks correctly.

Return Value

The previous setting of the tracking-enable flag.

AfxIsMemoryBlock

Syntax

BOOL PASCAL AfxIsMemoryBlock(const void* *p*, **UINT** *nBytes*,
LONG* *plRequestNumber* = **NULL);**

Parameters

p
A **void** pointer to the block of memory to be tested.

nBytes
The length of the memory block in bytes.

plRequestNumber
A pointer 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 **TRUE**.

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. The allocation sequence number that is returned in *plRequestNumber* if the function returns **TRUE** is the order in which the block was allocated relative to all other **new** allocations.

Return Value

TRUE if the memory block is currently allocated and the length is correct; otherwise **FALSE**.

Example

```
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject
if( AfxIsMemoryBlock( pcage, sizeof( CAge ) ) != TRUE )
    exit( 1 );   // Invalid memory
```

See Also

AfxIsValidAddress

AfxIsValidAddress

Syntax

BOOL FAR PASCAL AfxIsValidAddress(const void FAR* *lp*, **UINT** *nBytes*, **BOOL** *bReadWrite* = **TRUE);**

Parameters

lp
Points to the block of memory to be tested.

nBytes
Contains the length of the memory block in bytes.

bReadWrite
Specifies whether the memory is both for reading and writing.

Remarks

Tests any memory block to ensure that it is contained entirely within the program's memory space. The address is not restricted to blocks allocated by **new**.

Note With MS-DOS real mode, only addresses with null selectors are invalid; all others are valid. A huge pointer cast to a **FAR** pointer cannot be used as a parameter to **AfxIsValidAddress**.

Return Value

TRUE if the specified memory block is contained entirely within the program's memory space; otherwise **FALSE**.

Example

```
char* pbuf = (char*) malloc( 10 );
if( AfxIsValidAddress( pbuf, 10, TRUE ) != TRUE )
    exit( 1 );   // Invalid memory
```

See Also

AfxIsMemoryBlock

AfxSetAllocHook

Syntax

AFX_ALLOC_HOOK AfxSetAllocHook(AFX_ALLOC_HOOK *pfnAllocHook* **);**

Parameters

pfnAllocHook
The name of the function to call. The function must return a **BOOL** value and accept **size_t**, **BOOL**, and **long** arguments.

Remarks

Sets a hook that enables calling of the specified function each time memory 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 control whether to permit the allocation.

Allocation hook functions are prototyped as:

BOOL AllocHook(**size_t** *nSize*, **BOOL** *bObject*, **LONG** *lRequestNumber*)**;**

Parameter	Description
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

TRUE if you want to permit the allocation; otherwise **FALSE**.

AfxSetAllocStop

Syntax

void PASCAL AfxSetAllocStop(LONG *lRequestNumber* **);**

Parameters

lRequestNumber
The sequence number of the memory allocation on which the program will halt.

Remarks

Each memory allocation is assigned a sequential serial number. This function forces the program to halt (using the INT 3 interrupt) on the specified memory allocation sequence number. This is useful if you are running the program from within a debugger. You can obtain the allocation sequence number to pass in *lRequestNumber* by calling **AfxIsMemoryBlock**.

ASSERT Macro

Syntax

ASSERT(*booleanExpression* **);**

Parameters

booleanExpression
An expression (including pointer values) that evaluates to **TRUE** or **FALSE**.

Remarks

The **ASSERT** macro evaluates its argument. If the result is **FALSE**, the macro prints a diagnostic message and aborts the program. If the condition is **TRUE**, 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 environment, **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**.

Example

```
CAge* pcage = new CAge( 21 ); // CAge is derived from CObject
ASSERT( pcage->IsKindOf( RUNTIME_CLASS( CAge ) ) );
// Terminates program only if pcage is NOT a CAge*
```

See Also

VERIFY

ASSERT_VALID Macro

Syntax

ASSERT_VALID(*object* **);**

Parameters

object
An object of a class derived from **CObject** and with 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. By default, the **AssertValid** member function of class **CObject** is called, but typically you override **AssertValid** in classes that you derive from **CObject** so the overriding version will be called. In your **AssertValid** override, you can test the object's internal validity. For example, if the object represents a linked list, you could verify that the head and tail pointers are **NULL** if the list is empty and not **NULL** if the list is not empty.

For more information and examples, see the *Class Libraries User's Guide*, Chapters 4 and 11, "Phone Book: A Simple Windows Database," and "Diagnostics."

See Also **ASSERT**, **VERIFY**, **CObject**, **CObject::AssertValid**

CMemoryState::Checkpoint

Syntax **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

Syntax **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 cmOld, cmNew, cmDif;
    cmOld.Checkpoint();
    CAge* page1 = new CAge( 21 );
    CAge* page2 = new CAge( 22 );
    cmOld.DumpAllObjectsSince();
    cmNew.Checkpoint();
    cmDif.Difference( cmOld, cmNew );
    cmDif.DumpStatistics();
```

CMemoryState::Difference

Syntax

BOOL Difference(const CMemoryState& *oldState*,
  const CMemoryState& *newState* **);**

Parameters

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 checkpointed **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

Syntax

void DumpAllObjectsSince() const;

Remarks

Calls the **Dump** function for all objects of derived **CObject** classes that were allocated (and are still allocated) since the last **Checkpoint** call for this **CMemoryState** object.

Use **DumpAllObjectsSince** in conjunction with **AfxCheckMemory** to match reported corrupted memory with the contents of the objects contained there.

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

Syntax

void DumpStatistics() const;

Remarks

Prints, on **afxDump**, a concise memory statistics report from a **CMemoryState** object that is filled by the **Difference** member function. The report shows the following:

- **CObject** blocks still allocated on the heap.
- Non-**CObject** blocks still allocated on the heap.
- The maximum memory used by the program at any one time.
- The total memory currently used by the program.

For a detailed description of the report, see the *Class Libraries User's Guide* section "Detecting Memory Leaks," on page 290.

Example

See the example for the **CMemoryState** constructor.

TRACE Macro

Syntax

TRACE(*exp* **);**

Parameters

exp
A variable number of arguments used exactly in the same way as the run-time function **printf** uses them.

Remarks

In the Debug environment, the **TRACE** macro output goes to **afxDump**. In the Release environment, it does nothing. This is a convenient way of generating debugging output that will appear only in the Debug version of your program.

Example

```
int i = 1;
char sz[] = "one";
TRACE( "Integer = %d, String = %s\\n", i, sz );
// Output: 'Integer = 1, String = one'
```

VERIFY Macro

Syntax

VERIFY(*booleanExpression* **);**

Parameters

exp
An expression (including pointer values) that evaluates to **TRUE** or **FALSE**.

Remarks

In the Debug version of the Microsoft Foundation Class Library, the **VERIFY** macro evaluates its argument. If the result is **FALSE**, the macro prints a diagnostic message and halts the program. If the condition is **TRUE**, 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.

Example

```
CFile f;
VERIFY( f.Open( "file.dat", CFile::modeCreate | CFile::modeWrite ) );
// Terminates program if Open fails; always executes Open
```

See Also

ASSERT

Chapter 5

Exception Processing

This chapter describes macros and global functions that relate to exception processing.

For examples and more details, see the section "Exception Handling" (on page 61 in Chapter 2) and the section "Catching Exceptions" (in Chapter 12, "Exceptions"), both in the *Class Libraries User's Guide.* You may also wish to refer to class **CException**, later in this book.

Note The **AfxThrow** functions are equivalent to the **THROW** macro with the appropriate exception class as an argument.

To use these macros and global functions, add the following directive at the top of your program:

#include <afx.h>

5.1 Exception Macros

TRY
Designates a block of code for exception processing.

CATCH
Designates a block for catching an exception from the preceding **TRY** block.

AND_CATCH
Designates a block for catching additional exception types from the preceding **TRY** block.

END_CATCH
Ends the last **CATCH** or **AND_CATCH** block.

THROW
Throws a specified exception.

THROW_LAST
Invokes the exception handler in the next outer frame.

5.2 Exception Throwing Functions

AfxThrowArchiveException
Throws an archive exception.

AfxThrowFileException
Throws a file exception.

AfxThrowMemoryException
Throws a memory exception.

AfxThrowNotSupportedException
Throws a not-supported exception.

AfxThrowResourceException
Throws a Windows resource-not-found exception.

5.3 Termination Functions

AfxTerminate
Called internally if there is no applicable **TRY/CATCH** in effect.

AfxSetTerminate
Sets the final destination of calls to **AfxTerminate**.

AfxAbort
The default function called by **AfxTerminate**.

5.4 Functions and Macros

AfxAbort

Syntax

void CDECL AfxAbort();

Remarks

This is the default termination function supplied by the Microsoft Foundation classes.

See Also

AfxSetTerminate, **AfxTerminate**

AfxSetTerminate

Syntax

AFX_TERM_PROC AfxSetTerminate(AFX_TERM_PROC *proc* **);**

Parameters

proc
The name of a termination function that will be called by **AfxTerminate**. Termination functions must take no arguments and return nothing.

Remarks

Links **AfxTerminate** to the specified function. The default termination function is **AfxAbort**. **AfxTerminate** is called internally by Microsoft Foundation member functions when there is a fatal error, such as an uncaught exception.

Example

```
void MyTerminateProc()  // Called instead of AfxAbort
{
    printf( "Out of memory!\\n" );
    exit( 1 );
}

void main()
{
    AfxSetTerminate( MyTerminateProc );

    while  ( 1 )
    {
        // new calls AfxTerminate if unsuccessful
        BYTE * p = new BYTE[1024];        // Consume memory
        printf( "consumed memory at $%x\\n", p );
    }
}
```

See Also **AfxAbort**, **AfxTerminate**

AfxTerminate

Syntax **void CDECL AfxTerminate();**

Remarks Called internally by Microsoft Foundation member functions when there is a fatal error, such as an uncaught exception. Normally, **AfxTerminate** calls **AfxAbort**, but you can use **AfxSetTerminate** to enable the calling of a different function.

You can call **AfxTerminate** any time you encounter an error from which you cannot recover.

See Also **AfxAbort**, **AfxSetTerminate**

AfxThrowArchiveException

Syntax

void PASCAL AfxThrowArchiveException(int *cause* **);**

Parameters

cause
An integer that indicates the reason for the exception. For a list of the possible values, see **CArchiveException::m_cause**

Remarks

Throws **CArchiveException**. This is a helper function used in the implementation of the Microsoft Foundation classes.

AfxThrowFileException

Syntax

void PASCAL AfxThrowFileException(int *cause*, **LONG** *lOsError* = **–1);**

Parameters

cause
An integer that indicates the reason for the exception. For a list of the possible values, see **CFileException::m_cause**.

lOsError
An operating-system-specific reason for the exception, if available. The *lOsError* parameter provides more information than *cause*.

Remarks

Throws a **CFileException**. You are responsible for determining the cause based on the operating system error code. This is a helper function used in the implementation of the Microsoft Foundation classes.

Call this function when you implement your own low-level file operations in a derived file class.

See Also

CFileException::ThrowOsError

AfxThrowMemoryException

Syntax

void PASCAL AfxThrowMemoryException();

Remarks

Throws a **CMemoryException**. This is a helper function used in the implementation of the Microsoft Foundation classes.

Call this function if calls to underlying system memory allocators (such as **malloc** and **::GlobalAlloc**) fail. You do not need to call it for **new** because **new** makes the call internally.

AfxThrowNotSupportedException

Syntax

void PASCAL AfxThrowNotSupportedException();

Remarks

Throws a **CNotSupportedException**. This is a helper function used in the implementation of the Microsoft Foundation classes.

AfxThrowResourceException

Syntax

void PASCAL AfxThrowResourceException();

Remarks

Throws a **CResourceException**. It is normally called when a Windows resource cannot be loaded. This is a helper function used in the implementation of the Microsoft Foundation classes.

Note This function requires the statement **#include <afxwin.h>**.

AND_CATCH Macro

Syntax

AND_CATCH(*exception_class*, *exception_object_pointer_name* **)**

Parameters

exception_class
The specific exception type to test for. For a list of standard exception classes, see **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.

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. Invocation of the **THROW_LAST** macro within the **AND_CATCH** block shifts processing to the next outer exception frame.

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.

AND_CATCH marks the end of the preceding **CATCH** or **AND_CATCH** block.

See Also

TRY, **CATCH**, **THROW**, **END_CATCH**, **THROW_LAST**

CATCH Macro

Syntax

CATCH(*exception_class*, *exception_object_pointer_name* **)**

Parameters

exception_class
The specific exception type to test for. For a list of standard exception classes, see **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 **CATCH** block.

Remarks

Defines a block of code for catching the first exception type thrown in a 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. Invocation of the **THROW_LAST** macro shifts processing to the next outer exception frame.

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.

If *exception_class* is **CException**, then all exception types will be caught. You can use **CObject::IsKindOf** 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.

Note The exception object is created by the macro. You do not need to declare it yourself.

See Also

TRY, AND_CATCH, END_CATCH, THROW, THROW_LAST

END_CATCH Macro

Syntax

END_CATCH

Remarks

Marks the end of the last **CATCH** or **AND_CATCH** block.

See Also

TRY, CATCH, THROW, AND_CATCH, THROW_LAST

THROW Macro

Syntax

THROW(*exception_object_pointer* **);**

Parameters

exception_object_pointer
Points to an exception object derived from **CException**.

Remarks

Throws the specified exception. It 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.

See Also **TRY**, **CATCH**, **THROW_LAST**, **AND_CATCH**, **END_CATCH**

THROW_LAST Macro

Syntax **THROW_LAST();**

Remarks Rethrows the exception back to the next outer **CATCH** block. If your code does not contain an outer block, then the Microsoft Foundation Class Library prints an appropriate error message and terminates the program, just as it would if you provided no exception-processing logic.

This 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.

See Also **TRY**, **CATCH**, **THROW**, **AND_CATCH**, **END_CATCH**

TRY Macro

Syntax **TRY**

Remarks 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.

Note The **TRY** 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.

See Also **THROW**, **CATCH**, **AND_CATCH**, **END_CATCH**

Message Map Cross-Reference

Chapter 6

This chapter lists all possible **CWnd** message map entries along with the corresponding member function prototypes.

6.1 How to Use the Cross-Reference

In entries where the term *memberFxn* is used, you must write your own member function for a derived **CWnd** class. You can give these functions any name you like. Other functions, such as **OnActivate**, are member functions of the **CWnd** base class that, if called, pass the message to the **DefWindowProc** Windows function. If you wish to process Windows notification messages, you must override the corresponding **CWnd** function in your derived class. Your function should call the overridden function in your base class so that the base class(es), and Windows, can operate on the message.

In all cases you must put the function prototype in the **CWnd**-derived class header, and you must code the message map entry as shown. See Chapter 14 of the *Class Libraries User's Guide* cookbook for message map examples.

The term *id* is any user-defined menu item ID (**WM_COMMAND** messages) or control ID (child window notification messages). The terms *message* and *wNotifyCode* are the Windows message IDs as defined in WINDOWS.H. The term *nMessageVariable* is the name of a variable that contains the return value from the **RegisterWindowMessage** Windows function. It must be declared **NEAR**.

6.2 Message Map Function Categories

The rest of this section is divided into the following categories. Each category represents a group of Windows messages for which the Microsoft Foundation Class Library provides handler functions that you can override in your derived window classes.

- Handlers for **WM_COMMAND** messages generated by user menu selections
- Handlers for **WM_COMMAND** messages generated by keys
- Handlers for notification messages from child windows
- Handlers for **WM_** messages, such as **WM_PAINT**

6.3 Handlers for WM_COMMAND Messages

Map Entry	Function Prototype
ON_COMMAND(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );

6.4 Handlers for Child Window Notification Messages

Generic Control Notification Codes

Map Entry	Function Prototype
ON_CONTROL(*wNotifyCode*, *id*, *memberFxn* **)**	**afx_msg void** memberFxn( );

User Button Notification Codes

Map Entry	Function Prototype
ON_BN_CLICKED(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );
ON_BN_DISABLE(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );
ON_BN_DOUBLECLICKED(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );
ON_BN_HILITE(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );
ON_BN_PAINT(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );
ON_BN_UNHILITE(*id*, *memberFxn* **)**	**afx_msg void** memberFxn( );

Combo Box Notification Codes

Map Entry	Function Prototype
ON_CBN_DBLCLK(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_DROPDOWN(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_EDITCHANGE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_EDITUPDATE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_ERRSPACE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_KILLFOCUS(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_SELCHANGE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_CBN_SETFOCUS(*id, memberFxn* **)**	**afx_msg void** memberFxn( );

Edit Control Notification Codes

Map Entry	Function Prototype
ON_EN_CHANGE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_ERRSPACE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_HSCROLL(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_KILLFOCUS(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_MAXTEXT(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_SETFOCUS(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_UPDATE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_EN_VSCROLL(*id, memberFxn* **)**	**afx_msg void** memberFxn( );

List Box Notification Codes

Map Entry	Function Prototype
ON_LBN_DBLCLK(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_LBN_ERRSPACE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_LBN_KILLFOCUS(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_LBN_SELCHANGE(*id, memberFxn* **)**	**afx_msg void** memberFxn( );
ON_LBN_SETFOCUS(*id, memberFxn* **)**	**afx_msg void** memberFxn( );

6.5 Handlers for Windows Notification Messages

Map Entry	Function Prototype
ON_WM_ACTIVATE( )	afx_msg void OnActivate(UINT, CWnd*, BOOL);
ON_WM_ACTIVATEAPP( )	afx_msg void OnActivateApp(BOOL, HANDLE);
ON_WM_ASKCBFORMATNAME( )	afx_msg void OnAskCbFormatName(UINT, LPSTR);
ON_WM_CANCELMODE( )	afx_msg void OnCancelMode( );
ON_WM_CHANGECBCHAIN( )	afx_msg void OnChangeCbChain(HWND, HWND);
ON_WM_CHAR( )	afx_msg void OnChar(UINT, UINT, UINT);
ON_WM_CHARTOITEM( )	afx_msg int OnCharToItem(UINT, CWnd*, UINT);
ON_WM_CHILDACTIVATE( )	afx_msg void OnChildActivate( );
ON_WM_CLOSE( )	afx_msg void OnClose( );
ON_WM_COMPACTING( )	afx_msg void OnCompacting(UINT);
ON_WM_COMPAREITEM( )	afx_msg int OnCompareItem(LPCOMPAREITEMSTRUCT);
ON_WM_CREATE( )	afx_msg int OnCreate(LPCREATESTRUCT);
ON_WM_CTLCOLOR( )	afx_msg HBRUSH OnCtlColor(CDC*, CWnd*, UINT);
ON_WM_DEADCHAR( )	afx_msg void OnDeadChar(UINT, UINT, UINT);
ON_WM_DELETEITEM( )	afx_msg void OnDeleteItem (LPDELETEITEMSTRUCT);
ON_WM_DESTROY( )	afx_msg void OnDestroy( );
ON_WM_DESTROYCLIPBOARD( )	afx_msg void OnDestroyClipboard( );
ON_WM_DEVMODECHANGE( )	afx_msg void OnDevModeChange(LPSTR);
ON_WM_DRAWCLIPBOARD( )	afx_msg void OnDrawClipboard( );
ON_WM_DRAWITEM( )	afx_msg void OnDrawItem(LPDRAWITEMSTRUCT);
ON_WM_ENABLE( )	afx_msg void OnEnable(BOOL);
ON_WM_ENDSESSION( )	afx_msg void OnEndSession(BOOL);
ON_WM_ENTERIDLE( )	afx_msg void OnEnterIdle(UINT, CWnd*);
ON_WM_ERASEBKGND( )	afx_msg BOOL OnEraseBkgnd(CDC*);

Map Entry	Function Prototype
ON_WM_FONTCHANGE( )	afx_msg void OnFontChange( );
ON_WM_GETDLGCODE( )	afx_msg UINT OnGetDlgCode( );
ON_WM_GETMINMAXINFO( )	afx_msg void OnGetMinMaxInfo(LPPOINT);
ON_WM_HSCROLL( )	afx_msg void OnHScroll(UINT, UINT, CWnd*);
ON_WM_HSCROLLCLIPBOARD( )	afx_msg void OnHScrollClipboard(CWnd*, UINT, UINT);
ON_WM_ICONERASEBKGND( )	afx_msg void OnIconEraseBkgnd(CDC*);
ON_WM_INITMENU( )	afx_msg void OnInitMenu(CMenu*);
ON_WM_INITMENUPOPUP( )	afx_msg void OnInitMenuPopup(CMenu*, UINT, BOOL);
ON_WM_KEYDOWN( )	afx_msg void OnKeyDown(UINT, UINT, UINT);
ON_WM_KEYUP( )	afx_msg void OnKeyUp(UINT, UINT, UINT);
ON_WM_KILLFOCUS( )	afx_msg void OnKillFocus(CWnd*);
ON_WM_LBUTTONDBLCLK( )	afx_msg void OnLButtonDblClk(UINT, CPoint);
ON_WM_LBUTTONDOWN( )	afx_msg void OnLButtonDown(UINT, CPoint);
ON_WM_LBUTTONUP( )	afx_msg void OnLButtonUp(UINT, CPoint);
ON_WM_MBUTTONDBLCLK( )	afx_msg void OnMButtonDblClk(UINT, CPoint);
ON_WM_MBUTTONDOWN( )	afx_msg void OnMButtonDown(UINT, CPoint);
ON_WM_MBUTTONUP( )	afx_msg void OnMButtonUp(UINT, CPoint);
ON_WM_MDIACTIVATE( )	afx_msg void OnMDIActivate(BOOL, CWnd*, CWnd*);
ON_WM_MEASUREITEM( )	afx_msg void OnMeasureItem(LPMEASUREITEMSTRUCT);
ON_WM_MENUCHAR( )	afx_msg LONG OnMenuChar(UINT, UINT, CMenu*);
ON_WM_MENUSELECT( )	afx_msg void OnMenuSelect(UINT, UINT, HMENU);
ON_WM_MOUSEACTIVATE( )	afx_msg int OnMouseActivate(CWnd*, UINT, UINT);
ON_WM_MOUSEMOVE( )	afx_msg void OnMouseMove(UINT, CPoint);
ON_WM_MOVE( )	afx_msg void OnMove(int, int);
ON_WM_NCACTIVATE( )	afx_msg BOOL OnNcActivate(BOOL);

Map Entry	Function Prototype
ON_WM_NCCALCSIZE( )	afx_msg void OnNcCalcSize(LPRECT);
ON_WM_NCCREATE( )	afx_msg BOOL OnNcCreate(LPCREATESTRUCT);
ON_WM_NCDESTROY( )	afx_msg void OnNcDestroy( );
ON_WM_NCHITTEST( )	afx_msg UINT OnNcHitTest(CPoint);
ON_WM_NCLBUTTONDBLCLK( )	afx_msg void OnNcLButtonDblClk(UINT, CPoint);
ON_WM_NCLBUTTONDOWN( )	afx_msg void OnNcLButtonDown(UINT, CPoint);
ON_WM_NCLBUTTONUP( )	afx_msg void OnNcLButtonUp(UINT, CPoint);
ON_WM_NCMBUTTONDBLCLK( )	afx_msg void OnNcMButtonDblClk(UINT, CPoint);
ON_WM_NCMBUTTONDOWN( )	afx_msg void OnNcMButtonDown(UINT, CPoint);
ON_WM_NCMBUTTONUP( )	afx_msg void OnNcMButtonUp(UINT, CPoint);
ON_WM_NCMOUSEMOVE( )	afx_msg void OnNcMouseMove(UINT, CPoint);
ON_WM_NCPAINT( )	afx_msg void OnNcPaint( );
ON_WM_NCRBUTTONDBLCLK( )	afx_msg void OnNcRButtonDblClk(UINT, CPoint);
ON_WM_NCRBUTTONDOWN( )	afx_msg void OnNcRButtonDown(UINT, CPoint);
ON_WM_NCRBUTTONUP( )	afx_msg void OnNcRButtonUp(UINT, CPoint);
ON_WM_PAINT( )	afx_msg void OnPaint( );
ON_WM_PAINTCLIPBOARD( )	afx_msg void OnPaintClipboard(CWnd*, HANDLE);
ON_WM_PAINTICON( )	afx_msg void OnPaintIcon( );
ON_WM_PALETTECHANGED( )	afx_msg void OnPaletteChanged(CWnd*);
ON_WM_PARENTNOTIFY( )	afx_msg void OnParentNotify(UINT, LONG);
ON_WM_QUERYDRAGICON( )	afx_msg HCURSOR OnQueryDragIcon( );
ON_WM_QUERYENDSESSION( )	afx_msg BOOL OnQueryEndSession( );
ON_WM_QUERYNEWPALETTE( )	afx_msg BOOL OnQueryNewPalette( );
ON_WM_QUERYOPEN( )	afx_msg BOOL OnQueryOpen( );

Map Entry	Function Prototype
ON_WM_RBUTTONDBLCLK( )	afx_msg void OnRButtonDblClk(UINT, CPoint);
ON_WM_RBUTTONDOWN( )	afx_msg void OnRButtonDown(UINT, CPoint);
ON_WM_RBUTTONUP( )	afx_msg void OnRButtonUp(UINT, CPoint);
ON_WM_RENDERALLFORMATS( )	afx_msg void OnRenderAllFormats( );
ON_WM_RENDERFORMAT( )	afx_msg void OnRenderFormat(UINT);
ON_WM_SETCURSOR( )	afx_msg BOOL OnSetCursor(CWnd*, UINT, UINT);
ON_WM_SETFOCUS( )	afx_msg void OnSetFocus(CWnd*);
ON_WM_SHOWWINDOW( )	afx_msg void OnShowWindow(BOOL, UINT);
ON_WM_SIZE( )	afx_msg void OnSize(UINT, int, int);
ON_WM_SIZECLIPBOARD( )	afx_msg void OnSizeClipboard(CWnd*, HANDLE);
ON_WM_SPOOLERSTATUS( )	afx_msg void OnSpoolerStatus(UINT, UINT);
ON_WM_SYSCHAR( )	afx_msg void OnSysChar(UINT, UINT, UINT);
ON_WM_SYSCOLORCHANGE( )	afx_msg void OnSysColorChange( );
ON_WM_SYSCOMMAND( )	afx_msg void OnSysCommand(UINT, LONG);
ON_WM_SYSDEADCHAR( )	afx_msg void OnSysDeadChar(UINT, UINT, UINT);
ON_WM_SYSKEYDOWN( )	afx_msg void OnSysKeyDown(UINT, UINT, UINT);
ON_WM_SYSKEYUP( )	afx_msg void OnSysKeyUp(UINT, UINT, UINT);
ON_WM_TIMECHANGE( )	afx_msg void OnTimeChange( );
ON_WM_TIMER( )	afx_msg void OnTimer(UINT);
ON_WM_VKEYTOITEM( )	afx_msg int OnVKeyToItem(UINT, CWnd*, UINT);
ON_WM_VSCROLL( )	afx_msg void OnVScroll(UINT, UINT, CWnd*);
ON_WM_VSCROLLCLIPBOARD( )	afx_msg void OnVScrollClipboard(CWnd*, UINT, UINT);
ON_WM_WININICHANGE( )	afx_msg void OnWinIniChange(LPSTR);

6.6 User-Defined Message Codes

Map Entry	Function Prototype
ON_MESSAGE(*message, memberFxn* **)**	**afx_msg LONG** memberFxn(**UINT, LONG**);
ON_REGISTERED_MESSAGE (*nMessageVariable memberFxn*)	**afx_msg LONG** memberFxn(**UINT, LONG**);

For more about the **DECLARE_MESSAGE_MAP**, **BEGIN_MESSAGE_MAP**, and **END_MESSAGE_MAP** macros, see help.

Chapter 7

Structures and Enumerated Values for Windows

This chapter lists data structures used by the Microsoft Foundation Windows classes, as well as Clipboard and mouse enumerated values.

7.1 Structures

The following data structures are presented in alphabetical order. The structure definition is followed by a description of each field.

COMPAREITEMSTRUCT

```
typedef struct tagCOMPAREITEMSTRUCT {
    WORD   CtlType;
    WORD   CtlID;
    HWND   hwndItem;
    WORD   itemID1;
    DWORD  itemData1;
    WORD   itemID2;
    DWORD  itemData2;
} COMPAREITEMSTRUCT;
```

The **COMPAREITEMSTRUCT** structure supplies the identifiers and application-supplied data for two items in a sorted owner-draw combo box or list box.

Whenever an application adds a new item to an owner-draw combo or list box created with the **CBS_SORT** or **LBS_SORT** style, Windows sends the owner a **WM_COMPAREITEM** message. Override **OnCompareItem** to compare the two items and return a value indicating which item sorts before the other.

Members

CtlType
Is **ODT_LISTBOX** (which specifies an owner-draw list box) or **ODT_COMBOBOX** (which specifies an owner-draw combo box).

CtlID
Is the control ID for the list box or combo box.

hwndItem
Is the window handle of the control.

itemID1
Is the index of the first item in the list box or combo box being compared.

itemData1
Is 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
Is the index of the second item in the list box or combo box being compared.

itemData2
Is 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.

CREATESTRUCT

```
typedef struct tagCREATESTRUCT {
   LPSTR    lpCreateParams;
   HANDLE   hInstance;
   HANDLE   hMenu;
   HWND     hwndParent;
   int      cy;
   int      cx;
   int      y;
   int      x;
   LONG     style;
   LPSTR    lpszName;
   LPSTR    lpszClass;
   DWORD    dwExStyle;
} CREATESTRUCT;
```

The **CREATESTRUCT** structure defines the parameters used to initialize a window. When a window is created, it receives a **WM_CREATE** message with a pointer to this structure. For more information, see **CWnd::OnCreate**.

Members

lpCreateParams
Points to data to be used for creating 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.

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, the 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, the 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.

dwExStyle
Specifies extended style for the new window.

DELETEITEMSTRUCT

```
typedef struct tagDELETEITEMSTRUCT {
    WORD  CtlType
    WORD  CtlID;
    WORD  itemID;
    HWND  hwndItem;
    DWORD itemData;
} DELETEITEMSTRUCT;
```

The **DELETEITEMSTRUCT** structure describes a deleted owner-draw list-box or combo-box item. When an item is removed from the list box or combo box, or when the list box or combo box is destroyed, Windows sends the **WM_DELETEITEM** message to the owner for each deleted item along with a pointer to this structure. For more information, see **CWnd::OnDeleteItem**.

Members

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 owner-defined value that was assigned to this item when it was created.

DRAWITEMSTRUCT

```
typedef struct tagDRAWITEMSTRUCT {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemAction;
    WORD   itemState;
    HWND   hwndItem;
    HDC    hDC;
    RECT   rcItem;
    DWORD  itemData;
} DRAWITEMSTRUCT;
```

The **DRAWITEMSTRUCT** structure provides information the owner needs to determine how to paint an owner-draw control. The owner of the owner-draw control receives a pointer to this structure with a **WM_DRAWITEM** message. For more information, see **CWnd::OnDrawItem**.

Members

CtlType
Is the control type. The values for control types are as follows:

Value	Meaning
ODT_BUTTON	Owner-draw button
ODT_COMBOBOX	Owner-draw combo box
ODT_LISTBOX	Owner-draw list box
ODT_MENU	Owner-draw menu

CtlID
Is the control ID for a combo box, list box, or button. This member is not used for a menu.

itemID
Is 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 can be –1. This 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. This indicates to the user whether the list box or combo box has 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:

Value	Meaning
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. The **itemState** member 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:

Value	Meaning
ODS_CHECKED	This bit is set if the menu item is to be checked. This bit is used only in a menu.
ODS_DISABLFD	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. It contains the handle of the menu (**HMENU**) containing the item for menus.

hDC
Identifies a device context. This device context must be used when performing drawing operations on the control.

rcItem
Is 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 does not clip menu items. When drawing menu items, the owner

must ensure that the owner does not draw outside the boundaries of the rectangle defined by the **rcItem** member.

itemData
Contains the owner-defined value that was assigned to this item when it was created.

MEASUREITEMSTRUCT

```
typedef struct tagMEASUREITEMSTRUCT {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemWidth;
    WORD   itemHeight;
    DWORD  itemData
} MEASUREITEMSTRUCT;
```

When an owner-draw control is created, Windows sends the **WM_MEASUREITEM** message to the owner of the control, along with a pointer to a **MEASUREITEMSTRUCT** data structure.

The **MEASUREITEMSTRUCT** data structure must be filled in order for Windows to process user interaction with the control correctly. For more information, see **CWnd::OnMeasureItem**.

The **MEASUREITEMSTRUCT** data structure informs Windows of the dimensions of an owner-draw control. This allows Windows to correctly process user interaction with the control. The owner of an owner-draw control receives a pointer to this structure as the *lParam* parameter of an **WM_MEASUREITEM** message. The owner-draw control sends this message to its owner window when the control is created. The owner then fills in the appropriate members in the structure for the control and returns. This structure is common to all owner-draw controls.

Members

CtlType
Is the control type. The values for control types are as follows:

Value	Meaning
ODT_BUTTON	Owner-draw button
ODT_COMBOBOX	Owner-draw combo box
ODT_LISTBOX	Owner-draw list box
ODT_MENU	Owner-draw menu

CtlID
Is the control ID for a combo box, list box, or button. This member is not used for a menu.

itemID
Is 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 returning from the message.

itemHeight
Specifies the height of an individual item in a list box or a menu. Before returning 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
Contains the owner-defined value that was assigned to this item when it was created.

Remarks

Failure to assign values to **itemWidth** and **itemHeight** members in the **MEASUREITEMSTRUCT** structure will cause improper operation of the control.

PAINTSTRUCT

```
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 zero 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.

POINT

```
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.

RECT

```
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.

Remarks

Neither the width nor height of the rectangle defined by the **RECT** structure can exceed 32,767 units.

7.2 Clipboard Enumerated Values

The following list shows the enumerated values that specify system-defined Clipboard formats:

Value	Meaning
CF_BITMAP	The data is a bitmap.
CF_DIB	The data is a memory block containing a **BITMAPINFO** structure followed by the bitmap data.
CF_DIF	The data is in Data Interchange Format (Software Arts).

Value	Meaning
CF_DSPBITMAP	The data is a bitmap representation of a private format. This data is displayed in bitmap format in lieu of the privately formatted data.
CF_DSPMETAFILEPICT	The data is a metafile representation of a private data format. This data is displayed in metafile-picture format in lieu of the privately formatted data.
CF_DSPTEXT	The data is a textual representation of a private data format. This data is displayed in text format in lieu of the privately formatted data.
CF_METAFILEPICT	The data is a metafile (for more information, see description of **METAFILEPICT** structure).
CF_OEMTEXT	The data is an array of text characters in the OEM character set. Each line ends with a carriage return–linefeed combination. A null character signals the end of the data.
CF_OWNERDISPLAY	The data is in a private format that the Clipboard owner must display.
CF_PALETTE	The data is a color palette.
CF_SYLK	The data is in Microsoft Symbolic Link (SYLK) format.
CF_TEXT	The data is an array of text characters. Each line ends with a carriage return–linefeed combination. A null character signals the end of the data.
CF_TIFF	The data is in Tag Image File Format.

Private data formats in the range of **CF_PRIVATEFIRST** to **CF_PRIVATELAST** are not automatically freed when the data is deleted from the Clipboard. Data handles associated with these formats should be freed upon receiving a **WM_DESTROYCLIPBOARD** message.

Private data formats in the range of **CF_GDIOBJFIRST** to **CF_GDIOBJLAST** will be automatically deleted with a call to **CGdiObject::DeleteObject** when the data is deleted from the Clipboard.

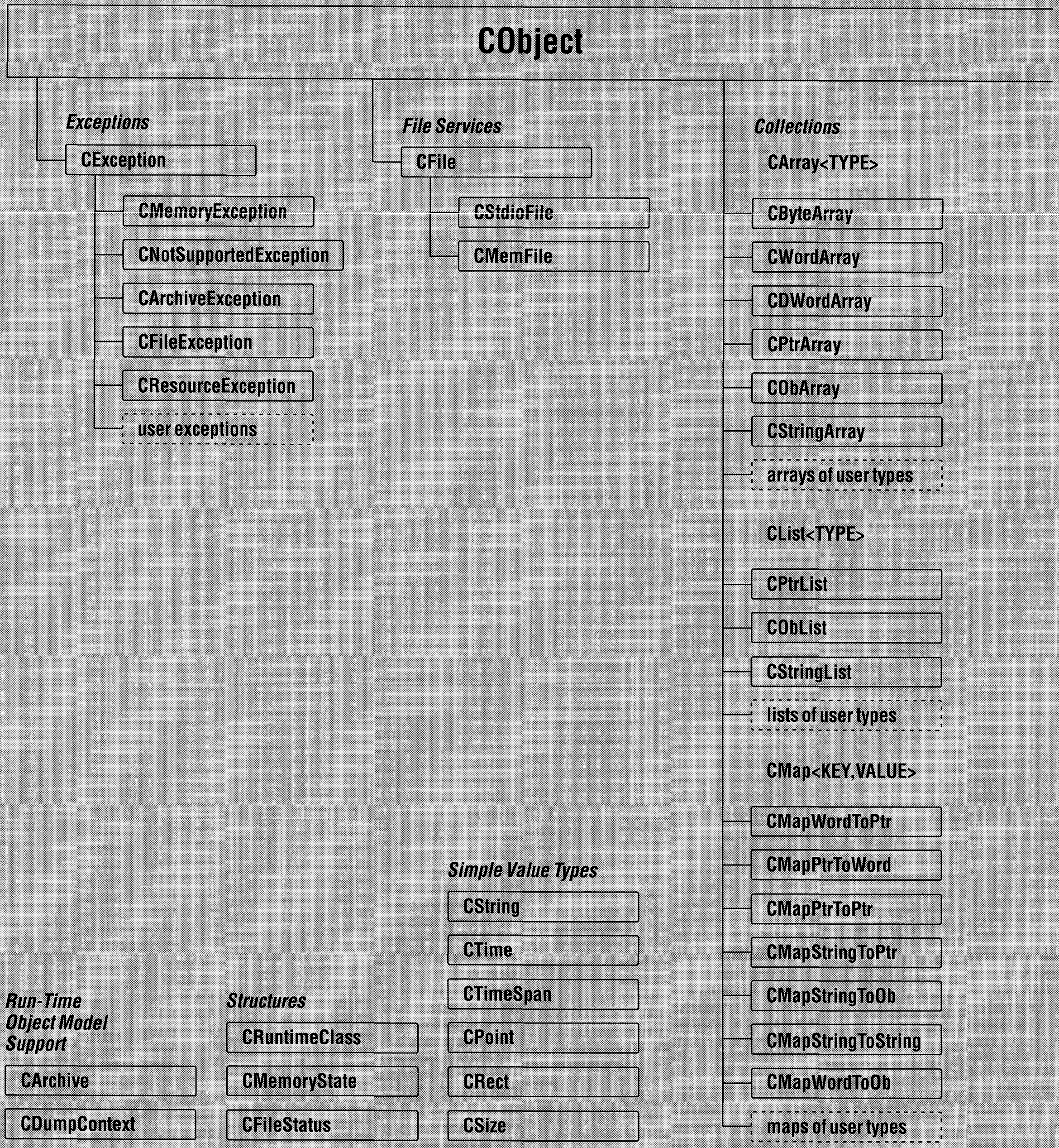
7.3 Mouse Enumerated Values

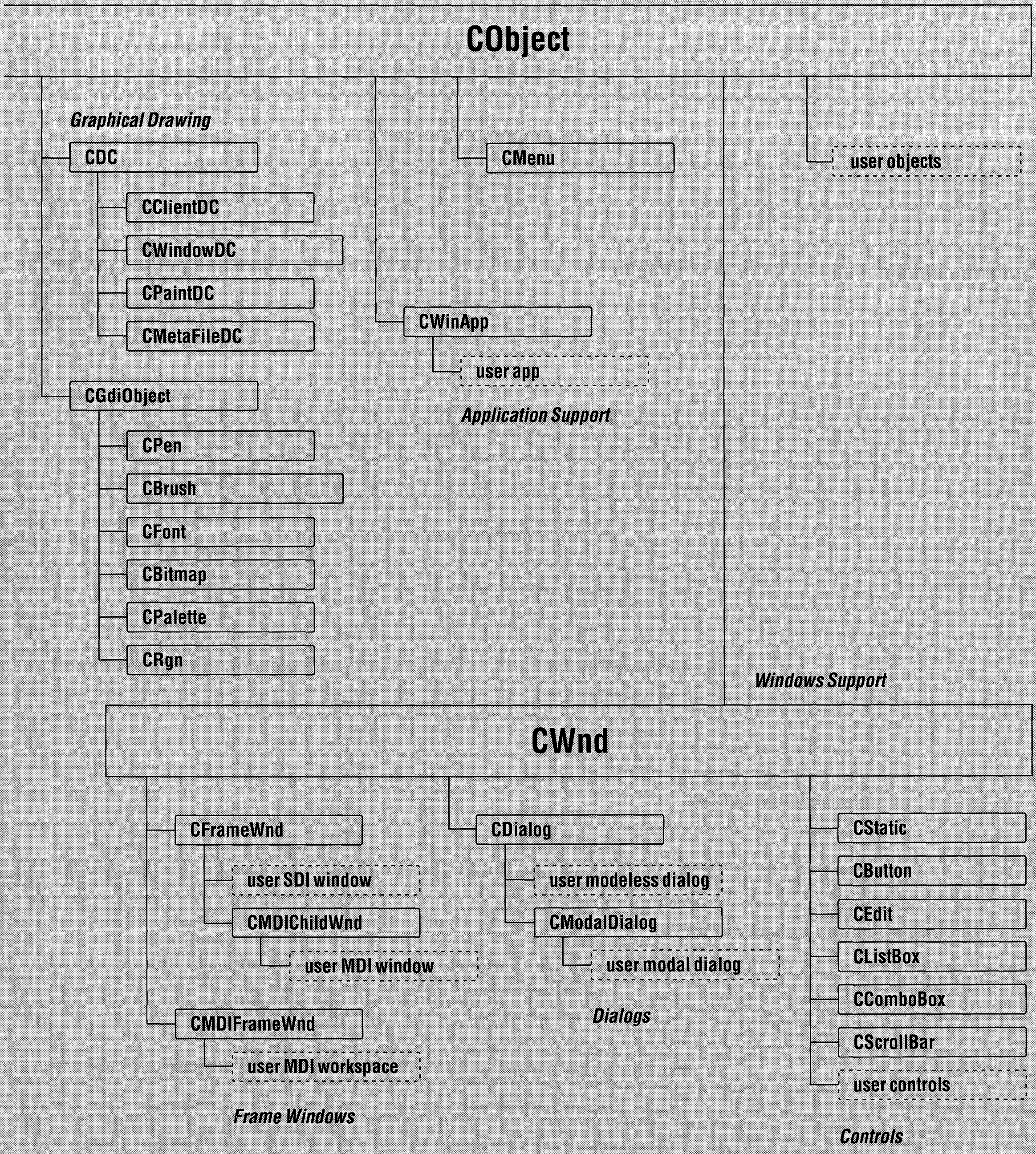
The following enumerated values are passed to the **CWnd::On*Message*** member functions that handle mouse messages, such as **CWnd::OnMouseActivate** and **CWnd::OnNcLButtonDblClk**.

Value	Meaning
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 caption area.
HTCLIENT	In a client area.
HTERROR	Same as **HTNOWHERE** except that default message processing 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.
HTMENU	In a menu area.
HTNOWHERE	On the screen background or on a dividing line between the windows.
HTREDUCE	In a Minimize box.
HTRIGHT	In the right border of the window.
HTSIZE	Same as **HTGROWBOX**.
HTSYSMENU	In a control-menu box (close box in child windows).
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 box.

Part 2

The Microsoft Foundation Class Library Reference


CObject
Exceptions
CException
CMemoryException
CNotSupportedException
CArchiveException
CFileException
CResourceException
user exceptions
File Services
CFile
CStdioFile
CMemFile
Collections
CArray<TYPE>
CByteArray
CWordArray
CDWordArray
CPtrArray
CObArray
CStringArray
arrays of user types
CList<TYPE>
CPtrList
CObList
CStringList
lists of user types
CMap<KEY,VALUE>
CMapWordToPtr
CMapPtrToWord
CMapPtrToPtr
CMapStringToPtr
CMapStringToOb
CMapStringToString
CMapWordToOb
maps of user types
Simple Value Types
CString
CTime
CTimeSpan
CPoint
CRect
CSize
Run-Time Object Model Support
CArchive
CDumpContext
Structures
CRuntimeClass
CMemoryState
CFileStatus


CObject
Graphical Drawing
CDC
CClientDC
CWindowDC
CPaintDC
CMetaFileDC
CGdiObject
CPen
CBrush
CFont
CBitmap
CPalette
CRgn
CMenu
CWinApp
user app
Application Support
user objects
Windows Support
CWnd
CFrameWnd
user SDI window
CMDIChildWnd
user MDI window
CMDIFrameWnd
user MDI workspace
Frame Windows
CDialog
user modeless dialog
CModalDialog
user modal dialog
Dialogs
CStatic
CButton
CEdit
CListBox
CComboBox
CScrollBar
user controls
Controls

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.

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**

Preconditions 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.

Public Members

Construction/Destruction

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

Flush	Flushes unwritten data from the archive buffer.
operator <<	Stores objects and primitive types to the archive.
operator >>	Loads objects and primitive types from the archive.
Read	Reads raw bytes.
Write	Writes raw bytes.

Status

GetFile	Gets the **CFile** object pointer for this archive.
IsLoading	Determines if the archive is loading.
IsStoring	Determines if the archive is storing.

Protected Members

Object Input/Output

ReadObject	Calls an object's **Serialize** function for loading.
WriteObject	Calls an object's **Serialize** function for storing.

Member Functions

CArchive::CArchive

Syntax

CArchive(CFile* *pFile*, **UINT** *nMode*, **int** *nBufSize* **= 512,**
void FAR* *lpBuf* **= NULL)**
throw(CMemoryException, CArchiveException, CFileException);

Parameters

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:

Value	Meaning
CArchive::load	Load data from the archive. Requires only **CFile** read permission.
CArchive::store	Save data to the archive. Requires **CFile** write permission.

nBufSize
An integer that specifies the size of the internal file buffer, in bytes.

Note 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.

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 );
```

See Also **CArchive::Close**, **CArchive::Flush**, **CFile::Close**

CArchive::~CArchive

Syntax **~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

Syntax **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

Syntax

void Flush()
throw(CFileException);

Remarks

Forces any data remaining in the archive buffer to be written to the file.

The 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

Syntax

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

Syntax

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**.

Example

```
int i;
extern CArchive ar;
if( ar.IsLoading() )
  ar >> i;
else
  ar << i;
```

See Also

CArchive::IsStoring

CArchive::IsStoring

Syntax

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**.

Example

```
int i;
extern CArchive ar;
if( ar.IsStoring() )
  ar << i;
else
  ar >> i;
```

See Also

CArchive::IsLoading

CArchive::Read

Syntax

UINT Read(void FAR* *lpBuf*, **UINT** *nMax* **)**
throw(CFileException);

Parameters

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

Syntax

Protected:
CObject* ReadObject(const CRuntimeClass* *pClass* **)**
throw(CFileException, CArchiveException, CMemoryException);

Parameters

pClass
A constant pointer to the **CRuntimeClass** structure that corresponds to the object that 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 protected function is usually called by the public **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

Syntax

void Write(const void FAR* *lpBuf*, **UINT** *nMax* **)**
throw(CFileException);

Parameters

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.

Example

```
extern CArchive ar;
char pb[100];
ar.Write( pb, 100 );
```

See Also

CArchive::Read

CArchive::WriteObject

Syntax

Protected:
void WriteObject(const CObject* *pOb* **)**
throw(CFileException, CArchiveException);

Parameters

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 protected function is normally called by the public **CArchive** insertion (**<<**) operator, overloaded for **CObject**. **WriteObject**, in turn, calls the **Serialize** function of the archived class.

You must use the **IMPLEMENT_SERIAL** macro to enable archiving. **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) is an implementation detail and 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 >>

Syntax

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);

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.

Example

```
int i;
extern CArchive ar;
if( ar.IsLoading() )
  ar >> i;
```

See Also

CArchive::ReadObject, **CObject::Serialize**

CArchive::operator <<

Syntax

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);

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.

Example

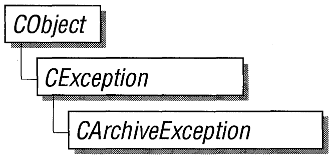
```
long l;
int i;
extern CArchive ar;
if( ar.IsStoring() )
  ar >> l >> i;
```

See Also

CArchive::WriteObject, CObject::Serialize

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.



#include <afx.h>

See Also **CArchive**, **AfxThrowArchiveException**, Chapter 5, "Exception Processing"

Comments **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.

Public Members

Data Members

m_cause	Indicates the exception cause.

Construction/Destruction

CArchiveException	Constructs a **CArchiveException** object.

Member Functions

CArchiveException::CArchiveException

Syntax

CArchiveException(int *cause* **= CArchiveException::none);**

Parameters

cause
An enumerated type variable that indicates the reason for the exception. See the **m_cause** data member for a list of the enumerators.

Remarks

Constructs a **CArchiveException** object, storing the *cause* code in the object. You can create a **CArchiveException** object on the heap and throw it yourself or let **AfxThrowArchiveException** handle it for you.

Do not use this constructor directly, but call the global function **AfxThrowArchiveException**.

Data Members

CArchiveException::m_cause

Syntax

int m_cause;

Remarks

Specifies the cause of the exception. Its values are defined by a **CArchiveException** enumerated type. The enumerators are:

Value	Meaning
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.

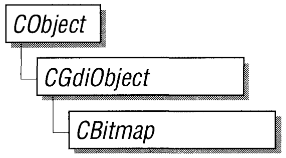
Note These **CArchiveException** cause enumerators are distinct from the **CFileException** cause enumerators.

Example

```
extern CFile f;
TRY
{
    CArchive ar( &f, CArchive::store );
}
CATCH( CArchiveException, e)
{
    if( e->m_cause == CArchiveException::readOnly )
        printf( "ERROR: Archive is read-only\\n" );
}
END_CATCH
```

class CBitmap : public CGdiObject

The **CBitmap** class encapsulates a Windows graphical design 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.



Public Members

Construction/Destruction

CBitmap	Constructs a **CBitmap** object.

Initialization

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 with a specified width, height, and bit pattern.
CreateBitmapIndirect	Initializes the object with a bitmap that has 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

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

Syntax

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

Syntax

BOOL CreateBitmap(int *nWidth*, **int** *nHeight*, **BYTE** *nPlanes*,
   BYTE *nBitcount*, **LPSTR** *lpBits* **);**

Parameters

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 the Windows Software Development Kit documentation.

Remarks

Initializes a device-dependent memory bitmap that has the specified width, height, and bit pattern. 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 **CMetaFileDC::SelectObject** and copied to any compatible device context by using the **CDC::BitBlt** function. When an application has finished using the bitmap created by the **CreateBitmap** function, it should select the bitmap out of the device context.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CDC::SelectObject, **CMetaFileDC::SelectObject**, **CDC::BitBlt**, **::CreateBitmap**

CBitmap::CreateBitmapIndirect

Syntax

BOOL CreateBitmapIndirect(LPBITMAP *lpBitmap* **);**

Parameters

lpBitmap

Points to a **BITMAP** structure that contains information about the bitmap.

The **BITMAP** structure has the following form:

```
typedef struct tagBITMAP {
   int    bmType;
   int    bmWidth;
   int    bmHeight;
   int    bmWidthBytes;
   BYTE   bmPlanes;
   BYTE   bmBitsPixel;
   LPSTR  bmBits;
} 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 **CMetaFileDC::SelectObject** and copied to any compatible device context by using the **CDC::BitBlt** function.

When an application has finished using the bitmap initialized by **CreateBitmapIndirect**, it should select the bitmap out of the device context.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CDC::SelectObject, **CMetaFileDC::SelectObject**, **CDC::BitBlt**, **::CreateBitmapIndirect**

CBitmap::CreateCompatibleBitmap

Syntax

BOOL CreateCompatibleBitmap(CDC* *pDC*, **int** *nWidth*, **int** *nHeight* **);**

Parameters

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 are finished with a **CBitmap** initialized with **CreateCompatibleBitmap**, you must select the bitmap out of the device context.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

::CreateCompatibleBitmap

CBitmap::CreateDiscardableBitmap

Syntax

BOOL CreateDiscardableBitmap(CDC* *pDC*, **int** *nWidth*, **int** *nHeight* **);**

Parameters

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** or **CMetaFileDC::SelectObject** function will return **NULL**.

When an application has finished using the bitmap created by the **CreateBitmapIndirect** function, it should select the bitmap out of the device context.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

::CreateDiscardableBitmap

CBitmap::FromHandle

Syntax

static CBitmap* FromHandle(HBITMAP *hBitmap* **);**

Parameters

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

Syntax

DWORD GetBitmapBits(DWORD *dwCount*, **LPSTR** *lpBits* **) const;**

Parameters

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 that is 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

Syntax

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** 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

Syntax

BOOL LoadBitmap(const char FAR* *lpBitmapName* **);**

BOOL LoadBitmap(UINT *nIDBitmap* **);**

Parameters

lpBitmapName
Points to a null-terminated string that contains the name of the bitmap resource.

nIDBitmap
Specifies the resource ID number of the bitmap resource.

Remarks

Loads the bitmap resource named by *lpBitmapName* or identified by the ID number in *nIDBitmap* from the application's executable file. The loaded bitmap is attached to the **CBitmap** object.

If the bitmap identified by *lpBitmapName* does not exist or if there is insufficient memory to load the bitmap, the function returns **FALSE**.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CBitmap::LoadOEMBitmap, **::LoadBitmap**

CBitmap::LoadOEMBitmap

Syntax

BOOL LoadOEMBitmap(UINT *nIDBitmap* **);**

Parameters

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 **TRUE** if successful; otherwise **FALSE**.

See Also **CBitmap::LoadBitmap**, **::LoadBitmap**

CBitmap::SetBitmapBits

Syntax **DWORD SetBitmapBits(DWORD** *dwCount*, **LPSTR** *lpBits* **);**

Parameters

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

Syntax **CSize SetBitmapDimension(int** *nWidth*, **int** *nHeight* **);**

Parameters

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. These values are not used internally by GDI; the **GetBitmapDimension** function can be used to retrieve them.

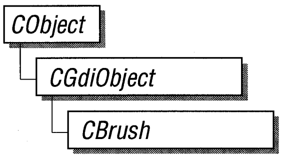
Return Value

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 CBrush : public CGdiObject



The **CBrush** class encapsulates a Windows graphical design 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.

See Also **CBitmap**, **CDC**

Public Members

Construction/Destruction

CBrush	Constructs a **CBrush** object.

Initialization

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

FromHandle	Returns a pointer to a **CBrush** object when given a handle to a Windows **HBRUSH** object.

Member Functions

CBrush::CBrush

Syntax

CBrush();

CBrush(DWORD *crColor* **)**
throw(CResourceExceptio*n* **);**

CBrush(int *nIndex***, DWORD** *crColor* **)**
throw(CResourceException);

CBrush(CBitmap* *pBitmap* **)**
throw(CResourceException);

Parameters

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:

Value	Meaning
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 **DWORD** 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

Syntax

BOOL CreateBrushIndirect(LPLOGBRUSH *lpLogBrush* **);**

Parameters

lpLogBrush
Points to a **LOGBRUSH** structure that contains information about the brush.

The **LOGBRUSH** structure has the following form:

```
typedef struct tagLOGBRUSH {
    WORD      lbStyle;
    COLORREF  lbColor;
    short 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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CBrush::CreateDIBPatternBrush, **CBrush::CreatePatternBrush**, **CBrush::CreateSolidBrush**, **CBrush::CreateHatchBrush**, **CGdiObject::CreateStockObject**, **::CreateBrushIndirect**

CBrush::CreateDIBPatternBrush

Syntax

BOOL CreateDIBPatternBrush(GLOBALHANDLE *hPackedDIB*, **UINT** *wUsage* **);**

Parameters

hPackedDIB
Identifies a global-memory object containing a packed device-independent bitmap.

wUsage
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:

Value	Meaning
DIB_PAL_COLORS	The color table contains literal RGB values.
DIB_RGB_COLORS	The color table consists of an array of 16-bit indexes.

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, you 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.

When an application selects a two-color DIB pattern brush into a monochrome device context, Windows 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

TRUE if successful; otherwise **FALSE**.

See Also

CBrush::CreatePatternBrush, CBrush::CreateBrushIndirect, CBrush::CreateSolidBrush, CBrush::CreateHatchBrush, CGdiObject::CreateStockObject, ::CreateDIBPatternBrush, ::GlobalAlloc

CBrush::CreateHatchBrush

Syntax

BOOL CreateHatchBrush(int *nIndex*, **DWORD** *crColor* **);**

Parameters

nIndex
Specifies the hatch style of the brush. It can be one of the following values:

Value	Meaning
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 **TRUE** if successful; otherwise **FALSE**.

See Also **CBrush::CreateBrushIndirect**, **CBrush::CreateDIBPatternBrush**, **CBrush::CreatePatternBrush**, **CBrush::CreateSolidBrush**, **CGdiObject::CreateStockObject**, **::CreateHatchBrush**

CBrush::CreatePatternBrush

Syntax **BOOL CreatePatternBrush(CBitmap*** *pBitmap* **);**

Parameters *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 bitmap identified by *pBitmap* is typically initialized by using the **CBitmap::CreateBitmap**, **CBitmap::CreateBitmapIndirect**, **CBitmap::LoadBitmap**, or **CBitmap::CreateCompatibleBitmap** function.

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 upper-left corner of the bitmap.

A pattern brush can be deleted without affecting the associated bitmap. This means 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

TRUE if successful; otherwise **FALSE**.

See Also

CBrush::CreateBrushIndirect, **CBrush::CreateDIBPatternBrush**, **CBrush::CreateHatchBrush**, **CBrush::CreateSolidBrush**, **CGdiObject::CreateStockObject**, **CBitmap::CreateBitmap**, **CBitmap::CreateBitmapIndirect**, **CBitmap::CreateCompatibleBitmap**, **CBitmap::LoadBitmap**, **::CreatePatternBrush**

CBrush::CreateSolidBrush

Syntax

BOOL CreateSolidBrush(DWORD *crColor* **);**

Parameters

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 subsequently be selected as the current brush for any device context.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CBrush::CreateBrushIndirect, **CBrush::CreateDIBPatternBrush**, **CBrush::CreateHatchBrush**, **CBrush::CreatePatternBrush**, **::CreateSolidBrush**

CBrush::FromHandle

Syntax

static CBrush* FromHandle(HBRUSH *hBrush* **);**

Parameters

hBrush
HANDLE to a Windows GDI brush.

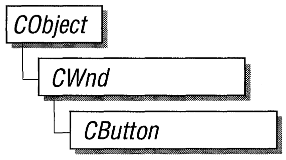
Remarks

Returns a pointer to a **CBrush** object when given a handle to a Windows **HBRUSH** object. If a **CBrush** object is not already attached to the handle, a temporary **CBrush** object is created and attached. This temporary **CBrush** object 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. 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 **CBrush** object if successful; **NULL** if not.

class CButton : public CWnd



The **CButton** class provides the functionality of Windows button control. 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 push button. A **CButton** object can become any of these, according to the style specified at its initialization by the **Create** member function.

You create a button control in two steps. 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 the Windows notification messages sent by a **CButton** object to its parent (usually a class derived from **CDialog** or **CModalDialog**), add the appropriate message-map entries and message-handler member functions to the parent class to handle the messages you want to process. Potential message-map entries are:

ON_COMMAND
ON_BN_CLICKED
ON_BN_DOUBLECLICKED

If you create a **CButton** object within a dialog box (through a dialog resource), the **CButton** object is automatically destroyed when the user closes the dialog box.

If you create a **CButton** object within a window, you may also need to destroy it. If you create the **CButton** object on the stack, it is destroyed automatically. 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 allocate any memory in the **CButton** object, override the **CButton** destructor to dispose of the allocations.

See Also **CWnd**, **CComboBox**, **CEdit**, **CListBox**, **CScrollBar**, **CStatic**, **CModalDialog**, **CDialog**

Public Members

Construction/Destruction

CButton	Constructs a **CButton** object.

Initialization

Create	Creates the Windows button control and attaches it to the **CButton** object.

Operations

GetState	Retrieves the 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.

Member Functions

CButton::CButton

Syntax

CButton();

Remarks

Constructs a **CButton** object.

See Also

CButton::Create

CButton::Create

Syntax

BOOL Create(const char FAR* *lpCaption*, **DWORD** *dwStyle*,
const RECT& *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

Parameters

lpCaption
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 resource 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.

When **Create** executes, Windows sends the **WM_NCCREATE**, **WM_CREATE**, **WM_NCCALCSIZE,** and **WM_GETMINMAXINFO** messages to the button 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 **CButton**, 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.

To handle Windows notification messages that the **CButton** object sends to its parent, add any of the following message-map entries that you want to process to the parent-class message map:

ON_COMMAND
ON_BN_CLICKED
ON_BN_DOUBLECLICKED

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:

Style	Application
WS_CHILD	Always.
WS_VISIBLE	Usually.
WS_DIABLED	Rarely.
WS_GROUP	To group controls.
WS_TABSTOP	To include the button in the tabbing order.

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

Use any combination of the following button styles for *dwStyle*:

Value	Meaning
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.

Value	Meaning
BS_AUTO3STATE	Same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, dimmed, and normal.
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 is useful for enabling 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-draw button. The owner window receives a **WM_MEASUREITEM** message when the button is created and a **WM_DRAWITEM** message when a visual aspect of the button has changed.
BS_PUSHBUTTON	Creates a push button that posts a **WM_COMMAND** message to the owner window when the user selects the button.

Value	Meaning
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.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CButton::CButton

CButton::GetButtonStyle

Syntax

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

Syntax

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

Syntax

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 dot (.). 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.

Mask	Meaning
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

Syntax **void SetButtonStyle(UINT** *nStyle*, **BOOL** *bRedraw* = **TRUE);**

Parameters

nStyle
Specifies the button style.

bRedraw
Specifies whether the button is to be redrawn. A value of **TRUE** redraws the button. A value of **FALSE** does not redraw the button. The button is redrawn by default.

Remarks Changes the style of a button. See the **Create** member function for a list of possible button styles.

See Also **CButton::GetButtonStyle**, **BM_SETSTYLE**

CButton::SetCheck

Syntax

void SetCheck(int *nCheck* **);**

Parameters

nCheck
Specifies the check state. This parameter can be one of the following values:

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 only be used 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 push button.

See Also

CButton::GetCheck, CButton::GetState, CButton::SetState, BM_SETCHECK

CButton::SetState

Syntax

void SetState(BOOL *bHighlight* **);**

Parameters

bHighlight
Specifies whether the button is to be highlighted. A value of **TRUE** highlights the button. A value of **FALSE** removes any highlighting.

Remarks

Sets the highlighting state of a button control.

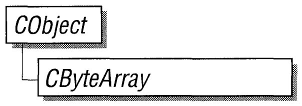
Highlighting affects the exterior of a button control. It has no effect on the check state of a radio button or check box.

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**

Public Members

Construction/Destruction

CByteArray	Constructs an empty array for bytes.

Bounds

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

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

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

SetAtGrow	Sets the value for a given index; grows the array if necessary.
Add	Adds an element to the end of the array.

Insertion/Removal

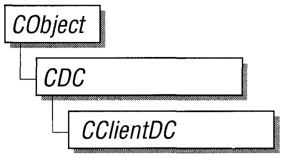
InsertAt	Inserts an element at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

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.



See Also **CDC**

Public Members

Construction/Destruction

CClientDC	Constructs a **CClientDC** object connected to the **CWnd**.
~CClientDC	Destroys a **CClientDC** object.

Protected Members

m_hWnd	The **HWND** of the window for which this **CClientDC** is valid.

Member Functions

CClientDC::CClientDC

Syntax

CClientDC(CWnd* *pWnd* **)**
throw(CResourceException);

Parameters

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 Windows.

CClientDC::~CClientDC

Syntax

virtual ~CClientDC();

Remarks

Destroys a **CClientDC** object and calls the Windows **ReleaseDC** function.

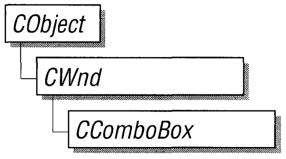
Data Members

CClientDC::m_hWnd

Remarks

The **HWND** of the **CWnd** pointer used to construct the **CClientDC** object.

class CComboBox : public CWnd



The **CComboBox** class provides the functionality of a Windows combo box. A combo box consists of an edit control plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a drop-down arrow next to the edit control, depending on the style of the combo box.

Depending on the style of the combo box, the user may or may not be able to edit the contents of the edit control. If the list box is visible, typing characters into the edit control will cause the first list-box entry that matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the edit control.

You create a combo box in two steps. First call the constructor **CComboBox** to construct the **CComboBox** object, then call the **Create** member function to create the button control and attach it to the **CComboBox** object.

Construction can be a one-step process in a class derived from **CComboBox**. Write a constructor for the derived class and call **Create** from within the constructor.

If you want to handle the Windows notification messages sent by a **CComboBox** object to its parent (usually a class derived from **CDialog** or **CModalDialog**), add the appropriate message-map entries and message-handler member functions to the parent class to handle the messages you want to process. Potential message-map entries are:

ON_COMMAND
ON_CBN_KILLFOCUS
ON_CBN_SETFOCUS
ON_CBN_DROPDOWN
ON_CBN_DBLCLK
ON_CBN_ERRSPACE
ON_CBN_SELCHANGE
ON_CBN_EDITCHANGE
ON_CBN_EDITUPDATE

If you create a **CComboBox** object within a dialog box (through a dialog resource), the **CComboBox** is automatically destroyed when the user closes the dialog box.

If you create a **CComboBox** object within a window, you may also 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 user terminates the Windows combo box.

If you allocate any memory in the **CComboBox** object, override the **CComboBox** destructor to dispose of the allocations.

See Also **CWnd**, **CButton**, **CEdit**, **CListBox**, **CScrollBar**, **CStatic**, **CModalDialog**, **CDialog**

Public Members

Construction/Destruction

CComboBox	Constructs a **CComboBox** object.

Initialization

Create	Creates the combo box and attaches it to the **CComboBox** object.

General Operations

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.
LimitText	Limits the length of the text that the user may enter into the edit control of a combo box.
SetEditSel	Select characters in the edit control of a combo box.
GetItemData	Retrieves the application-supplied 32-bit value associated with the specified combo-box item.
SetItemData	Sets the 32-bit value associated with the specified item 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.
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.

String Operations

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.
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.

Member Functions

CComboBox::AddString

Syntax

int AddString(const char FAR* *lpString* **);**

Parameters

lpString
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

Syntax

CComboBox();

Remarks

Constructs a **CComboBox** object.

See Also

CComboBox::Create

CComboBox::Clear

Syntax

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::Copy

Syntax

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

Syntax

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

Parameters

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** or **CModalDialog**). It must not be **NULL**.

nID
Specifies the combo box's resource 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.

To handle Windows notification messages sent from a **CComboBox** object to its parent, add any of the following message-map entries that you want processed to the parent-class message map:

ON_COMMAND
ON_CBN_KILLFOCUS
ON_CBN_SETFOCUS
ON_CBN_DROPDOWN
ON_CBN_DBLCLK
ON_CBN_ERRSPACE
ON_CBN_SELCHANGE
ON_CBN_EDITCHANGE
ON_CBN_EDITUPDATE

Apply the following window styles to a combo-box control:

Style	Application
WS_CHILD	Always.
WS_VISIBLE	Usually.
WS_DIABLED	Rarely.
WS_VSCROLL	For list boxes and combo boxes.
WS_HSCROLL	For list boxes and combo boxes.
WS_GROUP	To group controls.
WS_TABSTOP	To include the combo box in the tabbing order.

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

Use any combination of the following combo-box styles for *dwStyle*:

Style	Description
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.

Style	Description
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.

Return Value

Returns **TRUE** if successful; otherwise **FALSE**.

See Also

CComboBox::CComboBox

CComboBox::Cut

Syntax

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::DeleteString

Syntax

int DeleteString(UINT *nIndex* **);**

Parameters

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

Syntax

int Dir(UINT *attr*, **const char FAR*** *lpWildCard* **);**

Parameters

attr
Can be any combination of the **enum** values from **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 *lpWildCard* specifies a directory.
0x0020	File has been archived.
0x4000	Include all drives that match the name specified by *lpWildCard.*
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.

lpWildCard
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::FindString

Syntax
int FindString(int *nStartAfter*, **const char FAR*** *lpString* **) const;**

Parameters
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.

lpString
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::GetCount

Syntax **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

Syntax **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::GetEditSel

Syntax **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 is used on a combo box without an edit control, **CB_ERR** is returned.

See Also **CComboBox::SetEditSel, CB_GETEDITSEL**

CComboBox::GetItemData

Syntax

DWORD GetItemData(int *nIndex* **) const;**

Parameters

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.

Return Value

The 32-bit value associated with the item, or **CB_ERR** if an error occurs.

See Also

CComboBox::SetItemData, CB_GETITEMDATA

CComboBox::GetLBText

Syntax

int GetLBText(int *nIndex*, **char FAR*** *lpText* **) const;**

void GetLBText(int *nIndex*, **CString&** *rString* **) const;**

Parameters

nIndex
Contains the zero-based index of the list-box string to be copied.List boxes;combo boxes, getting string from, CComboBox::GetLBText

lpText
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

Syntax

int GetLBTextLen(int *nIndex* **) const;**

Parameters

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

Syntax

int InsertString(int *nIndex*, **const char FAR*** *lpString* **);**

Parameters

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.

lpString
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

Syntax

BOOL LimitText(int *nMaxChars* **);**

Parameters

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 may 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

TRUE 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::Paste

Syntax

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

Syntax

void ResetContent();

Remarks

Removes all items from the list box and edit control of a combo box.

See Also

CB_RESETCONTENT

CComboBox::SelectString

Syntax

int SelectString(int *nStartAfter*, **const char FAR*** *lpString* **);**

Parameters

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.

lpString
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

Syntax

int SetCurSel(int *nSelect* **);**

Parameters

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

Syntax

BOOL SetEditSel(int *nStartChar*, **int** *nEndChar* **);**

Parameters

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 **TRUE** if the member function is successful; otherwise **FALSE**. 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::SetItemData

Syntax **int SetItemData(int** *nIndex*, **DWORD** *dwItemData* **);**

Parameters

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.

Return Value **CB_ERR** if an error occurs.

See Also **CComboBox::GetItemData**, **CB_SETITEMDATA**, **CComboBox::AddString**, **CComboBox::InsertString**

CComboBox::ShowDropDown

Syntax **void ShowDropDown(BOOL** *bShowIt* = **TRUE);**

Parameters

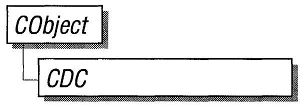
bShowIt
Specifies whether the drop-down list box is to be shown or hidden. A value of **TRUE** shows the list box. A value of **FALSE** hides the list box.

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 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 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, 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**—in particular class **CPaintDC**, which encapsulates calls to **BeginPaint** and **EndPaint**. Class **CClientDC** manages a display context associated with a window's client area. Class **CWindowDC** manages a display context associated with an entire window, including its frame and controls. Class **CMetaFileDC** associates a device context associated with a metafile.

CDC supports the Attach/Detach idiom for Windows handles described in **CWnd**.

See Also **CPaintDC**, **CWindowDC**, **CClientDC**, **CMetaFileDC**

Public Members

Construction/Destruction

CDC	Constructs a **CDC** object.
~CDC	Destroys a **CDC** object.

Initialization

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 **DC** associated with this **CDC** object.

Device-Context Functions

GetDCOrg	Obtains the final translation origin for the device context.
SaveDC	Saves the current state of the device context.
RestoreDC	Restores the device context to a previous state saved with **SaveDC**.
GetDeviceCaps	Retrieves a specified kind of device-specific information about a given display device's capabilities.

Drawing-Tool Functions

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

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

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

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

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

DPtoLP	Converts device points or rectangles into logical points or rectangles.
LPtoDP	Converts logical points or rectangles into device points or rectangles.

Region Functions

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

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

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

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

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 filled using the current brush.
RoundRect	Draws a rectangle with rounded corners using the current pen and filled using the current brush.

Bitmap Functions

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

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, using the current font to determine the dimensions.
GetTabbedTextExtent	Computes the width and height of a character string.
GrayString	Draws dimmed (gray) 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.
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

GetCharWidth	Retrieves the widths of individual characters in a consecutive group of characters from the current font.
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

Escape	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.
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 **EndDoc** escape.
EndDoc	Ends a print job started by a **StartDoc** escape.

Scrolling Functions

ScrollDC	Scrolls a rectangle of bits horizontally and vertically.

Metafile Functions

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

Syntax

int AbortDoc();

Remarks

Terminates the current print job, erasing everything the application has written to the device since the last call to **EndDoc**.

This member function is provided as a convenient way to send the **ABORTDOC** escape. It allows the application to access facilities of a particular device that are not directly available through GDI. The escape call is translated and sent to the device driver.

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.

Return Value

A positive value if successful, or a negative value if an error has occurred. The following list shows common error values:

Value	Meaning
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 **::AbortDoc**

CDC::Arc

Syntax

BOOL Arc(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3*, **int** *x4*, **int** *y4* **);**

BOOL Arc(LPRECT *lpRect*, **POINT** *ptStart*, **POINT** *ptEnd* **);**

Parameters

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 a **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.

Return Value

TRUE if the arc is drawn; otherwise **FALSE**.

See Also

CDC::Chord, **::Arc**

CDC::BitBlt

Syntax

BOOL BitBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **CDC*** *pSrcDC*, **int** *xSrc*, **int** *ySrc*, **DWORD** *dwRop* **);**

Parameters

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 graphics device interface (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*:

Code	Description
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.

Code	Description
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 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 or your own device context, you will have to register a window class rather than relying on the Foundation classes to do it for you. Use the Foundation global function **AfxRegisterWndClass**.

GDI transforms *nWidth* and *nHeight*, once by using the destination display context, and once by using the source display 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 are used in the conversion.

Note that not all device contexts support **BitBlt**. To check whether a given device context does support **BitBlt**, use **GetDeviceCaps**.

Return Value **TRUE** if the bitmap is drawn; otherwise **FALSE**.

See Also **CDC::GetDeviceCaps**, **CDC::PatBlt**, **CDC::SetTextColor**, **CDC::StretchBlt**, **::StretchDIBits**, **::BitBlt**

CDC::CDC

Syntax **CDC();**

Remarks Constructs a CDC object.

See Also **CDC::CreateDC**, **CDC::CreateIC**

CDC::~CDC

Syntax

virtual ~CDC();

Remarks

Destroys a **CDC** object. If a Windows **HDC** is attached to the **CDC** object, the destructor detaches the **HDC** and deletes it.

See Also

CDC::DeleteDC, **::DeleteDC**

CDC::Chord

Syntax

BOOL Chord(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3*, **int** *x4*, **int** *y4* **);**

BOOL Chord(LPRECT *lpRect*, **POINT** *ptStart*, **POINT** *ptEnd* **);**

Parameters

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.

Return Value

TRUE if the chord is drawn; otherwise **FALSE**.

See Also

CDC::Arc, **::Chord**

CDC::CreateCompatibleDC

Syntax

BOOL CreateCompatibleDC(CDC* *pDC* **);**

Parameters

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.

This function can only be used to create compatible device contexts for devices that support raster operations. For more information, see the **RC_BITBLT** raster capability in the member function **GetDeviceCaps**. GDI output functions can be used with a memory device context only if a bitmap has been created and selected into that context.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CDC::CDC, **CDC::GetDeviceCaps**, **::CreateCompatibleDC**

CDC::CreateDC

Syntax

BOOL CreateDC(const char FAR* *lpDriverName***,**
const char FAR* *lpDeviceName***, const char FAR*** *lpOutput***,**
LPSTR *lpInitData* **);**

Parameters

lpDriverName
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.

lpDeviceName
Points to a null-terminated string that specifies the name of the specific device to be supported (for example, EPSON FX-80). The *lpDeviceName* parameter is used if the module supports more than one device. You can also pass a **CString** object for this parameter.

lpOutput
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.

The **DEVMODE** structure has the following form:

```
#include <drvinit.h>

typedef struct _devicemode {
    char    dmDeviceName[32];
    WORD    dmSpecVersion;
    WORD    dmDriverVersion;
    WORD    dmSize;
    WORD    dmDriverExtra;
    DWORD   dmmembers;
    short   dmOrientation;
    short   dmPaperSize;
    short   dmPaperLength;
    short   dmPaperWidth;
    short   dmScale;
    short   dmCopies;
    short   dmDefaultSource;
    short   dmPrintQuality;
    short   dmColor;
    short   dmDuplex;
} DEVMODE;
```

Remarks

Creates a device context for the specified device. The *lpDriverName*, *lpDeviceName*, and *lpOutput* parameters specify the device driver, device name, and physical output medium (file or port), respectively.

The DRVINIT.H header file is required if the **DEVMODE** structure is used.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

::ExtDeviceMode, **::CreateDC**

CDC::CreateIC

Syntax

BOOL CreateIC(const char FAR* *lpDriverName*, **const char FAR*** *lpDeviceName*, **const char FAR*** *lpOutput*, **LPSTR** *lpInitData* **);**

Parameters

lpDriverName
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.

lpDeviceName
Points to a null-terminated string that specifies the name of the specific device to be supported (for example, EPSON FX-80). The *lpDeviceName* parameter is

used if the module supports more than one device. You can pass a **CString** object for this parameter.

lpOutput
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

TRUE if successful; otherwise **FALSE**.

See Also

CDC::CreateDC, **::CreateIC**

CDC::DeleteDC

Syntax

BOOL DeleteDC();

Remarks

In general, do not call this function; the destructor will do it for you. The **DeleteDC** member function deletes the Windows function **DC** attached to 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 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** class is provided to wrap this functionality.

The **DeleteDC** function is generally used to delete device contexts created with **CreateDC**, **CreateIC**, or **CreateCompatibleDC**.

Return Value

Specifies whether the device context has been deleted. **TRUE** if the device context is successfully deleted (regardless of whether the deleted device context is the last context for the device). **FALSE** if an error occurs.

See Also

CDC::CDC, **CDC::~CDC**, **::DeleteDC**

CDC::DPtoLP

Syntax

void DPtoLP(LPPOINT *lpPoints*, **int** *nCount* **= 1) const;**

void DPtoLP(LPRECT *lpRect* **) const;**

Parameters

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 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**

CDC::DrawFocusRect

Syntax

void DrawFocusRect(LPRECT *lpRect* **);**

Parameters

lpRect
Points to a **RECT** structure or a **CRect** object that specifies the coordinates of the rectangle to be drawn.

Remarks

Draws a rectangle in the style used to indicate focus.

Since this is an 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 **::DrawFocusRect**

CDC::DrawIcon

Syntax

BOOL DrawIcon(int *x*, **int** *y*, **HICON** *hIcon* **);**

BOOL DrawIcon(POINT *point*, **HICON** *hIcon* **);**

Parameters

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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CWinApp::LoadIcon, **CWinApp::LoadStandardIcon**, **CWinApp::LoadOEMIcon**, **::DrawIcon**

CDC::DrawText

Syntax

int DrawText(const char FAR* *lpString*, **int** *nCount*, **LPRECT** *lpRect*,
  UINT *nFormat* **);**

Parameters

lpString
  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 *lpString* 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):

Value	Meaning
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.

Value	Meaning
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** values 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.

Return Value

The height of the text.

See Also

::DrawText

CDC::Ellipse

Syntax

BOOL Ellipse(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

BOOL Ellipse(LPRECT *lpRect* **);**

Parameters

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.

If either the width or the height of the bounding rectangle is 0, no ellipse is drawn.

Return Value

TRUE if the ellipse is drawn; otherwise **FALSE**.

See Also

CDC::Arc, **CDC::Chord**, **::Ellipse**

CDC::EndDoc

Syntax

int EndDoc();

Remarks

Ends a print job started by a call to the **StartDoc** member function.

The member function is provided as a convenient way to send the **ENDDOC** escape. It allows the application to access facilities of a particular device that are not directly available through GDI. The escape call is translated and sent to the device driver.

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.

Return Value

Positive if the function is successful. Or a negative value if there is an error. The following list shows common error values:

Value	Meaning
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 **::EndDoc**

CDC::EndPage

Syntax **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.

The member function is provided as a convenient way to send the **NEWFRAME** escape. It allows the application to access facilities of a particular device that are not directly available through GDI. The escape call is translated and sent to the device driver.

Return Value Positive if successful; otherwise, it is an error value, which can be one of the following:

Value	Meaning
SP_ERROR	General error.
SP_APPABORT	Job was terminated because the application's abort function returned zero.
SP_USERABORT	User terminated the job through Print Manager.

Value	Meaning
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**, **::EndPage**

CDC::EnumObjects

Syntax

int EnumObjects(int *nObjectType*,
int (FAR PASCAL EXPORT* *lpfn* **)(LPSTR, LPSTR), LPSTR** *lpData* **);**

Parameters

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 new features of Microsoft C/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 **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

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 Foundation programs, you'll normally use the /GA and /GEs compiler options. The /Gw compiler option is not used with the Foundation classes. (If you do use **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 Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see Chapter 12 in the *Class Libraries User's Guide*.

Callback Function

The callback function passed to **EnumObjects** must use the Pascal calling convention and must be declared **FAR**.

int FAR PASCAL __export ObjectFunc(**LPSTR** *lpLogObject*,
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.

Parameter	Description
lpLogObject	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

::FreeProcInstance, **::MakeProcInstance**, **::EnumObjects**

CDC::Escape

Syntax

int Escape(int *nEscape*, **int** *nCount*, **LPSTR** *lpInData*, **LPSTR** *lpOutData* **);**

Parameters

nEscape
Specifies the escape function to be performed. For a complete list of escape functions, see the chapter on printer escapes in the Windows Software Development Kit documentation.

nCount
Specifies the number of bytes of data pointed to by *lpInData.*

lpInData
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 Software Development Kit documentation.

The Microsoft Foundation Class Library provides member functions for some of the most common escape functions.

Return Value

Positive if the function is successful, except for the **QUERYESCSUPPORT** escape, which only checks for implementation. Or 0 if the escape is not implemented. Or a negative value if there is an error. The following list shows common error values:

Value	Meaning
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::StartPage**, **CDC::EndPage**, **CDC::SetAbortProc**, **CDC::AbortDoc**, **CDC::EndDoc**, **::Escape**

CDC::ExcludeClipRect

Syntax

int ExcludeClipRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

int ExcludeClipRect(LPRECT *lpRect* **);**

Parameters

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.

Remarks

Creates a new clipping region that consists of the existing clipping region minus the specified rectangle.

Return Value

Specifies the new clipping region's type. It can be any one of the following values:

Value	Meaning
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

::ExcludeClipRect

CDC::ExcludeUpdateRgn

Syntax

int ExcludeUpdateRgn(CWnd* *pWnd* **);**

Parameters

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:

Value	Meaning
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

::ExcludeUpdateRgn

CDC::ExtFloodFill

Syntax

BOOL ExtFloodFill(int *x*, **int** *y*, **DWORD** *crColor*, **UINT** *nFillType* **);**

Parameters

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:

Value	Meaning
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. However, this member function provides more flexibility than **FloodFill**. 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

TRUE if the function is successful. **FALSE** 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

Syntax

BOOL ExtTextOut(int *x*, **int** *y*, **UINT** *nOptions*, **LPRECT** *lpRect*,
const char FAR* *lpString*, **UINT** *nCount*, **LPINT** *lpDxWidths* **);**

Parameters

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:

Value	Meaning
ETO_CLIPPED	Specifies that text is clipped to the rectangle.
ETO_OPAQUE	Specifies that the current background color fills the rectangle.

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.

lpString
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. This parameter can be **NULL**.

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**, using the current position instead.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::SetTextAlign, **CDC::TabbedTextOut**, **CDC::TextOut**, **::ExtTextOut**

CDC::FillRect

Syntax

void FillRect(LPRECT *lpRect*, **CBrush*** *pBrush* **);**

Parameters

lpRect
Points to a **RECT** or **CRect** 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 by using the specified brush. The function fills the complete rectangle, including the left and top borders, but does not fill the right and bottom borders.

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**, **::FillRect**

CDC::FillRgn

Syntax **BOOL FillRgn(CRgn*** *pRgn*, **CBrush*** *pBrush* **);**

Parameters

pRgn
Identifies 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*.

Return Value **TRUE** if the function is successful or **FALSE** if an error occurs.

See Also **CRgn**, **CDC::PaintRgn**, **CBrush**, **::FillRgn**

CDC::FloodFill

Syntax **BOOL FloodFill(int** *x*, **int** *y*, **DWORD** *crColor* **);**

Parameters

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

TRUE if the function is successful. **FALSE** 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

Syntax

void FrameRect(LPRECT *lpRect*, **CBrush*** *pBrush* **);**

Parameters

lpRect
Points to a **RECT** or **CRect** 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 one 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.

See Also

CBrush, **::FrameRect**

CDC::FrameRgn

Syntax

BOOL FrameRgn(CRgn* *pRgn*, **CBrush*** *pBrush*, **int** *nWidth*, **int** *nHeight* **);**

Parameters

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 in vertical brush strokes (in logical units).

nHeight
Specifies the height in horizontal brush strokes (in logical units).

Remarks

Draws a border around the region specified by *pRgn*, using the brush specified by *pBrush*. The *nWidth* parameter specifies the width of the border in vertical brush strokes; *nHeight* specifies the height in horizontal brush strokes.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

::FrameRgn, **CBrush**, **CRgn**

CDC::GetAspectRatioFilter

Syntax

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 **SetMapperFlags**.

Return Value

A **CSize** object representing the aspect ratio used by the current aspect-ratio filter.

See Also

CDC::SetMapperFlags, **::GetAspectRatioFilter**, **CSize**

CDC::GetBkColor

Syntax **DWORD 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

Syntax **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 Specifies the current background mode. It can be **OPAQUE** or **TRANSPARENT**.

See Also **CDC::GetBkColor**, **CDC::SetBkColor**, **CDC::SetBkMode**, **::GetBkMode**

CDC::GetBrushOrg

Syntax **CPoint GetBrushOrg() const;**

Remarks Retrieves the origin 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

Specifies the current origin of the brush (in device units) as a **CPoint** object.

See Also

CDC::SetBrushOrg, **CGdiObject::UnrealizeObject**, **::GetBrushOrg**

CDC::GetCharWidth

Syntax

BOOL GetCharWidth(UINT *nFirstChar*, **UINT** *nLastChar*,
LPINT *lpBuffer* **) const;**

Parameters

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. 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. For example, there must be at least 26 entries in the example given in the previous paragraph.

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

TRUE if the function is successful; otherwise **FALSE**.

See Also

::GetCharWidth

CDC::GetClipBox

Syntax

int GetClipBox(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to the **RECT** or **CRect** 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:

Value	Meaning
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

::GetClipBox

CDC::GetCurrentPosition

Syntax

CPoint GetCurrentPosition() const;

Remarks

Retrieves the current position (in logical coordinates).

Return Value

The current position as a **CPoint** object.

See Also

CDC::MoveTo, **CPoint**, **::GetCurrentPosition**

CDC::GetDCOrg

Syntax

CPoint GetDCOrg() const;

Remarks

Obtains the final translation origin for the device context. The final translation origin specifies the offset used by Windows to translate device coordinates into client coordinates for points in an application's window. The final translation origin is relative to the physical origin of the screen display.

Return Value

The final translation origin (in device coordinates) as a **CPoint** object.

See Also

CPoint, **::GetDCOrg**

CDC::GetDeviceCaps

Syntax

int GetDeviceCaps(int *nIndex* **) const;**

Parameters

nIndex
Specifies the item 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 values:

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
Flag that indicates the clipping capabilities of the device. It is 1 if the device can clip to a rectangle, 0 if it cannot.

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 version is 3.0 or higher.

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 version is 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 version is 3.0 or higher.

RASTERCAPS
Value that indicates the raster capabilities of the device, as shown in the following list:

Capability	Meaning
RC_BANDING	Requires banding support
RC_BITBLT	Capable of transferring bitmaps
RC_BITMAP64	Capable of supporting bitmaps larger than 64K
RC_DI_BITMAP	Capable of supporting **SetDIBits** and **GetDIBits**
RC_DIBTODEV	Capable of supporting the **SetDIBitsToDevice** function
RC_FLOODFILL	Capable of performing flood fills
RC_GDI20_OUTPUT	Capable of supporting Windows version 2.0 features
RC_PALETTE	Palette-based device
RC_SCALING	Capable of scaling
RC_STRETCHBLT	Capable of performing the **StretchBlt** function
RC_STRETCHDIB	Capable of performing the **StretchDIBits** function

CURVECAPS
A bitmask that indicates the curve capabilities of the device. The bits have the following meanings:

Bit	Meaning
0	Device can do circles
1	Device can do pie wedges
2	Device can do chord arcs
3	Device can do ellipses
4	Device can do wide borders
5	Device can do styled borders
6	Device can do borders that are wide and styled
7	Device can do interiors

The high byte is 0.

LINECAPS
A bitmask that indicates the line capabilities of the device. The bits have the following meanings:

Bit	Meaning
0	Reserved
1	Device can do polyline
2	Reserved
3	Reserved
4	Device can do wide lines
5	Device can do styled lines
6	Device can do lines that are wide and styled
7	Device can do interiors

The high byte is 0.

POLYGONALCAPS

A bitmask that indicates the polygonal capabilities of the device. The bits have the following meanings:

Bit	Meaning
0	Device can do alternate fill polygon
1	Device can do rectangle
2	Device can do winding number fill polygon
3	Device can do scanline
4	Device can do wide borders
5	Device can do styled borders
6	Device can do borders that are wide and styled
7	Device can do interiors

The high byte is 0.

TEXTCAPS

A bitmask that indicates the text capabilities of the device. The bits have the following meanings:

Bit	Meaning
0	Device can do character output precision
1	Device can do stroke output precision
2	Device can do stroke clip precision
3	Device can do 90-degree character rotation
4	Device can do any character rotation
5	Device can do scaling independent of x and y
6	Device can do doubled character for scaling
7	Device can do integer multiples for scaling
8	Device can do any multiples for exact scaling
9	Device can do double-weight characters

Bit	Meaning
10	Device can do italicizing
11	Device can do underlining
12	Device can do strikeouts
13	Device can do raster fonts
14	Device can do vector fonts
15	Reserved; must be returned 0

Remarks Retrieves device-specific information about a given display device. The *nIndex* parameter specifies the type of information desired.

Return Value The value of the desired item.

See Also **::GetDeviceCaps**

CDC::GetMapMode

Syntax **int GetMapMode() const;**

Remarks Retrieves the current mapping mode.

See **SetMapMode** for a description of the mapping modes.

Return Value The mapping mode.

See Also **CDC::SetMapMode**, **::GetMapMode**

CDC::GetNearestColor

Syntax

DWORD GetNearestColor(DWORD *crColor* **) const;**

Parameters

crColor
Specifies the color to be matched.

Remarks

Returns the closest logical color to a specified logical color that the given device can represent.

Return Value

An RGB color value that names the solid color closest to the *crColor* value that the device can represent.

See Also

::GetNearestColor

CDC::GetPixel

Syntax

DWORD GetPixel(int *x*, **int** *y* **) const;**

DWORD GetPixel(POINT *point* **) const;**

Parameters

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**

CDC::GetPolyFillMode

Syntax **int GetPolyFillMode() const;**

Remarks Retrieves the current polygon-filling mode.

See the **SetPolyFillMode** member function later in this reference for a description of the polygon filling modes.

See Also **CDC::SetPolyFillMode**, **::GetPolyFillMode**

CDC::GetROP2

Syntax **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 **SetROP2**.

See Also **CDC::GetDeviceCaps**, **CDC::SetROP2**, **::GetROP2**

CDC::GetStretchBltMode

Syntax

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 using **StretchBlt**.

The **BLACKONWHITE** and **WHITEONBLACK** modes are typically used to preserve foreground pixels in monochrome bitmaps.

The **COLORONCOLOR** mode is typically used to preserve color in color bitmaps.

Return Value

The current bitmap-stretching mode. It can be **WHITEONBLACK**, **BLACKONWHITE**, or **COLORONCOLOR**.

See Also

CDC::StretchBlt, **CDC::SetStretchBltMode**, **::GetStretchBltMode**

CDC::GetTabbedTextExtent

Syntax

CSize GetTabbedTextExtent(const char FAR* *lpString*, **int** *nCount*,
  int *nTabPositions*, **LPINT** *lpnTabStopPositions* **) const;**

Parameters

lpString
  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 pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.

Remarks

Computes the width and height of a character string. 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.

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::TabbedTextOut**, **::GetTabbedTextExtent**

CDC::GetTextAlign

Syntax

UINT GetTextAlign() const;

Remarks

Retrieves the status of the text-alignment flags for the device context.

The text-alignment flags determine how **TextOut** and **ExtTextOut** 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.
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:

Value	Meaning
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

Syntax **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.

Return Value The amount of the intercharacter spacing.

See Also **CDC::SetTextCharacterExtra**, **::GetTextCharacterExtra**

CDC::GetTextColor

Syntax

DWORD GetTextColor() const;

Remarks

Retrieves the current text color. The text color is the foreground color of characters drawn by using the GDI text-output 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

Syntax

CSize GetTextExtent(const char FAR* *lpString*, **int** *nCount* **) const;**

Parameters

lpString
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.

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).

See Also

CDC::GetTabbedTextExtent, **::GetTextExtentEx**, **CDC::SetTextJustification**

CDC::GetTextFace

Syntax

int GetTextFace(int *nCount*, **const char FAR*** *lpFacename* **) const;**

Parameters

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.

lpFacename
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. It is 0 if an error occurs.

See Also

CDC::GetTextMetrics, **CDC::SetTextAlign**, **CDC::TextOut**, **::GetTextFace**

CDC::GetTextMetrics

Syntax

BOOL GetTextMetrics(LPTEXTMETRIC *lpMetrics* **) const;**

Parameters

lpMetrics
Points to the **TEXTMETRIC** structure that receives the metrics. @HO = Remarks

Retrieves the metrics for the current font.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::GetTextAlign, **CDC::GetTextExtent**, **CDC::GetTextFace**, **CDC::SetTextJustification**, **::GetTextMetrics**

CDC::GetViewportExt

Syntax **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 **CSize, ::GetViewportExt**

CDC::GetViewportOrg

Syntax **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 **CPoint, ::GetViewportOrg**

CDC::GetWindowExt

Syntax **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 **CSize, ::GetWindowExt**

CDC::GetWindowOrg

Syntax

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

CPoint, **::GetWindowOrg**

CDC::GrayString

Syntax

BOOL GrayString(CBrush* *pBrush*,
BOOL (FAR PASCAL EXPORT* *lpfnOutput* **)(HDC, DWORD, int),**
DWORD *lpData*, **int** *nCount*, **int** *x*, **int** *y*, **int** *nWidth*, **int** *nHeight* **);**

Parameters

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 **OutputProc** 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.

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 **TextOut** (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 Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see Chapter 12 in the *Class Libraries 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**.

Callback Function

BOOL FAR PASCAL __export OutputFunc(**HDC** *hDC*, **DWORD** *lpData*, **int** *nCount*);

OutputFunc is a placeholder for the application-supplied callback function name. The actual name must be exported as described in the "Remarks" above. The callback function (*OutputFunc*) must draw an image relative to the coordinates (0,0) rather than (*x*, *y*).

Parameter	**Description**
hDC	Identifies a memory device context with a bitmap of at least the width and height specified by *nWidth* and *nHeight* to **GrayString**, respectively.
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 nonzero to indicate success. Otherwise, it is 0.

Return Value

TRUE if the string is drawn, or **FALSE** if either the **TextOut** function or the application-supplied output function returned **FALSE**, or there was insufficient memory to create a memory bitmap for dimming.

See Also

::GetSysColor, **CDC::SetTextColor**, **CDC::TextOut**, **::GrayString**

CDC::IntersectClipRect

Syntax

int IntersectClipRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

int IntersectClipRect(LPRECT *lpRect* **);**

Parameters

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.

Return Value

The new clipping region's type. It can be any one of the following values:

Value	Meaning
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**

CDC::InvertRect

Syntax

void InvertRect(LPRECT *lpRect* **);**

Parameters

lpRect
Points to a **RECT** or **CRect** 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. 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

::InvertRect, **CRect**

CDC::InvertRgn

Syntax

BOOL InvertRgn(CRgn* *pRgn* **);**

Parameters

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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CRgn, **::InvertRgn**

CDC::LineTo

Syntax

BOOL LineTo(int *x*, **int** *y* **);**

BOOL LineTo(POINT *point* **);**

Parameters

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

TRUE if the line is drawn; otherwise **FALSE**.

See Also

CDC::MoveTo, **CDC::GetCurrentPosition**, **::LineTo**

CDC::LPtoDP

Syntax

void LPtoDP(LPPOINT *lpPoints*, **int** *nCount* = **1) const;**

void LPtoDP(LPRECT *lpRect* **) const;**

Parameters

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.

See Also

::LPtoDP

CDC::MoveTo

Syntax

CPoint MoveTo(int *x*, **int** *y* **);**

CPoint MoveTo(POINT *point* **);**

Parameters

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

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**

CDC::OffsetClipRgn

Syntax

int OffsetClipRgn(int *x*, **int** *y* **);**

int OffsetClipRgn(SIZE *size* **);**

Parameters

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 given device 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:

Value	Meaning
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

::OffsetClipRgn

CDC::OffsetViewportOrg

Syntax

CPoint OffsetViewportOrg(int *nWidth*, **int** *nHeight* **);**

Parameters

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 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**

CDC::OffsetWindowOrg

Syntax

CPoint OffsetWindowOrg(int *nWidth*, **int** *nHeight* **);**

Parameters

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 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**

CDC::PaintRgn

Syntax

BOOL PaintRgn(CRgn* *pRgn* **);**

Parameters

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* with the selected brush.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CBrush, **CDC::SelectObject**, **CDC::FillRgn**, **::PaintRgn**

CDC::PatBlt

Syntax

BOOL PatBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **DWORD** *dwRop* **);**

Parameters

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:

Value	Meaning
PATCOPY	Copies pattern to destination bitmap.
PATINVERT	Combines destination bitmap with pattern using the Boolean OR operator.
DSTINVERT	Inverts the destination bitmap.
BLACKNESS	Turns all output black.
WHITENESS	Turns all output white.

Remarks

Creates a bit pattern on the specified 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 values of *dwRop* for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, an operation code that refers to a source cannot be used. Not all devices support the **PatBlt** function.

For more information, see the **RC_BITBLT** capability under the **GetDeviceCaps** member function.

Return Value

TRUE if the bit pattern is drawn; otherwise **FALSE**.

See Also

CDC::GetDeviceCaps, **::PatBlt**

CDC::Pie

Syntax

BOOL Pie(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3*, **int** *x4*, **int** *y4* **);**

BOOL Pie(LPRECT *lpRect*, **POINT** *ptStart*, **POINT** *ptEnd* **);**

Parameters

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*).

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::Chord, **::Pie**

CDC::PlayMetaFile

Syntax

BOOL PlayMetaFile(HANDLE *hMF* **);**

Parameters

hMF
Identifies the metafile.

Remarks

Plays the contents of the specified metafile on the given device. The metafile can be played any number of times.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

::PlayMetaFile

CDC::Polygon

Syntax

BOOL Polygon(LPPOINT *lpPoints*, **int** *nCount* **);**

Parameters

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 given in the array.

Remarks

Draws a polygon consisting of two or more points (vertices) connected by lines. 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 **GetPolyFillMode** and **SetPolyFillMode**.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::GetPolyFillMode, **::PolyLine**, **CDC::PolyPolygon**, **CDC::SetPolyFillMode**, **::Polygon**

CDC::Polyline

Syntax

BOOL Polyline(LPPOINT *lpPoints*, **int** *nCount* **);**

Parameters

lpPoints
Points to an array of points 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 **LineTo**, the **Polyline** function neither uses nor updates the current position.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::LineTo, **CDC::Polygon**, **::PolyLine**

CDC::PolyPolygon

Syntax

BOOL PolyPolygon(LPPOINT *lpPoints*, **LPINT** *lpPolyCounts*, **int** *nCount* **);**

Parameters

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 they may overlap.

Each polygon specified in a call to the **PolyPolygon** function must be closed. Unlike polygons created by the **Polygon** member function of **CDC**, 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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::GetPolyFillMode, **CDC::Polygon**, **CDC::Polyline**, **CDC::SetPolyFillMode**, **::PolyPolygon**

CDC::PtVisible

Syntax

BOOL PtVisible(int *x*, **int** *y* **) const;**

BOOL PtVisible(POINT *point* **) const;**

Parameters

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

Specifies whether the given point is within the clipping region of the device context.

Return Value

TRUE if the specified point is within the clipping region; otherwise **FALSE**.

See Also

CPoint, **::PtVisible**

CDC::RealizePalette

Syntax

UINT RealizePalette();

Remarks

Takes entries in the logical palette currently selected into a device context and maps them 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 color display, or with colors displayed by other windows.

When a window has input focus and calls the function, Windows ensures that it will display all the colors it requests, up to the maximum number simultaneously available on the display, and 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

Specifies 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

CPalette, **::RealizePalette**

CDC::Rectangle

Syntax

BOOL Rectangle(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

BOOL Rectangle(LPRECT *lpRect* **);**

Parameters

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.

Return Value **TRUE** if the function is successful; otherwise **FALSE**.

See Also **::Rectangle, ::PolyLine, CDC::RoundRect**

CDC::RectVisible

Syntax **BOOL RectVisible(LPRECT** *lpRect* **) const;**

Parameters

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 current display context.

Return Value **TRUE** if some portion of the given rectangle lies within the clipping region; otherwise **FALSE**.

See Also **::RectVisible**

CDC::RestoreDC

Syntax

BOOL RestoreDC(int *nSavedDC* **);**

Parameters

nSavedDC
Specifies the device context to be restored. It must be a value returned by a previous **SaveDC** function call. If *nSavedDC* is –1, the most recent device context saved is restored.

Remarks

Restores the Windows device context to the previous state identified by *nSavedDC*. **RestoreDC** restores the device context by copying state information saved on the Windows internal context stack by earlier calls to the **SaveDC** member function.

The Windows internal context 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 any state information between the device context specified by *nSavedDC* and the top of the stack. The deleted information is lost.

Return Value

TRUE if the specified context was restored; otherwise **FALSE**.

See Also

CDC::SaveDC, **::RestoreDC**

CDC::RoundRect

Syntax

BOOL RoundRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3* **);**

BOOL RoundRect(LPRECT *lpRect*, **POINT** *point* **);**

Parameters

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.

Return Value **TRUE** if the function is successful; otherwise **FALSE**.

See Also **CDC::Rectangle**, **::RoundRect**

CDC::SaveDC

Syntax **int SaveDC() const;**

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

Specifies the saved device context. It is 0 if an error occurs. This return value is used on a subsequent call to **RestoreDC** to restore the device context state.

See Also

CDC::RestoreDC, **::SaveDC**

CDC::ScaleViewportExt

Syntax

CSize ScaleViewportExt(int *xNum*, **int** *xDenom*, **int** *yNum*, **int** *yDenom* **);**

Parameters

xNum
Specifies the amount by which to multiply the current x-extent.

xDenom
Specifies the amount by which to divide the current x-extent.

yNum
Specifies the amount by which to multiply the current y-extent.

yDenom
Specifies the amount by which to divide the current y-extent.

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 extent is 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

::ScaleViewportExt

CDC::ScaleWindowExt

Syntax

CSize ScaleWindowExt(int *xNum*, **int** *xDenom*, **int** *yNum*, **int** *yDenom* **);**

Parameters

xNum
Specifies the amount by which to multiply the current x-extent.

xDenom
Specifies the amount by which to divide the current x-extent.

yNum
Specifies the amount by which to multiply the current y-extent.

yDenom
Specifies the amount by which to divide the current y-extent.

Remarks

Modifies the window extents relative to the current values. The formulas are written as follows:

```
xNewVE = ( xOldVE * xNum ) / xDenom
yNewVE = ( yOldVE * yNum ) / yDenom
```

The new extent is 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

::ScaleWindowExt

CDC::ScrollDC

Syntax

BOOL ScrollDC(int *dx*, **int** *dy*, **LPRECT** *lpRectScroll*, **LPRECT** *lpRectClip*, **CRgn*** *pRgnUpdate*, **LPRECT** *lpRectUpdate* **);**

Parameters

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 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, upon return, contains the coordinates of the rectangle that bounds the scrolling update region. This is the largest rectangular area that requires repainting.

Remarks

Scrolls a rectangle of bits horizontally and vertically. The *lpRectScroll* parameter describes the rectangle to be scrolled, *dx* specifies the number of units to be scrolled horizontally, and *dy* specifies the number of units to be scrolled 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 region pointer to the region uncovered by the scrolling process (defined by the **ScrollDC** member function). 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

TRUE if scrolling is executed; otherwise **FALSE**.

See Also

CWnd::ScrollWindow, **::ScrollDC**, **CRgn**

CDC::SelectClipRgn

Syntax

int SelectClipRgn(CRgn* *pRgn* **);**

Parameters

pRgn
Identifies the region to be selected.

Remarks

Selects the given region as the current clipping region for the specified 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 graphics at lower resolutions than text output to increase speed, but at the expense of quality. These devices scale coordinates for graphics so that one graphics device point corresponds to two or four true device points. This scaling factor affects clipping. If a region will be used to clip graphics, its coordinates must be divided down by the scaling factor. If the region will be used to clip text, no scaling adjustment is needed. The scaling factor is determined by using the **GETSCALINGFACTOR** printer escape.

Return Value

The region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New clipping region has overlapping borders.
ERROR	Device context or region handle is not valid.
NULLREGION	New clipping region is empty.
SIMPLEREGION	New clipping region has no overlapping borders.

See Also

CDC::Escape, **CRgn**, **::SelectClipRgn**

CDC::SelectObject

Syntax

CGdiObject* SelectObject(CGdiObject* *pObject* **);**

CPen* SelectObject(CPen* *pPen* **);**

CBrush* SelectObject(CBrush* *pBrush* **);**

CFont* SelectObject(CFont* *pFont* **);**

CBitmap* SelectObject(CBitmap* *pBitmap* **);**

int SelectObject(CRgn* *pRgn* **);**

Parameters

pObject
Identifies the object to be selected.

Note that this general version of the **SelectObject** member function does not work for regions. To select regions, see the version of **SelectObject** in the next group that is specialized for regions.

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 current device context. Class **CDC** provides a general version of **SelectObject** and 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*.

Note that class **CMetaFileDC** overrides the **SelectObject** member function. The **CMetaFileDC** class is derived from class **CDC** specifically for use with metafiles. For information on object selection in metafiles, see the **CMetaFileDC** class.

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 specified device; if it is not, **SelectObject** returns an error.

Return Value

A pointer to a **CGdiObject** object or to an object of one of the classes derived from **CGdiObject**, such as **CPen**, depending on which version of the function used. The object pointed to is being replaced by the object specified by the function's parameter. It is **NULL** if there is an error.

For the version of the member function that takes a region parameter, *pRgn*, the return value is one of the following:

Value	Meaning
COMPLEXREGION	New clipping region has overlapping borders.
ERROR	Device context or region handle 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

Syntax

CPalette* SelectPalette(CPalette* *pPalette*, **BOOL** *bForceBackground* **);**

Parameters

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 **TRUE**, the selected palette is always a background palette, regardless of whether the window has input focus. If *bForceBackground* is **FALSE**, the logical palette is a foreground palette when the window has input focus.

Remarks

Selects the logical palette 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 object into more than one device context, the device contexts must all belong to the same physical device (such as a display or printer).

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

CPalette, **::SelectPalette**

CDC::SelectStockObject

Syntax

CGdiObject* SelectStockObject(int *nIndex* **);**

Parameters

nIndex
Specifies the kind of stock object desired. It can be one of the following values:

Value	Meaning
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.

Value	Meaning
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

Syntax **int SetAbortProc(short (FAR PASCAL EXPORT*** *lpfn* **)(HDC, short));**

Parameters

lpfn
A pointer to the abort function to install as the abort procedure. For more about this callback function, see 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 or the **STARTDOC** escape, which are equivalent. 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 new features of Microsoft C/C++ let you use an ordinary function as the function passed to **SetAbortProc**. 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 **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

short FAR PASCAL __export AFunction(**HDC**, **short**);

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 Foundation programs, you'll normally use the /GA and /GEs compiler options. The /Gw compiler option is not used with the Foundation classes. (If you do use **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 Foundation exceptions before returning to Windows, since exceptions cannot be thrown across callback boundaries. For more information about exceptions, see Chapter 12 in the *Class Libraries User's Guide*.

Callback Function

The callback function must use the Pascal calling convention, must be exported with **__export**, and must be declared **FAR**.

short FAR PASCAL __export AbortFunc(**HDC** *hPr*, **short** *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.

Parameter	Description
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 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 Print Manager by calling the **PeekMessage** or **GetMessage** 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.

Value	Meaning
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.

CDC::SetBkColor

Syntax

DWORD SetBkColor(DWORD *crColor* **);**

Parameters

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 represent 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::GetBkColor, **CDC::GetBkMode**, **CDC::SetBkMode**, **::SetBkColor**

CDC::SetBkMode

Syntax

int SetBkMode(int *nBkMode* **);**

Parameters

nBkMode
Specifies the background mode. This parameter can be either of the following values:

Value	Meaning
OPAQUE	Background is filled with the current background color before the text, hatched brush, or pen is drawn.
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. It can be either **OPAQUE** or **TRANSPARENT**.

See Also

CDC::GetBkColor, **CDC::GetBkMode**, **CDC::SetBkColor**, **::SetBkMode**

CDC::SetBrushOrg

Syntax

CPoint SetBrushOrg(int *x*, **int** *y* **);**

CPoint SetBrushOrg(POINT *point* **);**

Parameters

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-coordinate 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 an application selects into a device context.

Do not use **SetBrushOrg** with stock **CBrush** objects.

Return Value

The previous origin of the brush in device units (which are relative to the origin of the desktop window).

See Also

CDC::GetBrushOrg, **CDC::SelectObject**, **CGdiObject::UnrealizeObject**, **::SetBrushOrg**

CDC::SetMapMode

Syntax

int SetMapMode(int *nMapMode* **);**

Parameters

nMapMode
Specifies the new mapping mode. It can be any one of the following values:

Value	Meaning
MM_ANISOTROPIC	Logical units are mapped to arbitrary units with arbitrarily scaled axes. The **SetWindowExt** and **SetViewportExt** member functions of class **CDC** must be used to specify the desired units, orientation, and scaling.
MM_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive x is to the right; positive y is up.
MM_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive x is to the right; positive y is up.
MM_ISOTROPIC	Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the x-axis is equal to one unit along the y-axis. The **SetWindowExt** and **SetViewportExt** member functions of class **CDC** must be used 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 mapped to 0.01 inch. Positive x is to the right; positive y is up.
MM_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive x is to the right; positive y is up.
MM_TEXT	Each logical unit is mapped to one device pixel. Positive x is to the right; positive y is down.
MM_TWIPS	Each logical unit is mapped to one-twentieth of a printer's point (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 transform logical units into device units, and 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, whose size varies from device to device.

The **MM_HIENGLISH**, **MM_HIMETRIC**, **MM_LOENGLISH**, **MM_LOMETRIC**, and **MM_TWIPS** modes are useful for applications that need to 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

Syntax

DWORD SetMapperFlags(DWORD *dwFlag* **);**

Parameters

dwFlag
Specifies whether the font mapper attempts to match a font's aspect height and width to the device. When the first bit is set to 1, the mapper will only select fonts whose x-aspect and y-aspect exactly match those of the specified device.

Remarks

Alters the algorithm that the font mapper uses when it maps logical fonts to physical fonts. When the first bit of *dwFlag* is set to 1, the mapper will only select fonts whose x-aspect and y-aspect exactly match those of the specified device. If no fonts exist with a matching aspect height and width, GDI chooses an aspect height and width and selects fonts with aspect heights and widths that match the one chosen by GDI.

The remaining bits of *dwFlag* must be 0.

Return Value

The previous value of the font-mapper flag.

See Also

::SetMapperFlags

CDC::SetPixel

Syntax

DWORD SetPixel(int *x*, **int** *y*, **DWORD** *crColor* **);**

DWORD SetPixel(POINT *point*, **DWORD** *crColor* **);**

Parameters

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 is ignored.

Not all devices support the function. For more information, see the **RC_BITBLT** capability in the **GetDeviceCaps** member function.

Return Value

An RGB color value for the color that the point is actually painted. This value can be different than 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**

CDC::SetPolyFillMode

Syntax

int SetPolyFillMode(int *nPolyFillMode* **);**

Parameters

nPolyFillMode
Specifies the new filling mode. This value may be either **ALTERNATE** or **WINDING**.

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.

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. It is 0 if there is an error.

See Also

CDC::GetPolyFillMode, **CDC::PolyPolygon**, **::SetPolyFillMode**

CDC::SetROP2

Syntax

int SetROP2(int *nDrawMode* **);**

Parameters

nDrawMode
Specifies the new drawing mode. It can be any one of the following values:

Value	Meaning
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 display 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 display color.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the display.
R2_MERGENOTPEN	Pixel is a combination of the display color and the inverse of the pen color.

Value	Meaning
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the display and the inverse of the pen.
R2_MERGEPEN	Pixel is a combination of the pen color and the display color.
R2_NOTMERGEPEN	Pixel is the inverse of the **R2_MERGEPEN** color.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the display.
R2_NOTMASKPEN	Pixel is the inverse of the **R2_MASKPEN** color.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the display, but not in both.
R2_NOTXORPEN	Pixel is the inverse of the **R2_XORPEN** color.

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.

Drawing modes are binary raster-operation codes, representing all possible Boolean functions of two variables, using the binary operations 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 Software Development Kit documentation.

See Also

CDC::GetDeviceCaps, **CDC::GetROP2**, **::SetROP2**

CDC::SetStretchBltMode

Syntax

int SetStretchBltMode(int *nStretchMode* **);**

Parameters

nStretchMode

Specifies the new bitmap-stretching mode. It can be one of the following values:

Value	Meaning
BLACKONWHITE	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.
COLORONCOLOR	Deletes the eliminated lines. Information in the eliminated lines is not preserved.
WHITEONBLACK	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 **StretchBlt**. The bitmap-stretching mode defines how information is removed from bitmaps that are compressed by using the function.

The **BLACKONWHITE** and **WHITEONBLACK** modes are typically used to preserve foreground pixels in monochrome bitmaps. The **COLORONCOLOR** mode is typically used to preserve color in color bitmaps.

See Also

CDC::GetStretchBltMode, CDC::StretchBlt, ::SetStretchBltMode

CDC::SetTextAlign

Syntax

UINT SetTextAlign(UINT *nFlags* **);**

Parameters

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. Only one flag may be chosen from each category.

The first category affects text alignment in the x direction:

Value	Meaning
TA_CENTER	Specifies alignment of the point and the horizontal center of the bounding rectangle.
TA_LEFT	Specifies alignment of the point and the left side of the bounding rectangle. This is the default setting.
TA_RIGHT	Specifies alignment of the point and the right side of the bounding rectangle.

The second category affects text alignment in the y direction:

Value	Meaning
TA_BASELINE	Specifies alignment of the point and the baseline of the chosen font.
TA_BOTTOM	Specifies alignment of the point and the bottom of the bounding rectangle.
TA_TOP	Specifies alignment of the point and 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:

Value	Meaning
TA_NOUPDATECP	Specifies that the current position is not updated after each call to a text-output function. This is the default setting.
TA_UPDATECP	Specifies that the current position is updated after each call to a text-output function.

Remarks

Sets the text-alignment flags.

The functions **TextOut** and **ExtTextOut** 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. The low-order byte contains the horizontal alignment and the high-order byte contains the vertical alignment. The return value is 0 if there is an error.

See Also

CDC::ExtTextOut, **CDC::GetTextAlign**, **CDC::TabbedTextOut**, **CDC::TextOut**, **::SetTextAlign**

CDC::SetTextCharacterExtra

Syntax

int SetTextCharacterExtra(int *nCharExtra* **);**

Parameters

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.

Return Value

The amount of the previous intercharacter spacing.

See Also

CDC::GetTextCharacterExtra, **::SetTextCharacterExtra**

CDC::SetTextColor

Syntax

DWORD SetTextColor(DWORD *crColor* **);**

Parameters

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 **SetBkColor** and **SetBkMode**.

Return Value

An RGB value for the previous text color.

See Also

CDC::GetTextColor, **CDC::BitBlt**, **CDC::SetBkColor**, **CDC::SetBkMode**, **::SetTextColor**

CDC::SetTextJustification

Syntax

int SetTextJustification(int *nBreakExtra*, **int** *nBreakCount* **);**

Parameters

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 **GetTextMetrics** to retrieve a font's break character.

After calling the **SetTextJustification** member function, a call to **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 function **GetTextExtent** is typically used with **SetTextJustification**. **GetTextExtent** computes the width of a given line before justification. This width is compared to the width of the line after justification to determine how much space to add to the line.

The **SetTextJustification** function can be used to justify a line that contains multiple runs in different fonts. In this case, the line must be created piecemeal by justifying and writing each run separately.

Because rounding errors can occur during justification, the system keeps a running error term that defines the current error. When justifying 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 justified, 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

Syntax

CSize SetViewportExt(int *x*, **int** *y* **);**

CSize SetViewportExt(SIZE *size* **);**

Parameters

x
Specifies the x-extent of the viewport (in device units).

y
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_HIMETRIC
MM_LOENGLISH
MM_LOMETRIC
MM_TEXT
MM_TWIPS

When **MM_ISOTROPIC** mode is set, an application must call **SetWindowExt** 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 are both set to 0.

See Also

CDC::SetWindowExt, **::SetViewportExt**

CDC::SetViewportOrg

Syntax

CPoint SetViewportOrg(int *x*, **int** *y* **);**

CPoint SetViewportOrg(POINT *point* **);**

Parameters

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 **SetWindowOrg**. 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**

CDC::SetWindowExt

Syntax

CSize SetWindowExt(int *x*, **int** *y* **);**

CSize SetWindowExt(SIZE *size* **);**

Parameters

x
Specifies the x-extent (in logical units) of the window.

y
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::SetViewportExt, **::SetWindowExt**, **CSize**

CDC::SetWindowOrg

Syntax

CPoint SetWindowOrg(int *x*, **int** *y* **);**

CPoint SetWindowOrg(POINT *point* **);**

Parameters

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 specified 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

CDC::StartDoc

Syntax

int StartDoc(const char FAR* *pDocName* **);**

Parameters

pDocName
Pointer to a null-terminated string that specifies the name of the document. The document name is displayed in the Print Manager window. The maximum length of this string is 31 characters plus the terminating null character.

Remarks

Informs the device driver that a new print job is starting and that all subsequent **NEWFRAME** escape calls should be spooled under the same job until an **ENDDOC** escape call occurs. This ensures that documents longer than one page will not be interspersed with other jobs.

Return Value

The value –1 if there is an error such as insufficient memory or an invalid port specification occurs. Otherwise, a positive value.

See Also

CDC::Escape, **CDC::EndDoc**

CDC::StartPage

Syntax

int StartPage();

Remarks

Prepares 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.

See Also

CDC::Escape, **CDC::EndPage**

CDC::StretchBlt

Syntax

BOOL StretchBlt(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **CDC*** *pSrcDC*, **int** *xSrc*, **int** *ySrc*, **int** *nSrcWidth*, **int** *nSrcHeight*, **DWORD** *dwRop* **);**

Parameters

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 y-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:

Code	Description
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

Moves 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, 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 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 are used in the conversion.

If **StretchBlt** must convert a monochrome bitmap to color, it sets white bits (1) to background color and black bits (0) to 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 function.

For more information, see the **RC_BITBLT** capability in **GetDeviceCaps** member function.

Return Value

TRUE if the bitmap is drawn; otherwise **FALSE**.

See Also

CDC::BitBlt, **CDC::GetDeviceCaps**, **CDC::SetStretchBltMode**, **::StretchBlt**

CDC::TabbedTextOut

Syntax

CSize TabbedTextOut(int *x*, **int** *y*, **const char FAR*** *lpString*, **int** *nCount*,
int *nTabPositions*, **LPINT** *lpnTabStopPositions*, **int** *nTabOrigin* **);**

Parameters

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.

lpString
Points to the character string to draw. You can pass either a **const FAR** 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 device 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 a specified location, expanding tabs to the values specified in an 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 **SetTextAlign** with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores *x* and *y* on subsequent calls to **TabbedTextOut**, using the current position instead.

Return Value

The dimensions of the string (in logical units) as a **CSize**.

See Also

CDC::GetTabbedTextExtent, **CDC::SetTextAlign**, **CDC::TextOut**, **::TabbedTextOut**

CDC::TextOut

Syntax

BOOL TextOut(int *x*, **int** *y*, **const char FAR*** *lpString*, **int** *nCount* **);**

BOOL TextOut(int *x*, **int** *y*, **const CString&** *str* **);**

Parameters

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.

lpString
The pointer to the characters to write.

nCount
Specifies the number of characters to write.

str
A **CString** object or null-terminated string that contains the chararacters to write.

Remarks

Writes a character string at a 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 **SetTextAlign** with *nFlags* set to **TA_UPDATECP**. When this flag is set, Windows ignores *x* and *y* on subsequent calls to **TextOut**, using the current position instead.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::ExtTextOut, **CDC::GetTextExtent**, **CDC::SetTextAlign**, **CDC::SetTextColor**, **CDC::TabbedTextOut**, **::TextOut**

CDC::UpdateColors

Syntax

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.

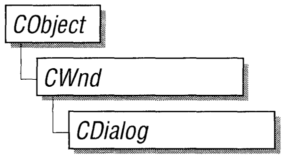
For more information on using color palettes, see the Windows Software Development Kit documentation.

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**

class CDialog : public CWnd



The **CDialog** class is an abstract class for displaying dialog boxes on the screen. To get a modeless dialog box, you must derive your own class from **CDialog**. To derive modal dialog boxes, use the **CModalDialog** class. The constructors for class **CDialog** are protected, so you must derive your own class.

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 modal dialog box requires the user to close the dialog box before the application continues.

You can create a modeless dialog in one step or two. To create it in one step, write the constructor so it calls the object's **Create** member function. To create it in two steps, don't include a call to **Create** in the constructor. Invoke the constructor for your dialog object, then call the object's **Create** member function.

A modeless dialog box receives messages from Windows like any other window. To process messages in your derived dialog-box class, provide message-handler member functions for the messages the dialog box can process.

Your message-handler member functions specify what happens when the user works with your dialog box. Typically, you'll override the **OnInitDialog** member function when you need to initialize controls (such as setting the initial text of an edit box).

You'll also override the **OnClose** member function of your derived dialog class to call **CWnd::DestroyWindow**. Instead of calling **DestroyWindow**, you can call the C++ **delete** operator on the **this** reference, which calls **DestroyWindow** for you.

Your derived dialog-box class can also add member variables to store data entered by the user or data for display to the user. You can add member functions to set or get these values. A modeless dialog box can also send messages to its parent window.

Create your dialog box from a dialog-box resource template, as in traditional Windows. The dialog-box resource specifies a template name or ID, a font to use, a set of controls, such as buttons and edit boxes, and the window styles that apply to the dialog box. To create a dialog box from a template, specify the template in your .RC file and compile it with a resource compiler. The resulting .RES file is sent to the linker, which incorporates the resource information with your executable program. Specify the name or ID of the template when you call the **Create** member function from your dialog-box constructor.

Instead of creating your dialog box from a compiled resource, you can build the resource yourself in memory, construct an object of your class derived from **CDialog**, and use the **CreateIndirect** member function to create the dialog box from the template in memory. The template constructed in memory uses a **DLGTEMPLATE** data structure, as described in the Windows Software Development Kit documentation.

If the dialog-box template (in a resource file or in memory) specifies the **WS_VISIBLE** style, the dialog-box window appears in its parent window. Otherwise, you must call the **ShowWindow** member function, which **CDialog** inherits from class **CWnd**.

After the call to **Create**, Windows sends a **WM_INITDIALOG** message to the dialog box. You can override the **OnInitDialog** member function to perform dialog-box initialization chores. For example, if your dialog box displays statistics about the current font, you can override **OnInitDialog** to set the current values of the static text controls in the dialog box to reflect the statistics.

Although the dialog-box template can specify the dialog-box font, you can also set the font on the fly. If the dialog-box template specifies the **DS_SETFONT** style, Windows sends a **WM_SETFONT** message to the dialog box before creating the controls. In response to this message, the application calls the **OnSetFont** member function. You can override that message-handler function to set the dialog-box font.

When the user terminates a modeless dialog box, call the **DestroyWindow** member function, which **CDialog** inherits from class **CWnd**, to remove the dialog window and destroy its data structures. You can call **DestroyWindow** from the **OnOK**, **OnCancel**, or **OnClose** member functions, which you can override from class **CWnd**. If you allocate any memory in the dialog object, override the **CDialog** destructor to dispose of the allocations.

See Also **CModalDialog**

Public Members

Operations

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 push button control for a dialog box to a specified push button.
GetDefID	Gets the ID of the default push button control for a dialog box.
EndDialog	Terminates a modal dialog box.

Overridables

OnInitDialog	Override to augment dialog-box initialization.
OnSetFont	Override to specify the font that a dialog-box control is to use when drawing text.

Protected Members

Construction/Destruction

CDialog	Constructs a **CDialog** object.

Initialization

Create	Initializes the **CDialog** object. Creates the modeless dialog and attaches it to the **CDialog** object.
CreateIndirect	Creates a modeless dialog box from a dialog-box template in memory.

Member Functions

CDialog::CDialog

Syntax

CDialog();

Remarks

The **CDialog** constructor is protected because you must derive your own dialog-box class to implement a dialog.

Construction of a modeless dialog is a two-step process. First invoke the constructor, then call either form of the **Create** member function. You can combine the steps by calling **Create** from within your constructor.

See Also

CDialog::Create, **CDialog::CreateIndirect**

CDialog::Create

Syntax

BOOL Create(const char FAR* *lpTemplateName*, **CWnd*** *pParentWnd* = **NULL);**

BOOL Create(UINT *nIDTemplate*, **CWnd*** *pParentWnd* = **NULL);**

Parameters

lpTemplateName
Contains a null-terminated string that is the name of a dialog-box resource template.

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, as shown in the following code:

```
if( pParentWnd == NULL )
    pParentWnd = AfxGetApp()->m_pMainWnd;
```

nIDTemplate
Contains the ID number of a dialog-box resource template.

Remarks

Call **Create** when you construct your dialog-box object. You can put the call to create inside the constructor or call it after the constructor executes.

Two forms of the **Create** member function are provided for access to the dialog template resource either by template name or by template ID number.

For either form, you also pass a pointer to the parent window object. If you don't, the dialog will be created with its parent window set to the main application window. Modeless dialogs can use this pointer to send messages to the parent if needed.

Before the dialog box is displayed, Windows sends the **WM_INITDIALOG** message to the dialog box. If the dialog box has the **DS_SETFONT** style, Windows also sends the **WM_SETFONT** message before the control windows are created. You can override the **OnInitDialog** and **OnSetFont** member functions to provide special handling of these messages.

The **Create** member function returns immediately after it creates the dialog box.

Use the **WS_VISIBLE** style in the dialog template if the dialog box should appear when the parent window is created. You can also specify other dialog styles in the template as explained in the Windows Software Development Kit documentation. These include styles that specify:

- The frame around the dialog box.
- Whether the dialog window is a pop-up or child window.
- Whether the dialog box has a border or a Control menu.
- How controls are to be grouped and the tabbing order between them.

Use the **CWnd::DestroyWindow** function to destroy a dialog box created by the **Create** function.

A dialog box can contain up to 255 controls.

Return Value

Both forms return **TRUE** if dialog creation and initialization was successful; otherwise **FALSE**.

See Also

CWnd::DestroyWindow, **CDialog::CreateIndirect**, **::CreateDialog**, **WM_SETFONT**, **WM_INITDIALOG**

CDialog::CreateIndirect

Syntax

Protected:
  BOOL CreateIndirect(const BYTE FAR* *lpDialogTemplate*,
    CWnd* *pParentWnd* **= NULL);**

Parameters

lpDialogTemplate
  Points to a global memory object that contains a dialog-box template used to create the dialog box. An application must build the dialog-box template according to the guidelines outlined in the description of the application-defined **DLGTEMPLATE** data structure.

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, as shown in the following code:

```
if( pParentWnd == NULL )
    pParentWnd = AfxGetApp()->m_pMainWnd;
```

Remarks

Creates a modeless dialog box from a dialog-box template in memory. **CreateIndirect** is protected.

Before the dialog box is displayed, Windows sends the **WM_INITDIALOG** message to the dialog box. If the dialog box has the **DS_SETFONT** style, Windows also sends the **WM_SETFONT** message before the control windows are created. You can override the **OnInitDialog** and **OnSetFont** member functions to provide special handling of these messages.

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 in the parent window upon creation. You can also specify other dialog styles in the template as explained in the Windows Software Development Kit documentation. These include styles that specify:

- The frame around the dialog box.
- Whether the dialog window is a pop-up or child window.
- Whether the dialog box has a border or a Control menu.
- How controls are to be grouped and the tabbing order between them.

Use the **CWnd::DestroyWindow** function to destroy a dialog box created by the **CreateIndirect** function.

A dialog box can contain up to 255 controls.

Return Value

TRUE if the dialog was created and initialized successfully; otherwise **FALSE**.

See Also

CDialog::Create, **CWnd::DestroyWindow**, **WM_INITDIALOG**, **WM_SETFONT**

CDialog::EndDialog

Syntax

void EndDialog(int *nResult* **);**

Parameters

nResult
Contains the value to be returned from the dialog box to the member function that created it.

Remarks

Used for modal dialog boxes. Modeless dialogs do not use this member function.

The **EndDialog** member function terminates a modal dialog box and returns the given result to the function that created the dialog box. The **EndDialog** function is required to complete processing whenever a modal dialog box is created and may not be used for any other purpose.

The dialog function can call **EndDialog** at any time, even during the processing of the **WM_INITDIALOG** message in **OnInitDialog**. If called during processing of the **WM_INITDIALOG** message, the dialog box is terminated 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 standard Foundation dialog-box function (**AfxDlgProc**) ends. The **EndDialog** function returns to the dialog-box function, so it must return control to Windows.

See Also

CModalDialog, **CDialog::Create**, **CDialog::CreateIndirect**, **WM_INITDIALOG**

CDialog::GetDefID

Syntax

DWORD GetDefID();

Remarks

Call the **GetDefID** member function to get the ID of the default push button control for a dialog box. This is usually an OK button.

Return Value

A 32-bit value (**DWORD**). If the default push button has an ID value, the high-order word contains **DC_HASDEFID** and the low-order word contains the ID value. If the default push button does not have an ID value, the return value is 0.

See Also

CDialog::SetDefID

CDialog::GotoDlgCtrl

Syntax

void GotoDlgCtrl(CWnd* *pWndCtrl* **);**

Parameters

pWndCtrl
Identifies the window (control) that is to receive the focus.

Remarks

Moves the focus to the specified control in the dialog.

To get a pointer to the control (child window) to pass as *pWndCtrl*, call the **GetDlgItem** member function, which returns the pointer as a pointer to a **CWnd** object. This pointer can then be cast to its specific type. **GetDlgItem** is inherited from class **CWnd**.

See Also

CWnd::GetDlgItem, **CDialog::PrevDlgCtrl**, **CDialog::NextDlgCtrl**

CDialog::IsDialogMessage

Syntax

BOOL IsDialogMessage(LPMSG *lpMsg* **);**

Parameters

lpMsg
Points to an **MSG** structure that contains the message to be checked.

The **MSG** structure looks like this:

```
typedef struct tagMSG {
   HWND    hwnd;
   WORD    message;
   WORD    wParam;
   LONG    lParam;
   DWORD   time;
   POINT   pt;
} MSG;
```

Remarks

Determines whether the given message is intended for the modeless dialog box and automatically processes the message if it is. 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.

A message processed by **IsDialogMessage** must not be passed to the **TranslateMessage** or **DispatchMessage** Windows functions. The message has already been processed.

IsDialogMessage sends the **WM_GETDLGCODE** message to determine which keys to process.

Return Value

Specifies whether the given message has been processed. It is **TRUE** if the message has been processed; otherwise **FALSE**. If the return is **FALSE**, call the **PreTranslateMessage** member function of the base class to process the message. The code looks like this in an override of the **CDialog PreTranslateMessage** member function:

```
CMyDlg::PreTranslateMessage( msg )
{
    if( IsDialogMessage( msg )
        return TRUE;
    else
        return CDialog::PreTranslateMessage( msg );
}
```

See Also

::DispatchMessage, ::TranslateMessage, ::GetMessage, CWnd::PreTranslateMessage, WM_GETDLGCODE

CDialog::MapDialogRect

Syntax

void MapDialogRect(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to a **RECT** structure that contains the dialog-box coordinates to be converted.

Remarks

Converts the dialog-box units of a rectangle to screen units. Dialog-box units are stated in terms of the current dialog base unit derived from the average width and height of characters in the font used for dialog-box text.

Typically, dialog boxes use the system font, but a different font may be specified by using the **DS_SETFONT** style in the resource-definition file. 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 Windows function **GetDialogBaseUnits** returns the dialog-box base units in pixels.

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, **CDialog::Create**, **CDialog::CreateIndirect**, **WM_SETFONT**

CDialog::NextDlgCtrl

Syntax

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::OnInitDialog

Syntax

virtual BOOL OnInitDialog();

Remarks

Called in response to the **WM_INITDIALOG** message. This message is sent during the **Create** or **CreateIndirect** call, which occurs immediately before the dialog box is displayed. Override it if you need to perform special processing when the dialog box is initialized.

The **OnInitDialog** function is called via the standard global dialog-box procedure, **AfxDlgProc**, common to all Microsoft Foundation Class Library dialogs, rather than through your message map, so you do not need a message-map entry for this member function.

Return Value

Returns **TRUE** by default, indicating successful dialog initialization.

See Also

CDialog::Create, **CDialog::CreateIndirect**, **WM_INITDIALOG**

CDialog::OnSetFont

Syntax

virtual void OnSetFont(CFont* *pFont* **);**

Parameters

pFont
Specifies a pointer to the font. If this parameter is **NULL**, the control will draw text using the default system font.

Remarks

Specifies which font a dialog-box control is to use when drawing text.

The dialog-box font normally gets set in the .RC resource file as part of the dialog-box resource template. If you want to set it instead at run time, specify the **DS_SETFONT** style in your dialog-box template. With that style set, Windows sends a **WM_SETFONT** message to the dialog box before creating the controls. The **OnSetFont** member function is then called automatically via the standard dialog-box procedure.

The application should call the **CGdiObject::DeleteObject** function to delete the font when it is no longer needed, such as after the control is destroyed. Also call **CGdiObject::DeleteObject** to delete the old font before you set the new one.

The size of the control is not changed as a result of receiving this message. To prevent Windows from clipping text that does not fit within the boundaries of the control, the application should correct the size of the control window before changing the font.

For more information about dialog resource templates, see the Windows Software Development Kit documentation.

See Also **WM_SETFONT**

CDialog::PrevDlgCtrl

Syntax **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 dialog box.

See Also **CDialog::NextDlgCtrl**, **CDialog::GotoDlgCtrl**

CDialog::SetDefID

Syntax **void SetDefID(UINT** *nID* **);**

Parameters

nID
Specifies the ID of the push button control that will become the default.

Remarks Changes the default push button control for a dialog box.

See Also **CDialog::GetDefID**

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.

Preconditions

Before you create your own **CDumpContext** object, you must create a **CFile** object that serves as the dump destination.

Comments

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 in hexadecimal form. If you need a custom dump format for a derived object, override **CObject::Dump**. Most Microsoft Foundation classes have a **Dump** member function defined.

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 class **Dump** functions are defined only in the Debug version. Use **#ifdef _DEBUG / #endif** statements to bracket your diagnostic code, including your custom **Dump** member functions.

#define _DEBUG

#include <afx.h>

See Also

CFile, **CObject**

Public Members

Construction/Destruction

CDumpContext	Constructs a **CDumpContext** object.

Basic I/O

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

GetDepth	Gets an integer corresponding to the depth of the dump.
SetDepth	Sets the depth of the dump.

Member Functions

CDumpContext::CDumpContext

Syntax

CDumpContext(CFile* *pFile* **)**
throw(CMemoryException, CFileException);

Parameters

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**.

Do not write to the underlying **CFile** while the dump context is active; otherwise you will interfere with the dump.

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

Syntax

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

Syntax **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**.

Example See the example for **SetDepth**.

See Also **SetDepth**

CDumpContext::HexDump

Syntax **void HexDump(const char*** *pszLine*, **BYTE*** *pby*, **int** *nBytes*, **int** *nWidth* **)**
throw(CFileException);

Parameters

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

Syntax

void SetDepth(int *nNewDepth* **);**

Parameters

nNewDepth
The new depth value.

Remarks

Sets the depth for the dump. If you are dumping primitive types or simple **CObject**s that contain 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.

Example

```
afxDump.SetDepth( 1 );  // specifies deep dump
ASSERT( afxDump.GetDepth() == 1 );
```

See Also

CObject::Dump

Operators

CDumpContext::operator <<

Syntax

CDumpContext& operator <<(const CObject* *pOb* **)**
throw(CFileException);

CDumpContext& operator <<(const CObject& *ob* **)**
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.

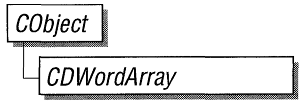
Return Value

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 double words.

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 CWordArray::GetAt( int <nIndex> ) const;
```

CDWordArray incorporates the **IMPLEMENT_SERIAL** macro to support serialization and dumping of its elements. If an array of double words 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>

Public Members

Construction/Destruction

CDWordArray	Constructs an empty array for double words.
~CDWordArray	Destroys a **CDWordArray** object.

Bounds

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

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

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 double word within the array.

Growing the Array

SetAtGrow	Sets the value for a given index, growing the array if necessary.
Add	Adds an element to the end of the array.

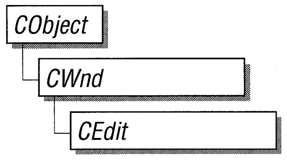
Insertion/Removal

InsertAt	Inserts an element at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

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 create an edit control in two steps. 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.

If you want to handle the Windows notification messages sent by a **CEdit** object to its parent (usually a class derived from **CDialog**), add the following message-map entries and message-handler member functions to the parent class:

ON_COMMAND
ON_EN_SETFOCUS
ON_EN_KILLFOCUS
ON_EN_MAXTEXT
ON_EN_CHANGE
ON_EN_UPDATE
ON_EN_HSCROLL
ON_EN_VSCROLL

If you create a **CEdit** object within a dialog box, the **CEdit** 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.

See Also **CWnd**, **CButton**, **CComboBox**, **CListBox**, **CScrollBar**, **CStatic**, **CModalDialog**, **CDialog**

Public Members

Construction/Destruction

CEdit	Constructs a **CEdit** control object.

Initialization

Create	Creates the Windows edit control and attaches it to the **CEdit** object.

Multiple-Line Operations

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 updating the control.
SetTabStops	Sets the tab stops in a multiple-line edit control.

General Operations

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.
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.
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

Syntax **BOOL CanUndo() const;**

Return Value **TRUE** if the last edit operation can be undone by a call to the **Undo** member function; **FALSE** if it cannot be undone.

See Also **CEdit::Undo**, **EM_CANUNDO**

CEdit::CEdit

Syntax **CEdit();**

Remarks Constructs a **CEdit** object.

See Also **CEdit::Create**

CEdit::Clear

Syntax **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

Syntax

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

Syntax

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

Parameters

dwStyle
Specifies the edit control's style.

rect
Specifies the edit control's size and position.

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.

To handle Windows notification messages sent from a **CEdit** object to its parent, add the following message-map entries to the parent class message map:

ON_COMMAND
ON_EN_SETFOCUS
ON_EN_KILLFOCUS
ON_EN_MAXTEXT
ON_EN_CHANGE
ON_EN_UPDATE
ON_EN_HSCROLL
ON_EN_VSCROLL

Apply the following window styles to an edit control:

Style	Application
WS_CHILD	Always.
WS_VISIBLE	Usually.
WS_DIABLED	Rarely.
WS_GROUP	To group controls.
WS_TABSTOP	To include edit control in the tabbing order.

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

Use any combination of the following edit control styles for *dwStyle*:

Style	Meaning
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.

Style	Meaning
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.

Style	Meaning
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.

Return Value

Create returns **TRUE** if initialization is successful; **FALSE** if unsuccessful.

See Also

CEdit::CEdit

CEdit::Cut

Syntax

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

Syntax

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

Syntax

BOOL FmtLines(BOOL *bAddEOL* **);**

Parameters

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

TRUE if any formatting occurs; otherwise **FALSE**.

See Also

CEdit::GetHandle, **CWnd::GetWindowText**, **EM_FMTLINES**

CEdit::GetHandle

Syntax

HANDLE 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 only processed 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 that holds 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

Syntax

int GetLine(int *nIndex*, **LPSTR** *lpBuffer* **) const;**

int GetLine(int *nIndex*, **LPSTR** *lpBuffer*, **int** *nMaxLength* **) const;**

Parameters

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.

lpBuffer
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 *lpBuffer* before making the call to Windows.

Remarks

Retrieves a line of text from an edit control and places it in *lpBuffer*. 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

Syntax

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

Syntax

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

TRUE if the edit-control contents have been modified; **FALSE** if they have remained unchanged.

See Also

CEdit::SetModify, **EM_GETMODIFY**

CEdit::GetRect

Syntax

void GetRect(LPRECT *lpRect* **) const;**

Parameters

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

Syntax

DWORD GetSel() const;

Remarks

Gets the starting and ending character positions of the current selection (if any) in an edit control.

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.

See Also

CEdit::SetSel, **EM_GETSEL**

CEdit::LimitText

Syntax

void LimitText(int *nChars* **= 0);**

Parameters

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 65,535 bytes. This is the default behavior.

Remarks

Limits the length of the text that the user may enter into an edit control.

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 by the **SetWindowText** member function in **CWnd**.

See Also

CWnd::SetWindowText, EM_LIMITTEXT

CEdit::LineFromChar

Syntax

int LineFromChar(int *nIndex* **= –1) const;**

Parameters

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, i.e., 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

Syntax

int LineIndex(int *nLine* **= –1) const;**

Parameters

nLine
Contains the index value for the desired character in the text of the edit control, or contains –1. If *nLine* is –1, it specifies the current line, i.e., 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 then the number of lines in the edit control.

See Also

CEdit::LineFromChar, **EM_LINEINDEX**

CEdit::LineLength

Syntax

int LineLength(int *nLine* **= –1) const;**

Parameters

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

Syntax

void LineScroll(int *nLines*, **int** *nChars* **= 0);**

Parameters

nLines
Specifies the number of lines to scroll vertically.

nChars
Specifies the number of character positions to scroll horizontally.

Remarks

Scrolls the text of a multiple-line edit control.

This member function is only processed by multiple-line edit controls.

The edit control will 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 will be adjusted such 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.

A call to this member function will be ignored if the multiple-line edit control was not created with the **ES_LEFT** style.

See Also

EM_LINESCROLL

CEdit::Paste

Syntax **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

Syntax **void ReplaceSel(const char FAR*** *lpNewText* **);**

Parameters

lpNewText
Points to a null-terminated string containing the replacement text.

Remarks Replaces the current selection in an edit control with the text specified by *lpNewText.*

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

Syntax **void SetHandle(HANDLE** *hBuffer* **);**

Parameters

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 only processed 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 Windows **LocalFree** function.

SetHandle clears the undo buffer (the **CanUndo** member function then returns **FALSE**) and the internal modification flag (the **GetModify** member function then returns **FALSE**). The edit-control window will be redrawn.

You may 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

Syntax

void SetModify(BOOL *bModified* **= TRUE);**

Parameters

bModified
A value of **TRUE** indicates that the text has been modified, and a value of **FALSE** indicates it is unmodified. By default, the modify flag is set.

Remarks

Sets or clears the modification flag for an edit control. The modification 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

Syntax

void SetPasswordChar(char *ch* **);**

Parameters

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 enters text. When a password character is set, that character is displayed for each character the user types in.

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

EM_SETPASSWORDCHAR

CEdit::SetRect

Syntax

void SetRect(LPRECT *lpRect* **);**

Parameters

lpRect
Points to the **RECT** or **CRect** that specifies the new dimensions of the formatting rectangle.

Remarks

Sets the dimensions of a rectangle using the specified coordinates. This call is only processed by a multiline edit control.

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 then 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.

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

Syntax **void SetRectNP(LPRECT** *lpRect* **);**

Parameters

lpRect
Points to a **RECT** or **CRect** 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. 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 then 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.

SetRectNP is identical to the **SetRect** member function except that the edit-control window is not redrawn.

This member is only processed by multiple-line edit controls.

See Also **CRect::CRect**, **CRect::CopyRect**, **CRect::operator =**, **CRect::SetRectEmpty**, **CEdit::GetRect**, **CEdit::SetRect**, **EM_SETRECT**

CEdit::SetSel

Syntax

void SetSel(DWORD *dwSelection* **);**

void SetSel(int *nStartChar*, **int** *nEndChar* **);**

Parameters

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.

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. The edit control does not display the selection set by this member function as it does when the user makes a selection.

See Also

CEdit::GetSel, **CEdit::ReplaceSel**, **EM_SETSEL**

CEdit::SetTabStops

Syntax

BOOL SetTabStops(int *nTabStops*, **LPINT** *rgTabStops* **);**

void SetTabStops();

BOOL SetTabStops(int *cxEachStop* **);**

Parameters

nTabStops
Specifies the number of tab stops contained in *rgTabStops*. If this parameter is greater then 1, then *rgTabStops* points to an array of tab stops.

rgTabStops
Points to an array of unsigned integers specifying the tab stops in dialog units. If *nTabStops* is 1, this parameter points to an unsigned integer containing the distance between all tab stops (in dialog units).

cxEachStop
Specifies that tab stops are to be set at every *cxEachStop* dialog units.

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.

If *nTabStops* is 0, *rgTabStops* is ignored and default tab stops are set at every 32 dialog units.

This member function is only processed by multiple-line edit controls.

SetTabStops does not automatically redraw the edit window. If the application is changing the tab stops for text already in the edit control, it needs to call **CWnd::InvalidateRect** to redraw the edit window.

Return Value

TRUE if the tabs were set; otherwise **FALSE**.

See Also

::GetDialogBaseUnits, **CWnd::InvalidateRect**, **EM_SETTABSTOPS**

CEdit::Undo

Syntax

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**, and remove the text again with a second call to **Undo** as long as there is no intervening edit operation.

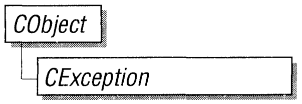
Return Value

For a single-line edit control, the return value is always **TRUE**. For a multiple-line edit control, the return value is **TRUE** if the undo operation is successful, or **FALSE** if the undo operation fails.

See Also

CEdit::CanUndo, **EM_UNDO**

class CException : public CObject



CException is the base class for all exceptions in the Microsoft Foundation Class Library. The derived classes 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

These exceptions are intended to be used with the **THROW**, **THROW_LAST**, **TRY**, **CATCH**, **AND_CATCH**, and **END_CATCH** macros. For more information on exception processing, see Chapter 5, "Exception Processing." Also see the cookbook in the *Class Libraries User's Guide.*

#include <afx.h>

Comments

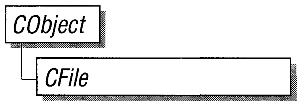
Use the derived classes to catch specific exceptions. Use **CException** if you need to catch all types of exceptions (and then use **CObject::IsKindOf** to differentiate among classes derived from **CException**.) All derived **CException** classes use the **IMPLEMENT_DYNAMIC** macro.

CException objects are deleted automatically. Do not delete them yourself.

Derivation

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.

class CFile : public CObject



CFile is the base class for Foundation class files. 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 archiving of 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 I/O stream 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 file status without opening the file.

#include <afx.h>

See Also **CStdioFile**, **CMemFile**

Public Members

Data Members

m_hFile	Usually contains the operating-system file handle.

Construction/Destruction

CFile	Constructs a **CFile** object from a path or file handle.
~CFile	Destroys the object, closing the file if it is open.
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

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

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

LockRange	Locks a range of bytes in a file.
UnlockRange	Unlocks a range of bytes in a file.

Status

GetPosition	Gets the current file pointer.
GetStatus	Obtains the status of this open file.

Static

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

Syntax

CFile();

CFile(int *hFile* **);**

CFile(const char* *pszFileName*, **UINT** *wOpenFlags*)
throw(CFileException);

Parameters

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.

wOpenFlags
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.

Value	Meaning
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.

Value	Meaning
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 the cookbook in the *Class Libraries 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.

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::~CFile

Syntax

virtual ~CFile();

Remarks

This destructor closes the operating-system file if it is open.

CFile::Close

Syntax

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

Syntax

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**.

Example

```
extern CFile* cfile1;
CFile* cfile2 = cfile1->Duplicate();
```

CFile::Flush

Syntax

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

Syntax

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

Syntax

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 double word.

Example

```
extern CFile cfile;
DWORD dwPosition = cfile.GetPosition();
```

CFile::GetStatus

Syntax

virtual BOOL GetStatus(CFileStatus& *rStatus* **) const;**

static BOOL GetStatus(const char* *pszFileName*,
CFileStatus& *rStatus* **) const;**

Parameters

rStatus
A reference to a user-supplied **CFileStatus** structure that will receive the status information. The **CFileStatus** structure has the following fields:

Field	Meaning
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 in bytes of the file, as reported by the MS-DOS command **dir**

Field	Meaning
BYTE m_attribute	The MS-DOS attribute byte of the file
char m_szFullName[_MAX_PATH]	The absolute filename (**_MAX_PATH** is defined in **stdlib.h**)

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

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 Attribut {
   normal =    0x00,
   readOnly =  0x01,
   hidden =    0x02,
   system =    0x04,
   volume =    0x08,
   directory = 0x10,
   archive =   0x20
   };
```

Return Value

TRUE if no error, in which case *rStatus* is valid; otherwise **FALSE**. **FALSE** indicates that the file does not exist.

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
   }
```

See Also **CFile::SetStatus**

CFile::LockRange

Syntax

virtual void LockRange(DWORD *dwPos*, **DWORD** *dwCount* **)**
throw(CFileException);

Parameters

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 **UnockRange** 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 load SHARE.EXE; otherwise, the function throws a **CFileException** object.

Note This function is not available for the **CMemFile**-derived class.

Example

```
extern DWORD dwPos;
extern DWORD dwCount;
extern CFile cfile;
cfile.LockRange( dwPos, dwCount );
```

See Also

CFile::UnlockRange

CFile::Open

Syntax

virtual BOOL Open(const char* *pszFileName***, UINT** *wStyleFlags***,**
CFileException* *pError* **= NULL);**

Parameters

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.

wStyleFlags
A **WORD** 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. Thus there is no possibility of a memory leak due to a failing constructor.

Return Value

TRUE if the open was successful; otherwise **FALSE**. The *pError* parameter is only meaningful if **FALSE** is returned.

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
   }
```

See Also **CFile::CFile**, **CFile::Close**

CFile::Read

Syntax

virtual UINT Read(void FAR* *lpBuf*, **UINT** *nCount* **)**
throw(CFileException);

Parameters

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 For all **CFile** classes, the return value may be less than *nCount* if the end of file was reached.

Example

```
extern CFile cfile;
char pbuf[100];
WORD nBytesRead = cfile.Read( pbuf, 100 );
```

See Also **CFile::Write**

CFile::Remove

Syntax

static void Remove(const char* *pszFileName* **)**
throw(CFileException);

Parameters

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 file is connected to an existing open file 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 << " not found\\n";
  #endif
}
END_CATCH
```

CFile::Rename

Syntax

static void Rename(const char* *pszOldName*, **const char*** *pszNewName* **)**
throw(CFileException);

Parameters

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

Syntax

virtual LONG Seek(LONG *lOff*, **UINT** *wFrom* **)**
throw(CFileException);

Parameters

lOff
Number of bytes to move the pointer.

wFrom
Pointer movement mode. Must be one of the following:

Value	Meaning
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 *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

Syntax

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

Syntax

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.

Example

```
extern CFile cfile;
DWORD dwActual = cfile.SeekToEnd();
```

See Also

CFile::GetLength, **CFile::Seek**, **CFile::SeekToBegin**

CFile::SetLength

Syntax

virtual void SetLength(const DWORD *dwNewLen* **)**
throw(CFileException);

Parameters

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

Syntax

static void SetStatus(const char* *pszFileName*, **const CFileStatus&** *status* **)**
throw(CFileException);

Parameters

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 **GetStatus** 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 **GetStatus** 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 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.

Example

```
char* pFileName = "test.dat";
extern BYTE newAttribute;
CFileStatus status;
CFile::GetStatus( pFileName, status );
status.m_attribute = newAttribute;
CFile::SetStatus( pFileName, status );
```

See Also **CFile::GetStatus**

CFile::UnlockRange

Syntax **virtual void UnlockRange(const DWORD** *dwPos*, **const DWORD** *dwCount* **) throw(CFileException);**

Parameters

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 **LockRange** 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.

Example

```
extern DWORD dwPos;
extern DWORD dwCount;
extern CFile cfile;
cfile.UnlockRange( dwPos, dwCount );
```

See Also **CFile::LockRange**

CFile::Write

Syntax

virtual void Write(const void FAR* *lpBuf*, **UINT** *nCount* **)**
throw(CFileException);

Parameters

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.

Example

```
extern CFile cfile;
char pbuf[100];
cfile.Write( pbuf, 100 );
```

See Also

CFile::Read, **CStdioFile::WriteString**

Data Members

CFile::m_hFile

Syntax

UINT m_hFile;

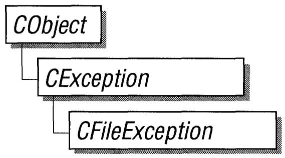
Remarks

Contains the operating-system file handle for an open file. It contains **CFile::m_hFileNull** (an operating-system-independent empty file indicator) if the handle has not been assigned.

Usage 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 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.



#include <afx.h>

See Also **CFile**, Chapter 5, "Exception Processing"

Comments **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.

Public Members

Data Members

m_cause	Contains portable code corresponding to the exception cause.
m_lOsError	Contains the related operating-system error number.

Construction/Destruction

CFileException	Constructs a **CFileException** object.

Code Conversion

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

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

Syntax

CFileException(int *cause* **= CFileException::none, LONG** *lOsError* **= –1);**

Parameters

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* provides.

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

Syntax

static int ErrnoToException(int *nErrno* **);**

Parameters

nErrno
An integer error code as defined in the run-time include file **errno.h**.

Remarks

This static function 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.

Example

```
#include <errno.h>
ASSERT( CFileException::ErrnoToException( EACCES ) ==
                    CFileException::accessDenied );
```

See Also **CFileException::OsErrorToException**

CFileException::OsErrorToException

Syntax **static int OsErrorToException(LONG** *lOsError* **);**

Parameters

lOsError
An operating-system-specific error code.

Remarks This static function 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.

Example

```
ASSERT( CFileException::OsErrorToException( 5 ) ==
                    CFileException::accessDenied );
```

See Also **CFileException::ErrnoToException**

CFileException::ThrowErrno

Syntax **static void ThrowErrno(int** *nErrno* **);**

Parameters

nErrno
An integer error code as defined in the run-time include file **errno.h**.

Remarks This static function constructs a **CFileException** object corresponding to a given *nErrno* value, then throws the exception.

Example

```
#include <errno.h>
CFileException::ThrowErrno( EACCES );  // "access denied"
```

See Also **CFileException::ThrowOsError**

CFileException::ThrowOsError

Syntax **static void ThrowOsError(LONG** *lOsError* **);**

Parameters

lOsError
An operating-system-specific error code.

Remarks This static function 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**.

Example

```
CFileException::ThrowOsError( 5 );  // "access denied"
```

See Also **CFileException::ThrowErrno**

Data Members

CFileException::m_cause

Syntax

int m_cause;

Remarks

Contains values defined by a **CFileException** enumerated type. The enumerators are:

Value	Meaning
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	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 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");
}
END_CATCH
```

CFileException::m_lOsError

Syntax

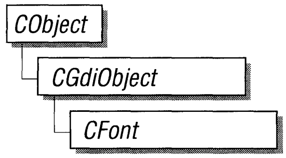
LONG m_lOsError;

Remarks

Contains the operating-system error code for this exception. See your operating-system technical manual for a listing of error codes.

class CFont : public CGdiObject

The **CFont** class encapsulates a Windows graphical design 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.



Public Members

Construction/Destruction

CFont	Constructs a **CFont** object.

Initialization

CreateFontIndirect	Initializes a **CFont** object with the characteristics given in a **LOGFONT** structure.
CreateFont	Initializes a **CFont** with the specified characteristics.

Operations

FromHandle	Returns a pointer to a **CFont** object when given a Windows **HFONT**.

Member Functions

CFont::CFont

Syntax

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

Syntax

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*, **const char FAR*** *lpFacename* **);**

Parameters

nHeight
Specifies the desired height (in logical units) of the font. The font height can be specified in three ways.

Height	Result
Greater than 0	Height is transformed into device units and matched against the cell height of the available fonts.
Equal to 0	A reasonable default size is used.
Less than 0	Height is transformed into device units and the absolute value is matched against the character height of the available fonts.

For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size. Then, if there is no such font, it looks for the smallest font available.

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.

nWeight
Specifies the font weight (in inked pixels per 1000). Although *nWeight* can be any integer value from 0 to 1000, the common values are as follows:

Value	Meaning
400	Normal
700	Bold

These values are approximate; the actual appearance depends on the typeface. If *nWeight* is 0, 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 values are predefined:

ANSI_CHARSET
OEM_CHARSET
SYMBOL_CHARSET

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.

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_STRING_PRECIS
OUT_STROKE_PRECIS

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_STROKE_PRECIS

nQuality
Specifies the font's output quality, which defines how carefully GDI must attempt to match the logical-font attributes to those of an actual physical font. It can be one of the following values:

Value	Meaning
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
FIXED_PITCH
VARIABLE_PITCH

The four high-order bits of the member specify the font family and can be one of the following values:

Value	Meaning
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. 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.

lpFacename
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 **EnumFonts** function can be used to enumerate all currently available fonts. If *lpFacename* is **NULL**, GDI uses a default 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 GDI's pool of physical fonts.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CFont::CreateFontIndirect, **::CreateFont**, **::EnumFonts**

CFont::CreateFontIndirect

Syntax

BOOL CreateFontIndirect(LPLOGFONT *lpLogFont* **);**

Parameters

lpLogFont
Points to a **LOGFONT** structure that defines the characteristics of the logical font.

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;
```

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** functions, 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.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CFont::CreateFont, CDC::SelectObject, CMetaFileDC::SelectObject, ::CreateFontIndirect

CFont::FromHandle

Syntax

static CFont* FromHandle(HFONT *hFont* **);**

Parameters

hFont
An **HFONT** handle to a Windows font.

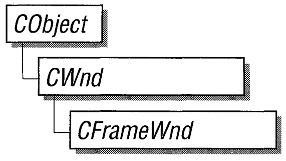
Remarks

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 CFrameWnd : public CWnd



The **CFrameWnd** class provides the functionality of a Windows (SDI) overlapped or pop-up frame 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.

You create a frame window in two steps. First, call the constructor **CFrameWnd** to construct the **CFrameWnd** object, then call the **Create** member function to create the frame window and attach it to the **CFrameWnd** object.

Construction can be a one-step process in a derived class. Write a constructor for the derived class and call **Create** from within the constructor.

When the user terminates your frame window, destroy the **CFrameWnd** object or call the **DestroyWindow** member function, which **CFrameWnd** inherits from class **CWnd**, to remove the window and destroy its data structures. If you allocate any memory in the **CFrameWnd** object, override the **CFrameWnd** destructor to dispose of the allocations.

See Also **CWnd**, **CMDIFrameWnd**, **CMDIChildWnd**

Public Members

Data Members

rectDefault	Pass this static **CRect** as a parameter when creating a **CFrameWnd** object to allow Windows to choose the window's size and position.

Construction/Destruction

CFrameWnd	Constructs a **CFrameWnd** object.
~CFrameWnd	Destroys a **CFrameWnd** object and the frame window, frees the accelerator table, and posts an application quit message.

Initialization

Create	Creates and initializes the Windows frame window associated with the **CFrameWnd** object.
LoadAccelTable	Loads an accelerator table.

Operations

GetChildFrame	An overridable member function that simply returns **this**. A derived class should provide this function for access to the active child.
GetParentFrame	An overridable member function that simply returns **this**. A derived class should provide this function for access to the parent frame.

Protected Members

m_hAccelTable	Contains the command accelerator table for this frame window.

Member Functions

CFrameWnd::CFrameWnd

Syntax

CFrameWnd();

Remarks

Constructs a **CFrameWnd** object. The frame window is not created until the **Create** member function is called.

See Also

CFrameWnd::Create, **CFrameWnd::~CFrameWnd**

CFrameWnd::~CFrameWnd

Syntax

virtual ~CFrameWnd();

Remarks

Destroys a **CFrameWnd** object and the frame window, frees the accelerator table if loaded, and posts a **WM_QUIT** message to terminate the application.

See Also

CFrameWnd::Create, **CFrameWnd::CFrameWnd**

CFrameWnd::Create

Syntax

BOOL Create(const char FAR* *lpClassName*,
  const char FAR* *lpWindowName*,
  DWORD *dwStyle* = **WS_OVERLAPPEDWINDOW,**
  const RECT& *rect* = **rectDefault, const CWnd*** *pParentWnd* = **NULL,**
  const char FAR* *lpMenuName* = **NULL);**

Parameters

lpClassName
  Points to a null-terminated character string that names the Windows class (a **WNDCLASS struct**). The class name can be any name registered with the **afxRegisterWndClass** function or any of the predefined control-class names. If **NULL**, uses the predefined default **CFrameWnd** attributes.

lpWindowName
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. See the **CreateEx** member function in the **CWnd** class for a full list of window styles.

rect
Specifies the size and position of the window. The **rectDefault** value allows Windows to specify the size and position of the new **CFrameWnd** object.

pParentWnd
Specifies the parent window of this frame window. This parameter should be **NULL** for top-level frame windows.

lpMenuName
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**.

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.

Return Value

TRUE if initialization is successful; otherwise **FALSE**.

See Also

CFrameWnd::CFrameWnd, **CFrameWnd::~CFrameWnd**, **CWnd::CreateEx**

CFrameWnd::GetChildFrame

Syntax

virtual CFrameWnd* GetChildFrame();

Remarks

An overridable member function that simply returns **this**. A derived class should override this function to provide access to the active child.

Return Value

Returns **this**.

See Also

CMDIFrameWnd::GetChildFrame

CFrameWnd::GetParentFrame

Syntax **virtual CFrameWnd* GetParentFrame();**

Remarks An overridable member function that simply returns **this**. A derived class should override this function to provide access to the parent frame.

Return Value Returns **this**.

See Also **CMDIChildWnd::GetParentFrame**

CFrameWnd::LoadAccelTable

Syntax **BOOL LoadAccelTable(const char FAR*** *lpAccelTableName* **);**

Parameters *lpAccelTableName*
Identifies the name of the accelerator resource. Use **MAKEINTRESOURCE** if the resource is identified with an integer ID.

Remarks Loads the specified accelerator table. The member function stores the table handle in **m_hAccelTable**. Only one table may be loaded at a time.

Return Value **TRUE** if the accelerator table was successfully loaded; otherwise **FALSE**.

See Also **::LoadAccelerators**

Data Members

CFrameWnd::m_hAccelTable

Remarks

A protected variable of type **HANDLE**, this data member specifies the command accelerator table for the frame window. If the frame window doesn't have an accelerator table, this data member's value is **NULL**. The default **PreTranslateMessage** function uses the accelerator table to translate the appropriate keys into commands.

CFrameWnd::rectDefault

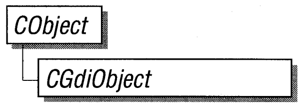
Remarks

Pass this static **CRect** as a parameter when creating a window to allow Windows to choose the window's size and position.

See Also

CW_USEDEFAULT

class CGdiObject : public CObject



The **CGdiObject** class provides a base class for various kinds of Windows graphical design 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**.

See Also **CBitmap**, **CBrush**, **CFont**, **CPalette**, **CPen**, **CRgn**

Public Members

Data Members

m_hObject	A **HANDLE** containing the **HBITMAP**, **HPALETTE**, **HRGN**, **HBRUSH**, **HPEN**, or **HFONT** attached to this object.

Construction/Destruction

CGdiObject	Constructs a **CGdiObject** object.
~CGdiObject	Destroys a **CGdiObject** object.

Operations

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

Syntax

BOOL Attach(HANDLE *hObject* **);**

Parameters

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

TRUE if attachment is successful; otherwise **FALSE**.

See Also

CGdiObject::Detach

CGdiObject::CGdiObject

Syntax

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::~CGdiObject

Syntax

virtual ~CGdiObject();

Remarks

Destructor for the **CGdiObject** class. Calls **DeleteObject** to delete the attached Windows GDI object and then deallocates the **CGdiObject** object. If you don't want to delete the attached Windows GDI object when deleting a **CGdiObject** object, call **Detach** before deleting the **CGdiObject** object.

See Also

CGdiObject::Detach, CGdiObject::DeleteObject

CGdiObject::CreateStockObject

Syntax

BOOL CreateStockObject(int *nIndex* **);**

Parameters

nIndex
A constant specifying the type of stock object desired. It can be one of the following values:

Value	Meaning
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.

Value	Meaning
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 Returns **TRUE** if the function is successful; otherwise **FALSE**.

See Also **CPen::CPen**, **CBrush::CBrush**, **CFont::CFont**, **CPalette::CPalette**

CGdiObject::DeleteObject

Syntax **void 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 C++ **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

Syntax **static void 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

Syntax **HANDLE 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, or **NULL** if no GDI object is attached.

See Also **CGdiObject::Attach**

CGdiObject::FromHandle

Syntax

static CGdiObject* FromHandle(HANDLE *hObject* **);**

Parameters

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

Syntax

int GetObject(int *nCount*, **LPSTR** *lpObject* **) const;**

Parameters

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, or 0 if an error occurs.

See Also **CBitmap::GetBitmapBits**, **CPalette::GetPaletteEntries**

CGdiObject::GetSafeHandle

Syntax **HANDLE 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, or **NULL** if no object is attached.

CGdiObject::UnrealizeObject

Syntax

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

TRUE if successful; otherwise **FALSE**.

See Also

CDC::RealizePalette, **CDC::SetBrushOrg**

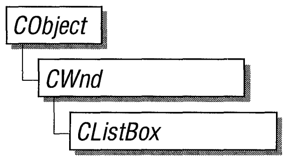
Data Members

CGdiObject::m_hObject

Remarks

A **HANDLE** containing the **HBITMAP**, **HRGN**, **HBRUSH**, **HPEN**, **HPALETTE**, or **HFONT** attached to this object.

class 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 only select 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 create a list-box control in two steps. First, 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 the Windows notification messages sent by a **CListBox** object to its parent (usually a class derived from **CDialog** or **CModalDialog**), add the appropriate message-map entries and message-handler member functions to the parent class to handle the messages you want to process. Potential message-map entries are:

ON_COMMAND
ON_LBN_DBLCLK
ON_LBN_ERRSPACE
ON_LBN_KILLFOCUS
ON_LBN_SELCHANGE
ON_LBN_SETFOCUS

If you create a **CListBox** object within a dialog box (through a dialog resource), the **CListBox** is automatically destroyed when the user closes the dialog box.

If you create a **CListBox** object within a window, you may also need to destroy it. 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.

See Also **CWnd, CButton, CComboBox, CEdit, CScrollBar, CStatic, CModalDialog, CDialog**

Public Members

Construction/Destruction

CListBox	Constructs a **CListBox** object.

Initialization

Create	Creates the Windows list box and attaches it to the **CListBox** object.

General Operations

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.
SetItemData	Sets the 32-bit value associated with the list-box item.
GetItemRect	Returns the bounding rectangle of the list-box item as it is currently displayed.
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

GetCurSel	Returns the zero-based index of the currently selected string in a list box.
SetCurSel	Selects a list-box string.

Multiple Selection Operations

SetSel	Selects or deselects a list box item 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

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.
SelectString	Searches for and selects a string in a single-selection list box.

Member Functions

CListBox::AddString

Syntax

int AddString(const char FAR* *lpItem* **);**

Parameters

lpItem
Points to the null-terminated string that is to be added.

Remarks

Adds 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.

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, LB_ADDSTRING

CListBox::CListBox

Syntax

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::Create

Syntax

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

Parameters

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 resource 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.

To handle Windows notification messages sent from a **CListBox** object to its parent, add any of the following message-map entries that you want to process to the parent class message map:

ON_COMMAND
ON_LBN_DBLCLK
ON_LBN_ERRSPACE
ON_LBN_KILLFOCUS
ON_LBN_SELCHANGE
ON_LBN_SETFOCUS

Apply the following window styles to a list-box control:

Style	Application
WS_CHILD	Always.
WS_VISIBLE	Usually.
WS_DISABLED	Rarely.
WS_VSCROLL	Adds a vertical scroll bar.
WS_HSCROLL	Adds a horizontal scroll bar.
WS_GROUP	To group controls.
WS_TABSTOP	To allow tabbing to this control.

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

Use any combination of the following list-box styles for *dwStyle*:

Style	Meaning
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.

Style	Meaning
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.)

Style	Meaning
LBS_WANTKEYBOARDINPUT	The owner of the list box receives **WM_VKEYTOITEM** or **WM_CHARTOITEM** messages whenever the user presses a key when the list box has input focus. This allows an application to perform special processing on the keyboard input.

Return Value **TRUE** if successful; otherwise **FALSE**.

See Also **CListBox::CListBox**

CListBox::DeleteString

Syntax **int DeleteString(UINT** *nIndex* **);**

Parameters *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 the *nIndex* specifies an index greater then the number of items in the list.

See Also **LB_DELETESTRING**, **CListBox::AddString**, **CListBox::InsertString**

CListBox::Dir

Syntax

int Dir(UINT *attr*, **const char FAR*** *lpWildCard* **);**

Parameters

attr

Can be any combination of the **CFile::GetStatus enum** flags, 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 *lpWildCard* specifies a directory.
0x0020	File has been archived.
0x4000	Include all drives that match the name specified by *lpWildCard*.
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.

lpWildCard

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::FindString

Syntax

int FindString(int *nStartAfter*, **const char FAR*** *lpItem* **) const;**

Parameters

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.

lpItem
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::GetCount

Syntax

int GetCount() const;

Remarks

Retrieves the number of items in a list box.

The returned count is one greater then 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

Syntax **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

Syntax **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

Syntax

DWORD GetItemData(int *nIndex* **) const;**

Parameters

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.

The 32-bit value was the *lpItem* 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::InsertString**, **CListBox::SetItemData**, **LB_GETITEMDATA**

CListBox::GetItemRect

Syntax

int GetItemRect(int *nIndex*, **LPRECT** *lpRect* **) const;**

Parameters

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

Syntax

int GetSel(int *nIndex* **) const;**

Parameters

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

Syntax

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

Syntax

int GetSelItems(int *nMaxItems*, **LPINT** *rgIndex* **) const;**

Parameters

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

Syntax

int GetText(int *nIndex*, **char FAR*** *lpBuffer* **) const;**

void GetText(int *nIndex*, **CString&** *rString* **) const;**

Parameters

nIndex
Specifies the zero-based index of the string to be retrieved.

lpBuffer
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

Syntax

int GetTextLen(int *nIndex* **) const;**

Parameters

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

Syntax

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

Syntax

int InsertString(int *nIndex*, **const char FAR*** *lpItem* **);**

Parameters

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.

lpItem
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::ResetContent

Syntax

void ResetContent();

Remarks

Removes all items from a list box.

See Also

LB_RESETCONTENT

CListBox::SelectString

Syntax

int SelectString(int *nStartAfter*, **const char FAR*** *lpItem* **);**

Parameters

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.

lpItem
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 *lpItem.*

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

Syntax **int SelItemRange(BOOL** *bSelect,* **int** *nFirstItem,* **int** *nLastItem* **);**

Parameters

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::SetColumnWidth

Syntax

void SetColumnWidth(int *cxWidth* **);**

Parameters

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

Syntax

int SetCurSel(int *nSelect* **);**

Parameters

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

Syntax

void SetHorizontalExtent(int *cxExtent* **);**

Parameters

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.

See Also

CListBox::GetHorizontalExtent, LB_SETHORIZONTALEXTENT

CListBox::SetItemData

Syntax

int SetItemData(int *nIndex*, **DWORD** *dwItemData* **);**

Parameters

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::GetItemData, LB_SETITEMDATA

CListBox::SetSel

Syntax

int SetSel(int *nIndex*, **BOOL** *bSelect* = **TRUE);**

Parameters

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

Syntax

BOOL SetTabStops(int *nTabStops*, **LPINT** *rgTabStops* **);**

void SetTabStops();

BOOL SetTabStops(int *cxEachStop* **);**

Parameters

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 one 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.

cxEachStop
Tab stops are set at every *cxEachStop* dialog units.

Remarks

Sets the tab-stop positions in a list box.

If **SetTabStops()** is called, *nTabStops* is 0, *rgTabStops* is **NULL**, and the default tab stop is 2 dialog units.

If *nTabStops* is 1, tab stops will be separated by the distance specified by *rgTabStops*.

If *rgTabStops* points to more than a single value, then 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

TRUE if all the tabs were set; otherwise **FALSE**.

See Also

LB_SETTABSTOPS, **::GetDialogBaseUnits**

CListBox::SetTopIndex

Syntax

int SetTopIndex(int *nIndex* **);**

Parameters

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.

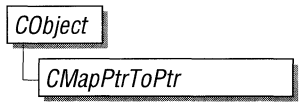
Return Value

LB_ERR if an error occurs.

See Also

CListBox::GetTopIndex, **LB_SETTOPINDEX**

class CMapPtrToPtr : public CObject



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>

Public Members

Construction/Destruction

CMapPtrToPtr	Constructs a collection that maps **void** pointers to **void** pointers.
~CMapPtrToPtr	Destroys a **CMapPtrToPtr** object.

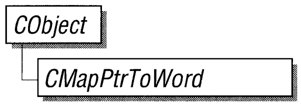
Operations

Lookup	Looks up a **void** pointer, based on the **void** pointer key. The pointer value is used for the key comparison, not the entity it points to.
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

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>

Public Members

Construction/Destruction

CMapPtrToWord	Constructs a collection that maps **void** pointers to 16-bit words.
~CMapPtrToWord	Destroys a **CMapPtrToWord** object.

Operations

Lookup	Returns a **WORD**, using a **void** pointer as a key. The pointer value is used for the key comparison, not the entity it points to.
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

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapStringToOb : public CObject

CObject
CMapStringToOb

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. If a map 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 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.

#include <afxcoll.h>

See Also

CMapPtrToPtr, **CMapPtrToWord**, **CMapStringToPtr**, **CMapStringToString**, **CMapWordToOb**, **CMapWordToPtr**

Derivation

Map class derivation is similar to list derivation. See the tutorial in the *Microsoft Class Library User's Guide* for an illustration of the derivation of a special-purpose list class.

Public Members

Construction/Destruction

CMapStringToOb	Constructs a collection that maps **CString** values to **CObject** pointers.
~CMapStringToOb	Destroys a **CMapStringToOb** object.

Operations

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

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

Member Functions

CMapStringToOb::CMapStringToOb

Syntax

CMapStringToOb(int *nBlockSize* **= 10);**

Parameters

nBlockSize
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::~CMapStringToOb

Syntax

~CMapStringToOb();

Remarks

Destroys a **CMapStringToOb** object, including all **CString** key objects contained in the map, but does not destroy the **CObject** objects.

CMapStringToOb::GetCount

Syntax

int GetCount() const;

Return Value

The number of elements in this map.

Example

```
CMapStringToOb map;

map.SetAt( "Bart", new CAge( 13 ) );
map.SetAt( "Homer", new CAge( 36 ) );
ASSERT( map.GetCount() == 2 );
```

See Also

CMapStringToOb::IsEmpty

CMapStringToOb::GetNextAssoc

Syntax

void GetNextAssoc(POSITION& *rNextPosition*, **CString&** *rKey*, **CObject*&** *rValue* **) const;**

Parameters

rNextPosition
A reference to a **POSITION** value returned by a previous **GetNextAssoc** or **GetStartPosition** call.

rKey
The returned key of the retrieved element (a string).

rValue
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**.

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
```

See Also **CMapStringToOb::GetStartPosition**

CMapStringToOb::GetStartPosition

Syntax **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 function **GetNextAssoc**.

CMapStringToOb::IsEmpty

Syntax

BOOL IsEmpty() const;

Return Value

TRUE if this map contains no elements; otherwise **FALSE**.

Example

See the example for **RemoveAll**.

See Also

CMapStringToOb::GetCount

CMapStringToOb::Lookup

Syntax

BOOL Lookup(const char* *key*, **CObject*&** *rValue* **) const;**

Parameters

key
The string key that identifies the element to be looked up.

rValue
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**.

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?
```

See Also

CMapStringToOb::operator [ ]

CMapStringToOb::RemoveAll

Syntax **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.

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
```

See Also **CMapStringToOb::RemoveKey**

CMapStringToOb::RemoveKey

Syntax **BOOL RemoveKey(const char*** *key* **);**

Parameters

key
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**.

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: age 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
```

See Also **CMapStringToOb::RemoveAll**

CMapStringToOb::SetAt

Syntax

void SetAt(const char* *key*, **CObject*** *newValue* **)**
throw(CMemoryException);

Parameters

key
The string that is the key of the new element.

newValue
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.

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
```

See Also **CMapStringToOb::Lookup**, **CMapStringToOb::operator [ ]**

Operators

CMapStringToOb::operator [ ]

Syntax

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.

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
```

See Also

CMapStringToOb::SetAt, **CMapStringToOb::Lookup**

class CMapStringToPtr : public CObject

CObject
CMapStringToPtr

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>

Public Members

Construction/Destruction

CMapStringToPtr	Constructs a collection that maps **CString** objects to **void** pointers.
~CMapStringToPtr	Destroys a **CMapStringToPtr** object.

Operations

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

GetCount	Returns the number of elements in this map.
IsEmpty	Tests for the empty-map condition (no elements).

class CMapStringToString : public CObject

CObject
CMapStringToString

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. If a map 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 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>

Public Members

Construction/Destruction

CMapStringToString	Constructs a collection that maps **CString** objects to **CString** objects.
~CMapStringToString	Destroys a **CMapStringToString** object.

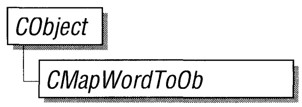
Operations

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

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. If a map 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 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>

Public Members

Construction/Destruction

CMapWordToOb	Constructs a collection that maps words to **CObject** pointers.
~CMapWordToOb	Destroys a **CMapWordToOb** object.

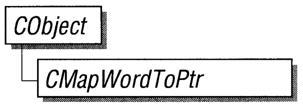
Operations

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

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>

Public Members

Construction/Destruction

CMapWordToPtr	Constructs a collection that maps words to **void** pointers.
~CMapWordToPtr	Destroys a **CMapWordToPtr** object.

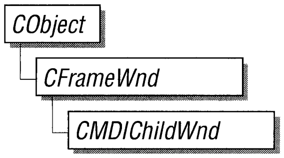
Operations

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

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 data and methods to manipulate the window. An MDI child window looks much like a typical application window, except that the MDI child window lacks a menu. The menu on the main application window applies to MDI child windows.

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.

You create an MDI child window in two steps. First, call the constructor **CMDIChildWnd** to construct the **CMDIChildWnd** object, then call the **Create** member function to create the MDI child window and attach it to the **CMDIChildWnd** object.

Construction can be a one-step process in a derived class. Write a constructor for the derived class and call **Create** from within the constructor.

When the user terminates your MDI child window, destroy the **CMDIChildWnd** object, or call the **DestroyWindow** member function, which **CMDIChildWnd** inherits from class **CWnd**, to remove the **CMDIChildWnd** and destroy its data structures. If you allocate any memory in the **CMDIChildWnd** object, override the **CMDIChildWnd** destructor to dispose of the allocations.

See Also **CWnd**, **CFrameWnd**, **CMDIFrameWnd**

Public Members

Construction/Destruction

CMDIChildWnd	Called when a **CMDIChildWnd** object is constructed.

Initialization

Create	Creates the Windows MDI child window associated with the **CMDIChildWnd** object.

Operations

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.
GetParentFrame	Returns the parent MDI frame.

Protected Members

Data Members

m_pMDIFrameWnd	Points to the parent **CMDIFrameWnd** of this **CMDIChildWnd**.

Member Functions

CMDIChildWnd::CMDIChildWnd

Syntax

CMDIChildWnd();

Remarks

Called when a **CMDIChildWnd** object is constructed. The Windows MDI child window is not created until the **Create** member function is called.

See Also

CMDIChildWnd::Create

CMDIChildWnd::Create

Syntax

BOOL Create(const char FAR* *lpClassName*,
  const char FAR* *lpWindowName*, **DWORD** *dwStyle* = **0,**
  const RECT& *rect* = **rectDefault, CMDIFrameWnd*** *pParentWnd* = **NULL**);

Parameters

lpClassName
  Points to a null-terminated character string that names the Windows class (a **WNDCLASS struct**). The class name can be any name registered with the **afxRegisterWndClass** function or any of the predefined control-class names. Should be **NULL** for a standard **CMDIChildWnd**.

lpWindowName
  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. See the **CreateEx** 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.

Remarks

Creates the Windows MDI child window and attaches it to the **CMDIChildWnd** object.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CMDIChildWnd::CMDIChildWnd

CMDIChildWnd::GetParentFrame

Syntax

virtual CFrameWnd* GetParentFrame();

Remarks

Returns the parent MDI frame. The actual parent, as returned by the **GetParent** member function, is a special, invisible window of type **MDICLIENT**.

CMDIChildWnd::MDIActivate

Syntax

void MDIActivate();

Remarks

An MDI child window is activated 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

Syntax	**void MDIDestroy();**
Remarks	Destroys this MDI child window.
	Removes the title of the child window from the frame window and deactivates the child window.
See Also	**WM_MDIDESTROY, CMDIChildWnd::Create**

CMDIChildWnd::MDIMaximize

Syntax	**void MDIMaximize();**
Remarks	Maximizes this 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 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

Syntax	**void MDIRestore();**
Remarks	Restores this MDI child window from maximized or minimized size.
See Also	**CMDIChildWnd::MDIMaximize, WM_MDIRESTORE**

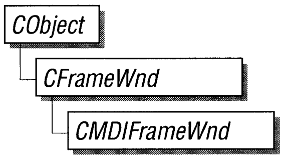
Data Members

CMDIChildWnd::m_pMDIFrameWnd

Remarks Points to the parent **CMDIFrameWnd** of this **CMDIChildWnd**.

See Also **CMDIChildWnd:GetParentFrame**, **CMDIFrameWnd**

class CMDIFrameWnd : public CFrameWnd



The **CMDIFrameWnd** class provides the functionality of a Windows multiple document interface (MDI) frame window, and also provides 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 create an MDI frame window in two steps. First, call the constructor **CMDIFrameWnd** to construct the **CMDIFrameWnd** object, then call the **Create** member function to create the MDI frame window and attach it to the **CMDIFrameWnd** object.

Construction can be a one-step process in a derived class. Write a constructor for the derived class and call **Create** from within the constructor.

When the user terminates your MDI frame window, destroy the **CMDIFrameWnd** object, or call the **DestroyWindow** member function, which **CMDIFrameWnd** inherits from class **CWnd**, to remove the **CMDIFrameWnd** and destroy its data structures. If you allocate any memory in the **CMDIFrameWnd** object, override the **CMDIFrameWnd** destructor to dispose of the allocations.

See Also **CWnd**, **CFrameWnd**, **CMDIChildWnd**

Public Members

Construction/Destruction

CMDIFrameWnd	Constructs a **CMDIFrameWnd**.

Initialization

Create	Creates and attaches the Windows MDI frame window associated with the **CMDIFrameWnd** object.
CreateClient	Fills out a **CLIENTCREATESTRUCT** and creates a Windows **MDICLIENT** window for this **CMDIFrameWnd**. Called by the **OnCreate** member function.

Operations

MDIActivate	Activates a different MDI child window.
MDICascade	Arranges all child windows in a cascade format.
MDIGetActive	Retrieves the current active MDI child window, along with a flag indicating whether the child is maximized or not.
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.
GetChildFrame	Returns the active MDI child.

Data Members

m_hWndMDIClient	The **HWND** for the MDI client window.

Member Functions

CMDIFrameWnd::CMDIFrameWnd

Syntax

CMDIFrameWnd();

Remarks

Constructs a **CMDIFrameWnd** object. The Windows MDI frame window is not created and attached to the **CMDIFrameWnd** object until the **Create** member function is called.

See Also

CMDIFrameWnd::Create

CMDIFrameWnd::Create

Syntax

BOOL Create(const char FAR* *lpClassName*,
  const char FAR* *lpWindowName*, **DWORD** *dwStyle*, **const RECT&** *rect*,
  const CWnd* *pParentWnd*, **const char FAR*** *lpMenuName* **);**

Parameters

lpClassName
  Points to a null-terminated character string that names the Windows class (a **WNDCLASS struct**). The class name can be any name registered with the **afxRegisterWndClass** function or any of the predefined control-class names. If **NULL**, uses the predefined default **CMDIFrameWnd** attributes.

lpWindowName
  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. See the **CreateEx** 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 **CMDIFrameWnd**.

pParentWnd
Specifies the parent window of this MDI frame window. This parameter should be **NULL** for top-level MDI frame windows.

lpMenuName
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**.

Remarks Creates the Windows MDI frame window and attaches it to the **CMDIFrameWnd** object.

Return Value **TRUE** if successful; otherwise **FALSE**.

See Also **CMDIFrameWnd::CMDIFrameWnd**

CMDIFrameWnd::CreateClient

Syntax **virtual BOOL CreateClient(LPCREATESTRUCT** *lpCreateStruct*, **CMenu*** *pWindowMenu* **);**

Parameters

lpCreateStruct
Pointer to a **CREATESTRUCT** structure.

pWindowMenu
Pointer to the window pop-up menu.

Remarks Creates the MDI client window that manages the **CMDIChildWnds** and fills out a **CLIENTCREATESTRUCT**.

CreateClient should be called if you override the **OnCreate** member function.

Return Value **TRUE** if successful; otherwise **FALSE**.

See Also **CMDIFrameWnd::CMDIFrameWnd**, **CMDIFrameWnd::Create**

CMDIFrameWnd::GetChildFrame

Syntax

virtual CFrameWnd* GetChildFrame();

Return Value

Returns the active MDI child if successful. Returns a pointer to the **CMDIFrameWnd** if unsuccessful.

See Also

CMDIFrameWnd::MDIGetActive

CMDIFrameWnd::MDIActivate

Syntax

void MDIActivate(CWnd* *pWndActivate* **);**

Parameters

pWndActivate
Points to the MDI child window to be activated.

Remarks

Activates a different MDI child window. The **WM_MDIACTIVATE** message is sent to both the child window being activated and the child window being deactivated.

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

Syntax **void MDICascade();**

Remarks Arranges all the MDI child windows in a cascade format.

See Also **CMDIFrameWnd::MDIIconArrange**, **CMDIFrameWnd::MDITile**, **WM_MDICASCADE**

CMDIFrameWnd::MDIGetActive

Syntax **CMDIChildWnd* MDIGetActive(BOOL*** *pbMaximized* **= NULL) const;**

Parameters

pbMaximized
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

Syntax **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

Syntax

void MDIMaximize(CWnd* *pWnd* **);**

Parameters

pWnd
Points to the window to maximize.

Remarks

Maximizes an 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 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, CMDIFrameWnd::MDIRestore

CMDIFrameWnd::MDINext

Syntax

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, restores the currently active child and maximizes the newly activated child.

See Also

CMDIFrameWnd::MDIActivate, CMDIFrameWnd::MDIGetActive, WM_MDINEXT

CMDIFrameWnd::MDIRestore

Syntax

void MDIRestore(CWnd* *pWnd* **);**

Parameters

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

Syntax

CMenu* MDISetMenu(CMenu* *pFrameMenu*, **CMenu*** *pWindowMenu* **);**

Parameters

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

Replaces 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 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.

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

Syntax **void MDITile();**

void MDITile(int *nType* **);**

Parameters *nType*

Specifies a tiling flag. This parameter can be one of the following flags:

Value	Meaning
MDITILE_HORIZONTAL	Tiles MDI child windows horizontally (one window appears beside another).
MDITILE_SKIPDISABLED	Prevents disabled MDI child windows from being tiled.
MDITILE_VERTICAL	Tiles MDI child windows vertically (one window appears above another).

Remarks Arranges all child windows in a tiled format.

See Also **CMDIFrameWnd::MDICascade**, **CMDIFrameWnd::MDIIconArrange**, **WM_MDITILE**

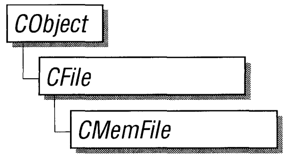
Data Members

CMDIFrameWnd::m_hWndMDIClient

Remarks

The **HWND** for the MDI client window owned by **CMDIFrameWnd**.

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.

#include <afx.h>

Comments

The following **CFile** functions are not implemented for **CMemFile**:

- **Duplicate**
- **LockRange**
- **UnlockRange**

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.

Derivation

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 Windows medium model, for example, derive a class with the four protected functions overridden. Your replacement functions should call the Windows **GlobalAlloc** family of functions.

Public Members

Construction/Destruction

CMemFile	Constructs a memory file using internally allocated memory.
~CMemFile	Closes the memory file, freeing allocated memory.

Member Functions

CMemFile::CMemFile

Syntax

CMemFile(UINT *nGrowBytes* **= 1024)**
throw (CFileException, CMemoryException);

Parameters

nGrowBytes
The memory-allocation increment in bytes.

Remarks

This constructor allocates memory and opens an empty memory file.

Example

```
CMemFile f; // ready to use - no Open necessary
```

CMemFile::~CMemFile

Syntax

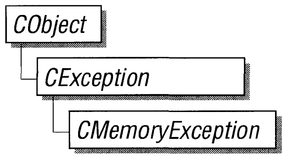
virtual ~CMemFile();

Remarks

This destructor 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>

Public Members

Construction/Destruction

CMemoryException	Constructs a **CMemoryException** object.

Member Functions

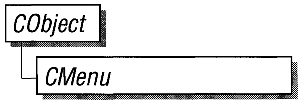
CMemoryException::CMemoryException

Syntax **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.

See Also Chapter 5, "Exception Processing," **AfxThrowMemoryException**

class CMenu : public CObject



The **CMenu** class is an encapsulation of the Windows **HMENU**. It provides member functions for creating, tracking, updating, and destroying menus. When you create a **CMenu** object, you associate it with a handle to a menu resource. Then you can use the class member functions to manage the menu.

See Also **CObject**

Public Members

Construction/Destruction

CMenu	Constructs a **CMenu** object.
~CMenu	Destroys a **CMenu** object.

Initialization

Attach	Attaches a Windows menu handle to a **CMenu** object.
Detach	Detaches a Windows menu handle from a **CMenu** object and returns the handle.
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

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

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.

Member Functions

CMenu::AppendMenu

Syntax

BOOL AppendMenu(UINT *nFlags*, **UINT** *nIDNewItem* **= 0,**
const char FAR* *lpNewItem* **= NULL);**

BOOL AppendMenu(UINT *nFlags*, **UINT** *nIDNewItem*,
const CBitmap* *pBmp* **);**

Parameters

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**.

lpNewItem
Specifies the content of the new menu item. The interpretation of *lpNewItem* depends on the setting of *nFlags* as shown below:

nFlags	Interpretation of lpNewItem
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.
MF_STRING	Contains a pointer to a null-terminated string. This is the default interpretation.

The *lpNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

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 **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*:

Value	Interpretation of nFlags
MF_CHECKED	Acts as a toggle in conjunction with **MF_UNCHECKED** to place the default check mark next to the item. When the application supplies check-mark bitmaps (see **SetMenuItemBitmaps**), 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.
MF_GRAYED	Disables the menu item so that it cannot be selected, and dims it.
MF_MENUBARBREAK	Places 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 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.

Value	Interpretation of nFlags
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.
MF_UNCHECKED	Acts as a toggle in conjunction with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see **SetMenuItemBitmaps**), the "check mark off" bitmap is displayed.

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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CWnd::DrawMenuBar, **CMenu::InsertMenu**, **CMenu::RemoveMenu**, **CMenu::SetMenuItemBitmaps**, **CMenu::Detach**, **CMenu::~CMenu**, **::AppendMenu**

CMenu::Attach

Syntax

BOOL Attach(HMENU *hMenu***);**

Parameters

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

TRUE if the operation was successful; otherwise **FALSE**.

See Also

CMenu::Detach, **CMenu::CMenu**

CMenu::CheckMenuItem

Syntax

UINT CheckMenuItem(UINT *nIDCheckItem*, **UINT** *nCheck* **);**

Parameters

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:

Value	Interpretation of nCheck
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).

Value	Interpretation of nCheck
MF_CHECKED	Acts as a toggle in conjunction with **MF_UNCHECKED** to place the default check mark next to the item. When the application supplies check-mark bitmaps (see **SetMenuItemBitmaps**), the "check mark on" bitmap is displayed.
MF_UNCHECKED	Acts as a toggle in conjunction with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see **SetMenuItemBitmaps**), the "check mark off" bitmap is displayed.

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_CHECKED**, or –1 if the menu item did not exist.

See Also

CMenu::GetMenuState, ::CheckMenuItem

CMenu::CMenu

Syntax

CMenu();

Remarks

The menu is not created until you call one of the create or load member functions.

See Also

CMenu::CreateMenu, CMenu::CreatePopupMenu, CMenu::LoadMenu, CMenu::LoadMenuIndirect, CMenu::~CMenu, CMenu::Attach

CMenu::~CMenu

Syntax

virtual ~CMenu();

Destroys the attached menu. If the **m_hMenu** data member was appended or inserted into another menu, it should be detached from this **CMenu** object before the destructor destroys it.

See Also

CMenu::CMenu, **CMenu::DestroyMenu**, **CMenu::Detach**

CMenu::CreateMenu

Syntax

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 functions.

If the menu is assigned to a window, it is automatically destroyed when the window is destroyed.

Return Value

TRUE if the menu was created successfully; otherwise **FALSE**.

See Also

CMenu::CMenu, **CMenu::DestroyMenu**, **CMenu::InsertMenu**, **CWnd::SetMenu**, **::CreateMenu**

CMenu::CreatePopupMenu

Syntax

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 functions. The application can add the pop-up menu to an existing menu or pop-up menu. **TrackPopupMenu** may be used to display this menu as a floating 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.

Return Value

TRUE if the pop-up menu was successfully created; otherwise **FALSE**.

See Also

CMenu::CreateMenu, **CMenu::InsertMenu**, **CWnd::SetMenu**, **CMenu::TrackPopupMenu**, **::CreatePopupMenu**

CMenu::DeleteMenu

Syntax

BOOL DeleteMenu(UINT *nPosition*, **UINT** *nFlags* **);**

Parameters

nPosition
Specifies the menu item that is to be deleted, as determined by *nFlags.*

nFlags
The following list shows how *nFlags* is used to interpret *nPosition.*

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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CWnd::DrawMenuBar, **::DeleteMenu**

CMenu::DestroyMenu

Syntax

BOOL DestroyMenu();

Remarks

Destroys the menu and any Windows 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

TRUE if the menu is destroyed; otherwise **FALSE**.

See Also

CMenu::~CMenu, **::DestroyMenu**

CMenu::Detach

Syntax

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::~CMenu, **CMenu::Attach**

CMenu::EnableMenuItem

Syntax

UINT EnableMenuItem(UINT *nIDEnableItem*, **UINT** *nEnable* **);**

Parameters

nIDEnableItem
Specifies the menu item to be enabled, as determined by *nEnable*.

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:

Value	Interpretation of nEnable
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).
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.

Return Value Previous state (**MF_DISABLED**, **MF_ENABLED**, or **MF_GRAYED**) or –1 if not valid.

See Also **CMenu::GetMenuState**, **::EnableMenuItem**

CMenu::GetMenuItemCount

Syntax **UINT GetMenuItemCount()**

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

Syntax

UINT GetMenuItemID(int *nPos* **) const;**

Parameters

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, ::GetMenuItemID

CMenu::GetMenuState

Syntax

UINT GetMenuState(UINT *nID*, **UINT** *nFlags* **) const;**

Parameters

nID
Specifies the menu item ID, as determined by *nFlags*.

nFlags
Specifies the nature of *nID*. It can be one of the following values:

Value	Interpretation of nFlags
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

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):

Value	Meaning
MF_CHECKED	Acts as a toggle in conjunction with **MF_UNCHECKED** to place the default check mark next to the item. When the application supplies check-mark bitmaps (see **SetMenuItemBitmaps**), 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.
MF_GRAYED	Disables the menu item so that it cannot be selected, and dims it.
MF_MENUBARBREAK	Places 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 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 in conjunction with **MF_CHECKED** to remove a check mark next to the item. When the application supplies check-mark bitmaps (see **SetMenuItemBitmaps**), the "check mark off" bitmap is displayed.

See Also

::GetMenuState, CMenu::CheckMenuItem, CMenu::EnableMenuItem

CMenu::GetMenuString

Syntax

int GetMenuString(UINT *nIDItem*, **LPSTR** *lpString*, **int** *nMaxCount*,
UINT *nFlags* **) const;**

Parameters

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

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::GetSubMenu

Syntax

CMenu* GetSubMenu(int *nPos* **) const;**

Parameters

nPos
Specifies the position of the pop-up menu contained in the menu. Position values start at 0 for the first menu item.

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

Syntax

BOOL InsertMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem* **= 0, const char FAR*** *lpNewItem* **= NULL);**

BOOL InsertMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem*, **const CBitmap*** *pBmp* **);**

Parameters

nPosition
Specifies the menu item before which the new menu item is to be inserted. The interpretation of *nPosition* depends on the setting of *nFlags*, as shown in the following list:

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 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. *nIDNewItem* is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

lpNewItem
Specifies the content of the new menu item. The interpretation of *lpNewItem* depends on the setting of *nFlags* as shown below:

nFlags	Interpretation of lpNewItem
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.
MF_STRING	Contains a long pointer to a null-terminated string. This is the default interpretation.

The *lpNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

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.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CMenu::AppendMenu, **CWnd::DrawMenuBar**, **CMenu::SetMenuItemBitmaps**, **CMenu::Detach**, **CMenu::~CMenu**, **::InsertMenu**

CMenu::LoadMenu

Syntax

BOOL LoadMenu(const char FAR* *lpMenuName* **);**

BOOL LoadMenu(UINT *nIDMenu* **);**

Parameters

lpMenuName
Points to a null-terminated string that contains the name of the menu resource to load.

nIDMenu
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.

Return Value

TRUE if the menu resource was loaded successfully; otherwise **FALSE**.

See Also

CMenu::AppendMenu, **CMenu::DestroyMenu**, **CMenu::LoadMenuIndirect**, **::LoadMenu**.

CMenu::LoadMenuIndirect

Syntax

BOOL LoadMenuIndirect(const BYTE FAR* *lpMenuTemplate* **);**

Parameters

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 {
   WORD   versionNumber;
   WORD   offset;
} MENUITEMTEMPLATEHEADER;
```

The **MENUITEMTEMPLATE** structure has the following generic form:

```
typedef struct {
   WORD mtOption;
   WORD mtID;
   char mtString;
} MENUITEMTEMPLATE[1];
```

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 **AppendMenu** 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.

Return Value

TRUE if the menu resource was loaded successfully; otherwise **FALSE.**

See Also

CMenu::DestroyMenu, **CMenu::LoadMenu**, **::LoadMenuIndirect**

CMenu::ModifyMenu

Syntax

BOOL ModifyMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem* **= 0, const char FAR*** *lpNewItem* **= NULL);**

BOOL ModifyMenu(UINT *nPosition*, **UINT** *nFlags*, **UINT** *nIDNewItem*, **const CBitmap*** *pBmp***);**

Parameters

nPosition
Specifies the menu item to be changed. The interpretation of *nPosition* depends on the setting of *nFlags* as shown in the following list:

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**.

lpNewItem
Specifies the content of the new menu item. The interpretation of *lpNewItem* depends on the setting of *nFlags* as shown below:

nFlags	Interpretation of lpNewItem
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**.

The *lpNewItem* parameter is ignored (not needed) if *nFlags* is set to **MF_SEPARATOR**.

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** functions.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CMenu::AppendMenu, **CMenu::InsertMenu**, **CMenu::CheckMenuItem**, **CWnd::DrawMenuBar**, **CMenu::EnableMenuItem**, **CMenu::SetMenuItemBitmaps**, **CMenu::Detach**, **CMenu::~CMenu**, **::ModifyMenu**

CMenu::RemoveMenu

Syntax

BOOL RemoveMenu(UINT *nPosition*, **UINT** *nFlags* **);**

Parameters

nPosition
Specifies the menu item to be removed. The interpretation of *nPosition* depends on the setting of *nFlags* as shown in the following list:

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, allowing the menu to be reused. Before calling this function, the application may call **GetSubMenu** 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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CWnd::DrawMenuBar, **CMenu::GetSubMenu**, **::RemoveMenu**

CMenu::SetMenuItemBitmaps

Syntax

BOOL SetMenuItemBitmaps(UINT *nPosition*, **UINT** *nFlags*,
const CBitmap* *pBmpUnchecked*, **const CBitmap*** *pBmpChecked*);

Parameters

nPosition
Specifies the menu item to be changed. The interpretation of *nPosition* depends on the setting of *nFlags* as shown in the following list:

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, Windows displays the appropriate bitmap next to the menu item.

If either *pBmpUnchecked* or *pBmpChecked* is **NULL**, then Windows displays nothing next to the menu item for the corresponding attribute. If both parameters are **NULL**, Windows 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. It is the responsibility of the application to 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

TRUE if the function is successful; otherwise **FALSE**.

See Also

::GetMenuCheckMarkDimensions, **::SetMenuItemBitmaps**

CMenu::TrackPopupMenu

Syntax

BOOL TrackPopupMenu(UINT *nFlags*, **int** *x*, **int** *y*, **const CWnd*** *pWnd*, **LPRECT** *lpRectReserved* **= 0);**

Parameters

nFlags

Specifies a screen-position flag and a mouse-button flag. The screen-position flag can be one of the following:

Value	Meaning
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:

Value	Meaning
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 left side of the menu on the screen.

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.

lpRectReserved
Points to a **RECT** structure or **CPoint** 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 version 3.0.

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.

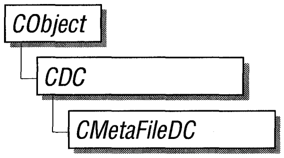
Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CMenu::CreatePopupMenu, **CMenu::GetSubMenu**, **::TrackPopupMenu**

class CMetaFileDC : public CDC



A Windows metafile contains a sequence of GDI commands that can be replayed to create a desired image or text.

To implement a Windows metafile, first create a **CMetaFileDC** object.

You create a **CMetaFileDC** in two steps. First, call the constructor **CMetaFileDC** to construct the **CMetaFileDC** object, then call the **Create** member function, which creates a Windows metafile device context and attaches it to the **CMetaFileDC** object.

After the **CMetaFileDC** object is created, send a sequence of **CDC** GDI commands to the metafile device context. Use only those GDI commands that create output, such as **MoveTo** and **LineTo**.

Then call the **Close** member function, which closes the metafile device context and returns a metafile handle. Use the **CMetaFileDC** destructor to destroy the **CMetaFileDC** object.

CDC::PlayMetaFile can then use the metafile handle to play the metafile repeatedly, and 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.

See Also **CDC**

Public Members

Construction/Destruction

CMetaFileDC	Constructs a **CMetaFileDC** object.

Initialization

Create	Creates the Windows metafile device context and attaches it to the **CMetaFileDC** object.

Operations

Close	Closes the device context, and creates a metafile handle.
SelectObject	Selects a GDI drawing object into the specified device context, which replaces the previous object of the same type.
SelectStockObject	Retrieves one of the predefined stock pens, brushes, or fonts and causes the stock object to become the currently selected object of its type.

Member Functions

CMetaFileDC::Close

Syntax

HANDLE 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** function and also used to manipulate the metafile with Windows functions such as **CopyMetaFile**.

The metafile should be deleted after use by calling the Windows **DeleteMetaFile** function.

Return Value

A valid **HANDLE** to a metafile if the function is successful. Otherwise, it is **NULL**.

See Also

CDC::PlayMetaFile, **::CloseMetaFile**, **::GetMetaFileBits**, **::CopyMetaFile**, **::DeleteMetaFile**

CMetaFileDC::CMetaFileDC

Syntax

CMetaFileDC();

Remarks

Construction of a **CMetaFileDC** object is a two-step process. 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

Syntax

BOOL Create(const char FAR* *lpFilename* = **NULL);**

Parameters

lpFilename
Points to a null-terminated character string. Names an existing metafile on disk to load. If *lpFilename* is **NULL**, a new in-memory metafile is created.

Remarks

You 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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CMetaFileDC::CMetaFileDC, ::CreateMetaFile

CMetaFileDC::SelectObject

Syntax

BOOL SelectObject(CGdiObject* *pObject* **);**

Parameters

pObject
Identifies the object to be selected into the **CMetaFileDC**. The selected object can be one of the following:

CBitmap
CBrush
CFont
CPen
CRgn
CPalette

Remarks

Selects an object into the **CMetaFileDC**.

CMetaFileDC performs its own object cleanup, so an application does not have to reselect default objects when recording a metafile.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CGdiObject::DeleteObject, **CDC::SelectClipRgn**, **CDC::SelectPalette**, **::SelectObject**

CMetaFileDC::SelectStockObject

Syntax

BOOL SelectStockObject(int *nIndex* **);**

Parameters

nIndex
Specifies the type of stock object desired. See **CDC::SelectObject** for a list of the possible stock objects.

Remarks

Selects one of the predefined stock pens, brushes, or fonts into the **CMetaFileDC**.

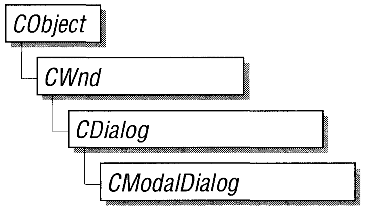
Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CDC::SelectStockObject

class CModalDialog : public CDialog



The **CModalDialog** class provides modal dialog boxes. In this type of dialog box, the user must respond before any processing outside the dialog box is possible. This is untrue for modeless dialog boxes.

Except in the most trivial cases, such as an About dialog box, you must derive your own modal dialog class from **CModalDialog**.

In your derived class, you can add member variables and member functions to specify the behavior of the dialog box. Add member variables to store data entered by the user via the dialog box controls or to store data for display through the controls. Add member functions to set and get this data. Add message-handler member functions to process messages for the controls in the dialog box.

Like other classes derived from class **CWnd**, classes derived from **CModalDialog** need their own message maps. If you declare any message-handler member functions, you must also provide a message map that connects the Windows messages with your handlers.

Note The three most common functions, **OnInitDialog**, **OnOK**, and **OnCancel**, do not need message-map entries.

Create a modal dialog object by constructing it. To do this, create a dialog object using the **CModalDialog** constructor, as shown in the example below.

Once the dialog object has been constructed, call its **DoModal** member function to run the dialog box. For example, to construct a modal dialog object of class `CMyModal` and run the dialog box, use the following coding:

```
CMyModal  myModalDlg;
myModalDlg.DoModal();
```

When the user clicks one of the dialog-box push buttons, such as OK or Cancel, the dialog box closes and it is removed from the screen.

After the user closes the dialog box, its member variables are accessed through the member functions that you defined to get information entered by the user.

For example, for a modal dialog box that has an editable text control, you can override the **OnOK** message-handler function in your derived modal dialog class so that when the user clicks the OK button, **OnOK** retrieves the text entered in the control and stores it in a data member of the dialog object. Later, after

DoModal returns, you can call a member function of the dialog object to retrieve the stored text.

You are responsible for supplying the member variables and member functions needed to access the dialog's data. Declare them in the class you derive from **CModalDialog**.

CDialog::EndDialog is called automatically in **OnOK** and **OnCancel** when the user closes the dialog box.

If the dialog box requires some initialization, override the **CDialog::OnInitDialog** member function to perform the initialization. For example, if an edit field in the dialog box is to display a default value that the user can accept or replace, override **OnInitDialog** to initialize the default text in the edit field. **OnInitDialog** is called automatically while the dialog is being created before the dialog box appears on the screen.

See Also **CModalDialog::DoModal, CDialog::EndDialog, CWnd::MessageBox, CModalDialog::OnOK, CModalDialog::OnCancel, WM_INITDIALOG, WM_CLOSE, WM_SETFONT**

Public Members

Construction/Destruction

CModalDialog	Constructs a **CModalDialog** object and stores the parameters for use when the member function **DoModal** is called.

Initialization

CreateIndirect	Initializes a **CModalDialog** object as the second phase of indirect dialog-box creation (nonresource based). The parameters are stored until the function **DoModal** is called.

Operations

DoModal	Invokes the dialog box and returns when done.

Overridables

OnOK	Override this member to perform the OK button action. The default terminates the dialog box, and **DoModal** will return **IDOK**.
OnCancel	Override this member to perform the Cancel button action. The default terminates the dialog box, and **DoModal** will return **IDCANCEL**.

Member Functions

CModalDialog::CModalDialog

Syntax

CModalDialog(const char FAR* *lpTemplateName*,
CWnd* *pParentWnd* **= NULL);**

CModalDialog(UINT *nIDTemplate*, **CWnd*** *pParentWnd* **= NULL);**

Parameters

lpTemplateName
Contains a string that is the name of a dialog-box resource template.

pParentWnd
Points to the parent window object (of type **CWnd**) of the dialog object. If it is **NULL**, the dialog object's parent window is set to the main application window, as shown in the following code:

```
if( pParentWnd == NULL )
    pParentWnd = AfxGetApp()->m_pMainWnd;
```

nIDTemplate
Contains a dialog resource template ID number.

Remarks

Provides two public constructors, with different argument signatures, to permit the construction of **CModalDialog** objects directly or from resource templates.

When you construct the dialog object to be used with **CreateIndirect**, pass **NULL** for the first parameter because there is no resource file template to be used in this case.

See Also

CModalDialog::CreateIndirect

CModalDialog::CreateIndirect

Syntax

BOOL CreateIndirect(HANDLE *hDialogTemplate* **);**

Parameters

hDialogTemplate
Contains a resource handle to global memory containing a dialog-box resource template. The template data structure is of type **DLGTEMPLATE**, which identifies the block of memory used as a dialog-box template.

Remarks

This member function uses a dialog resource constructed in memory to initialize a modal dialog object. The resource has the form of a **DLGTEMPLATE** structure. For more information on this structure, see the Windows Software Development Kit documentation.

To create a modal dialog indirectly, first create a **DLGTEMPLATE** structure in memory and retain a **HANDLE** to it. Then call the **CModalDialog** constructor to construct the dialog object. In this case, pass **NULL** for the first parameter to the constructor. Next, call **CreateIndirect** to store your handle to the in-memory dialog template. The Windows dialog is created and displayed later, when the **DoModal** member function runs.

Return Value

TRUE if the dialog object was created and initialized successfully; otherwise **FALSE**.

See Also

WM_INITDIALOG, **DS_SETFONT**, **DLGTEMPLATE**, **CWnd::DestroyWindow**, **CModalDialog::CModalDialog**

CModalDialog::DoModal

Syntax

int DoModal();

Remarks

Invokes the dialog box and returns 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 push buttons 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** and **OnCancel** member functions will end the dialog with results **IDOK** and **IDCANCEL**, respectively. You can override these message-handler functions to alter this behavior.

Return Value

An **int** value that specifies the value of the *nResult* parameter that was passed to the **EndDialog** member function, which is used to terminate the dialog box.The return value is –1 if the function could not create the dialog box.

See Also

::DialogBox

CModalDialog::OnCancel

Syntax **virtual void OnCancel();**

Remarks Override this member function to perform Cancel button action. The default simply terminates the dialog box and causes **DoModal** to return **IDCANCEL**.

See Also **CModalDialog::OnOK, WM_COMMAND**

CModalDialog::OnOK

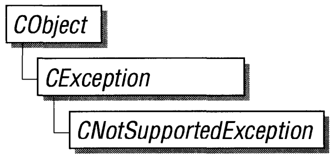
Syntax **virtual void OnOK();**

Remarks Override this member function to perform OK button action. The default simply terminates the dialog box and causes **DoModal** to return **IDOK**.

See Also **CModalDialog::OnCancel, WM_COMMAND**

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>

Public Members

CNotSupportedException	Constructs a **CNotSupportedException** object.

Member Functions

CNotSupportedException ::CNotSupportedException

Syntax **CNotSupportedException();**

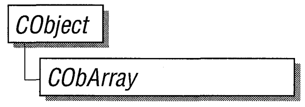
Remarks Constructs a **CNotSupportedException** object.

Do not use this constructor directly, but rather call the global function **AfxThrowNotSupportedException**.

See Also Chapter 5, "Exception Processing," **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.

#include <afxcoll.h>

See Also **CStringArray**, **CPtrArray**, **CByteArray**, **CWordArray**, **CDWordArray**

Derivation Array class derivation is similar to list derivation. For details on the derivation of a special-purpose list class, see the tutorial 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.

Public Members

Construction/Destruction

CObArray	Constructs an empty array for **CObject** pointers.
~CObArray	Destroys a **CObArray** object.

Bounds

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

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

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

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

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator [ ]	Sets or gets the element at the specified index.

Member Functions

CObArray::Add

Syntax

int Add(CObject* *newElement* **)**
throw(CMemoryException);

Parameters

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.

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
```

See Also

CObArray::SetAt, **CObArray::SetAtGrow**, **CObArray::InsertAt**, **CObArray::operator [ ]**

CObArray::CObArray

Syntax

CObArray();

Remarks

Constructs an empty **CObject** pointer array. The array grows one element at a time.

Example

See the **CObList** constructor for a listing of the `CAge` class used in all collection examples.

```
CObArray array;  // Array on the stack
```

```
CObArray* parray = new CObArray;  // Array on the heap
```

See Also

CObList constructor

CObArray::~CObArray

Syntax

~CObArray();

Remarks

Destroys a **CObArray** object but does not destroy the **CObject** objects that are referenced in the array.

CObArray::ElementAt

Syntax

CObject*& ElementAt(int *nIndex* **);**

Parameters

nIndex
An integer index that is greater than or equal to 0 and less than or equal to **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 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

Syntax **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

Syntax **CObject* GetAt(int** *nIndex* **) const;**

Parameters

nIndex
An integer index that is greater than or equal to 0 and less than or equal to **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.

Example

```
CObArray array;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
ASSERT( *(CAge*) array.GetAt( 0 ) == CAge( 21 ) );
```

See Also **CObArray::SetAt**, **CObArray::operator [ ]**

CObArray::GetSize

Syntax

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

Syntax

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.

Example

```
CObArray array;

array.Add( new CAge( 21 ) ); // Element 0
array.Add( new CAge( 40 ) ); // Element 1
ASSERT( array.GetUpperBound() == 1 ); // Largest index
```

See Also

CObArray::GetSize, **CObArray::SetSize**

CObArray::InsertAt

Syntax

void InsertAt(int *nIndex*, **CObject*** *newElement*, **int** *nCount* **= 1)**
throw(CMemoryException);

void InsertAt(int *nStartIndex*, **CObArray*** *pNewArray* **)**
throw(CMemoryException);

Parameters

nIndex
An integer index that may be greater than **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 **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.

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
```

See Also

CObArray::SetAt, **CObArray::RemoveAt**

CObArray::RemoveAll

Syntax

void RemoveAll();

Remarks

Removes all the pointers from this array but does not actually delete the **CObject** objects. If the array is empty already, 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

Syntax

void RemoveAt(int *nIndex*, **int** *nCount* **= 1);**

Parameters

nIndex
An integer index that is greater than or equal to 0 and less than or equal to **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.

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
```

See Also

CObArray::SetAt, **CObArray::SetAtGrow**, **CObArray::InsertAt**

CObArray::SetAt

Syntax

void SetAt(int *nIndex*, **CObject*** *newElement* **);**

Parameters

nIndex
An integer index that is greater than or equal to 0 and less than or equal to **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.

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
```

See Also

CObArray::GetAt, **CObArray::SetAtGrow**, **CObArray::ElementAt**, **CObArray::operator [ ]**

CObArray::SetAtGrow

Syntax

void SetAtGrow(int *nIndex*, **CObject*** *newElement* **)**
throw(CMemoryException);

Parameters

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).

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
```

See Also

CObArray::GetAt, **CObArray::SetAt**, **CObArray::ElementAt**, **CObArray::operator [ ]**

CObArray::SetSize

Syntax

void SetSize(int *nNewSize*, **int** *nGrowBy* **= –1)**
throw(CMemoryException);

Parameters

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 [ ]

Syntax

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.

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
```

See Also

CObArray::GetAt, **CObArray::SetAt**

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

Refer to Part 1 for a detailed description of these features.

Note **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.

#include <afx.h>

Derivation

You will realize major benefits from **CObject** derivation if you use some of the optional macros in your class implementation and declarations.

The first-level macros, **DECLARE_DYNAMIC** and **IMPLEMENT_DYNAMIC**, permit run-time access to the class name and its position in the hierarchy. This, in turn, allows meaningful diagnostic dumping.

The second-level macros, **DECLARE_SERIAL** and **IMPLEMENT_SERIAL**, include all the functionality of the first-level macros, and they enable an object to be *serialized* to and from an *archive.*

For important information about deriving Microsoft Foundation classes and C++ classes in general, see "How to Derive a Class from CObject" in Chapter 8 of the *Class Libraries User's Guide.*

Public Members

Construction/Destruction

~CObject	Virtual destructor.
operator new	Special **new** operator.
operator delete	Special **delete** operator.

Diagnostics

AssertValid	Validates this object's integrity.
Dump	Produces a diagnostic dump of this object.

Serialization

IsSerializable	Tests to see if this object can be serialized.
Serialize	Loads or stores an object from/to an archive.

Miscellaneous

GetRuntimeClass	Returns the **CRuntimeClass** structure corresponding to this object's class.
IsKindOf	Tests this object's relationship to a given class.
Construct	An internal function that must be public—do not use.

Protected Members

CObject	Default constructor.

Private Members

CObject	Copy constructor.
operator =	Assignment operator.

Macros

RUNTIME_CLASS	Returns the **CRuntimeClass** structure corresponding to the named class.
DECLARE_DYNAMIC	Permits access to run-time class information (used in each class declaration).
IMPLEMENT_DYNAMIC	Permits access to run-time class information (used once in the class implementation).
DECLAR_SERIAL	Permits serialization and access to run-time class information (used in each class declaration).
IMPLEMENT_SERIAL	Permits serialization and access to run-time class information (used once in the class implementation).

Member Functions and Macros

CObject::AssertValid

Syntax

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 ) && ( m_years < 105 ) );
}
```

CObject::CObject

Syntax

CObject();

CObject(const CObject& *objectSrc* **);**

Parameters

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 "How to Derive a Class from CObject" in Chapter 8 of the *Class Libraries User's Guide.*

The standard 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

Syntax

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, 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.

DECLARE_DYNAMIC Macro

Syntax

DECLARE_DYNAMIC(*class_name* **)**

Parameters

class_name
The actual name of the class (without quotes).

Remarks

DECLARE_DYNAMIC generates the C++ header code necessary for a **CObject**-derived class with accessible run-time information. Use the **DECLARE_DYNAMIC** 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 "How to Derive a Class from CObject" in Chapter 8 of the *Class Libraries User's Guide.*

If **DECLARE_DYNAMIC** is included in the class declaration, then **IMPLEMENT_DYNAMIC** must be included in the class implementation.

The **DECLARE_SERIAL** macro includes all the functionality of **DECLARE_DYNAMIC** but adds the ability to serialize the object.

See Also

DECLARE_SERIAL, IMPLEMENT_DYNAMIC

DECLARE_SERIAL Macro

Syntax

DECLARE_SERIAL(*class_name* **)**

Parameters

class_name
The actual name of the class (without quotes).

Remarks

DECLARE_SERIAL generates the C++ header code necessary for a **CObject**-derived class that can be serialized. 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 "How to Derive a Class from CObject," in Chapter 8 of the *Class Libraries 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**.

See Also **DECLARE_DYNAMIC, IMPLEMENT_SERIAL**

CObject::Dump

Syntax **virtual void Dump(CDumpContext&** *dc* **) const;**

Parameters

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 at the end of its output.

Dump calls make sense only in the Debug version of the Microsoft Foundation 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 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 << m_years;
}
```

CObject::GetRuntimeClass

Syntax

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**.

Example

```
CAge a(21);
CRuntimeClass* prt = a.GetRuntimeClass();
ASSERT( strcmp( prt->m_pszClassName, "CAge" )  == 0 );
```

See Also **CObject::IsKindOf**, **RUNTIME_CLASS**

IMPLEMENT_DYNAMIC Macro

Syntax **IMPLEMENT_DYNAMIC(** *class_name*, *base_class_name* **)**

Parameters

class_name
The actual name of the class (without quotes).

base_class_name
The name of the base class (without quotes).

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 "How to Derive a Class from CObject," in Chapter 8 of the *Class Libraries User's Guide.*

See Also **IMPLEMENT_SERIAL**

IMPLEMENT_SERIAL Macro

Syntax **IMPLEMENT_SERIAL(** *class_name*, *base_class_name*, *wSchema* **)**

Parameters

class_name
The actual name of the class (without quotes).

base_class_name
The name of the base class (without quotes).

wSchema
Placeholder for future implementation. The class schema number must not be –1. This is a *version number* that will be encoded in the archive to enable a deserializing program to identify and handle data created by earlier program versions.

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 "How to Derive a Class from CObject" in Chapter 8 of the *Class Libraries User's Guide.*

See Also

IMPLEMENT_DYNAMIC

CObject::IsKindOf

Syntax

BOOL IsKindOf(const CRuntimeClass* *pClass* **) const;**

Parameters

pClass
A pointer to a **CRuntimeClass** structure associated with your **CObject**-derived class.

Remarks

IsKindOf tests this object to see if (1) it is an object of the specified class or (2) if 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 C++ polymorphism feature. Use virtual functions instead.

Return Value

TRUE if the object corresponds to the class; otherwise **FALSE**.

Example

```
CAge a(21); // must use IMPLEMENT_DYNAMIC or IMPLEMENT_SERIAL
ASSERT( a.IsKindOf( RUNTIME_CLASS( CAge ) ) );
```

See Also

CObject::GetRuntimeClass, **RUNTIME_CLASS**

CObject::IsSerializable

Syntax

BOOL IsSerializable() const;

Remarks

IsSerializable 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**.

Example

```
CAge a(21);
ASSERT( a.IsSerializable() );
```

See Also

CObject::Serialize

CObject::Serialize

Syntax

virtual void Serialize(CArchive& *ar* **)**
throw(CMemoryException, CArchiveException, CFileException);

Parameters

ar
A **CArchive** object to serialize to or from.

Remarks

Serialize 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 both the cookbook and the tutorial in the *Class Libraries User's Guide.*

Example

```
   void CAge::Serialize( CArchive& ar )
{
       CObject::Serialize( ar );
    if( ar.IsStoring() )
           ar << m_years;
    else
           ar >> m_years;
}
```

RUNTIME_CLASS Macro

Syntax

RUNTIME_CLASS(*class_name* **)**

Parameters

class_name
  The actual name of the class (without quotes).

Remarks

RUNTIME_CLASS returns a pointer to a **CRuntimeClass** structure for the class specified by *class_name.*

Example

```
CRuntimeClass* prt = RUNTIME_CLASS( CAge );
ASSERT( strcmp( prt->m_pszClassName, "CAge" )  == 0 );
```

See Also

DECLARE_DYNAMIC, CObject::GetRuntimeClass, IMPLEMENT_DYNAMIC

Operators

CObject::operator =

Syntax

void operator =(const CObject& *src* **);**

Remarks

The standard 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

Syntax

void operator delete(void* *p* **);**

Remarks

For the Release version of the library, **delete** simply frees the memory allocated by **new**. In the Debug version, **delete** participates in an allocation-monitoring scheme designed to detect memory leaks.

Note If you override **new** and **delete**, you forfeit the diagnostic capability.

See Also

CObject::operator new

CObject::operator new

Syntax

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, **new** performs an optimal memory allocation in a manner similar to **malloc**. In the Debug version, **new** participates in an allocation-monitoring scheme designed to detect memory leaks.

If you use the code line:

```
#define new DEBUG_NEW
```

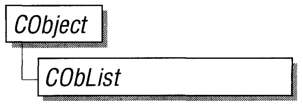
before any of your implementations in a .CPP file, then the second version of **new** will be used, storing the filename and line number in the allocated block for later reporting. You do not have to worry about supplying the extra parameters; a macro takes care of that for you.

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 non-unique **CObject** pointers accessible sequentially or by pointer value. **CObList** lists behave like doubly linked lists.

A variable of type **POSITION** is a kind of 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 in order 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 the 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.

#include <afxcoll.h>

See Also **CStringList**, **CPtrList**

Derivation The tutorial in the *Class Library User's Guide* illustrates the derivation of a `CPersonList` class from **CObList**. This new list class, designed to hold pointers to `CPerson` objects, adds a new data member 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.

Public Members

Construction/Destruction

CObList	Constructs an empty list for **CObject** pointers.
~CObList	Destroys a **CObList** object.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

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

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

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

InsertBefore	Inserts a new element before a given position.
InsertAfter	Inserts a new element after a given position.

Searching

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

GetCount	Returns the number of elements in this list.
IsEmpty	Tests for the empty list condition (no elements).

Member Functions

CObList::AddHead

Syntax

POSITION AddHead(CObject* *newElement* **)**
throw(CMemoryException);

void AddHead(CObList* *pNewList* **)**
throw(CMemoryException);

Parameters

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.

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
```

See Also

CObList::GetHead, CObList::RemoveHead

CObList::AddTail

Syntax

POSITION AddTail(CObject* *newElement* **)**
throw(CMemoryException);

void AddTail(CObList* *pNewList* **)**
throw(CMemoryException);

Parameters

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.

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
```

See Also

CObList::GetTail, CObList::RemoveTail

CObList::CObList

Syntax

CObList(int *nBlockSize* **= 10);**

Parameters

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::~CObList

Syntax

~CObList();

Remarks

Destroys a **CObList** object but does not destroy the **CObject** objects that are referenced in the list.

CObList::Find

Syntax

POSITION Find(CObject* *searchValue*,
POSITION *startAfter* = **NULL) const;**

Parameters

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.

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 ) );
}
```

See Also **CObList::GetNext**, **CObList::GetPrev**

CObList::FindIndex

Syntax **POSITION FindIndex(int** *nIndex* **) const;**

Parameters

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.

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 ) );
}
```

See Also **CObList::Find**, **CObList::GetNext**, **CObList::GetPrev**

CObList::GetAt

Syntax

CObject*& GetAt(POSITION *position* **);**

CObject* GetAt(POSITION *position* **) const;**

Parameters

position
A **POSITION** value returned by a previous **BeginIterate** or **Find** member function call.

Remarks

A variable of type **POSITION** is a kind of "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 library asserts.

Return Value

See the return value description for **GetHead**.

Example

See the example for **FindIndex**

See Also

CObList::Find, **CObList::SetAt**, **CObList::GetNext**, **CObList::GetPrev**, **CObList::GetHead**

CObList::GetCount

Syntax

int GetCount() const;

Remarks

Gets the number of elements in this list.

Return Value

An integer value containing the element count.

Example

```
CObList list;

list.AddHead( new CAge( 21 ) );
list.AddHead( new CAge( 40 ) ); // List now contains (40, 21)
ASSERT( list.GetCount() == 2 );
```

See Also **CObList::IsEmpty**

CObList::GetHead

Syntax

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 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.

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
```

See Also **CObList::GetTail, CObList::GetTailPosition, CObList::AddHead, CObList::RemoveHead**

CObList::GetHeadPosition

Syntax **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.

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 ) );
}
```

See Also **CObList::GetTailPosition**

CObList::GetNext

Syntax

CObject*& GetNext(POSITION& *rPosition* **);**

CObject* GetNext(POSITION& *rPosition* **) const;**

Parameters

rPosition
A reference to a **POSITION** value returned by a previous **GetNext**, **GetHeadPosition**, or other member function call.

Remarks

GetNext 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 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**.

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
   for( pos = list.GetHeadPosition(); pos != NULL; )
   {
#ifdef _DEBUG
       afxDump << list.GetNext( pos ) << "\\n";
#endif
   }
```

The results from this program are as follows:

```
a CAge at $479C 40
a CAge at $46C0 21
```

See Also

CObList::Find, **CObList::GetHeadPosition**, **CObList::GetTailPosition**, **CObList::GetPrev**, **CObList::GetHead**

CObList::GetPrev

Syntax

CObject*& GetPrev(POSITION& *rPosition* **);**

CObject* GetPrev(POSITION& *rPosition* **) const;**

Parameters

rPosition
A reference to a **POSITION** value returned by a previous **GetPrev** or other member function call.

Remarks

GetPrev 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 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**.

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
```

See Also

CObList::Find, **CObList::GetTailPosition**, **CObList::GetHeadPosition**, **CObList::GetNext**, **CObList::GetHead**

CObList::GetTail

Syntax

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 library asserts. Use **IsEmpty** to verify that the list contains elements.

Return Value

See the return value description for **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 ) );
```

See Also

CObList::GetTail, **CObList::AddHead**, **CObList::RemoveHead**, **CObList::GetHead**

CObList::GetTailPosition

Syntax

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.

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 ) );
}
```

See Also

CObList::GetHeadPosition, **CObList::GetTail**

CObList::InsertAfter

Syntax

POSITION InsertAfter(POSITION *position*, **CObject*** *newElement* **);**
throw (CMemoryException);

Parameters

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.

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
```

See Also

CObList::Find, **CObList::InsertBefore**

CObList::InsertBefore

Syntax

POSITION InsertBefore(POSITION *position*, **CObject*** *newElement* **)**
throw (CMemoryException);

Parameters

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.

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
```

See Also **CObList::Find**, **CObList::InsertAfter**

CObList::IsEmpty

Syntax **BOOL IsEmpty() const;**

Remarks Indicates if this list contains no elements.

Return Value **TRUE** if this list is empty; **FALSE** otherwise.

Example See the example for **RemoveAll**.

See Also **CObList::GetCount**

CObList::RemoveAll

Syntax **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

Syntax

void RemoveAt(POSITION *position* **);**

Parameters

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 library asserts.

Example

Be careful when removing an element during a list iteration. The following example shows a removal technique that guarantees a good **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

Syntax

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 library asserts. Use **IsEmpty** to verify that the list contains elements.

Return Value

The **CObject** pointer previously at the head of the list.

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;
```

See Also

CObList::GetHead, CObList::AddHead

CObList::RemoveTail

Syntax

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 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.

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
```

See Also

CObList::GetTail, **CObList::AddTail**

CObList::SetAt

Syntax

void SetAt(POSITION *pos*, **CObject*** *newElement* **);**

Parameters

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 kind of "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 library asserts.

Example

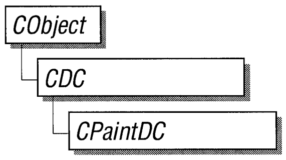
```
   CObList list;
   CObject* pa;
   POSITION pos;

   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
```

See Also **CObList::Find**, **CObList::GetAt**, **CObList::GetNext**, **CObList::GetPrev**

class CPaintDC : public CDC



The **CPaintDC** class is a device-context class derived from **CDC**. It performs a **BeginPaint** at construction time and **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.

See Also **CDC**

Public Members

Data Members

m_ps	Contains the **PAINTSTRUCT** used to paint the client area.

Construction/Destruction

CPaintDC	Constructs a **CPaintDC** connected to the specified **CWnd**.
~CPaintDC	Destroys a **CPaintDC**.

Protected Members

m_hWnd	The **HWND** to which this **CPaintDC** object is attached.

Member Functions

CPaintDC::CPaintDC

Syntax

CPaintDC(CWnd* *pWnd* **)**
throw(CResourceException);

Parameters

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 Windows.

CPaintDC::~CPaintDC

Syntax

virtual ~CPaintDC()

Remarks

Destroys a **CPaintDC** object and marks the end of painting the application window. In the process, the destructor destroys the paint structure.

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**.

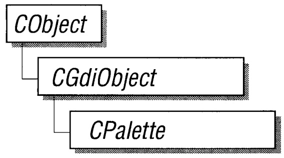
The **PAINTSTRUCT** contains information that the application uses to paint the client area of the window associated with a **CPaintDC** object.

The **PAINTSTRUCT** structure looks like this:

```
typedef struct tagPAINTSTRUCT {
   HDC  hdc;
   BOOL fErase;
   RECT rcPaint;
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved[16];
} PAINTSTRUCT;
```

Note You can access the device-context handle through the **PAINTSTRUCT**. However, you can access the handle more directly through the **m_hDC** member variable, which **CPaintDC** inherits from **CDC**.

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. Windows uses the application's logical palette (a list of needed colors) in conjunction with 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 GDI object) and to manipulate its entries and other properties.

Public Members

Construction/Destruction

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

CreatePalette	Initializes a **CPalette** object by creating a Windows color palette and attaching the palette to the **CPalette** object.

Operations

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 Windows 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

Syntax

void AnimatePalette(UINT *nStartIndex*, **UINT** *nNumEntries*,
LPPALETTEENTRY *lpPaletteColors* **);**

Parameters

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

Syntax

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

Syntax

BOOL CreatePalette(LPLOGPALETTE *lpLogPalette* **);**

Parameters

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;
```

Remarks

Initializes a **CPalette** object by creating a Windows logical color palette and attaching it to the **CPalette** object.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

::CreatePalette

CPalette::FromHandle

Syntax

static CPalette* FromHandle(HPALETTE *hPalette* **);**

Parameters

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. In other words, 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

Syntax

UINT GetNearestPaletteIndex(DWORD *crColor* **) const;**

Parameters

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

Syntax

UINT GetPaletteEntries(UINT *nStartIndex*, **UINT** *nNumEntries*,
LPPALETTEENTRY *lpPaletteColors* **) const;**

Parameters

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, or 0 if the function failed.

See Also

::GetPaletteEntries

CPalette::ResizePalette

Syntax

BOOL ResizePalette(UINT *nNumEntries* **);**

Parameters

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

TRUE if the palette was successfully resized; otherwise **FALSE**.

See Also

::ResizePalette

CPalette::SetPaletteEntries

Syntax

UINT SetPaletteEntries(UINT *nStartIndex*, **UINT** *nNumEntries*,
LPPALETTEENTRY *lpPaletteColors* **);**

Parameters

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

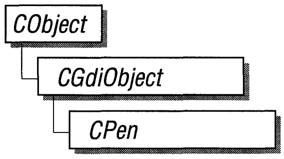
The number of entries set in the logical palette, or 0 if the function failed.

See Also

CDC::RealizePalette, **::SetPaletteEntries**

class CPen : public CGdiObject

The **CPen** class encapsulates a Windows graphical design interface (GDI) pen.



Public Members

Construction/Destruction

CPen	Constructs a **CPen** object.

Initialization

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

FromHandle	Returns a pointer to a **CPen** object when given a Windows **HPEN**.

Member Functions

CPen::CPen

Syntax

CPen();

CPen(int *nPenStyle*, **int** *nWidth*, **DWORD** *crColor*)
throw(CResourceException);

Parameters

nPenStyle
Specifies the pen style. This parameter can be one of the following values:

Value	Meaning
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 in which a line is drawn inside the frame of ellipses and rectangles produced by using the **Ellipse**, **Rectangle**, and **RoundRect** Windows functions.

nWidth
Specifies width (in pixels) of the pen.

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 **CreatePen**, **CreatePenIndirect**, or **CreateStockObject**. 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

Syntax

BOOL CreatePen(int *nPenStyle*, **int** *nWidth*, **DWORD** *crColor* **);**

Parameters

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).

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.

Return Value

TRUE if the function is successful; otherwise **FALSE**.

See Also

CPen::CreatePenIndirect, **CPen::CPen**

CPen::CreatePenIndirect

Syntax

BOOL CreatePenIndirect(LPLOGPEN *lpLogPen* **);**

Parameters

lpLogPen
Points to the Windows **LOGPEN** structure that contains information about the pen.

The **LOGPEN** structure has the following form:

```
typedef struct tagLOGPEN {
   WORD      lopnStyle;
   POINT     lopnWidth;
   COLORREF  lopnColor;
} LOGPEN;
```

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

TRUE if the function is successful; otherwise **FALSE**.

See Also

CPen::CreatePen, **CPen::CPen**

CPen::FromHandle

Syntax

static CPen* FromHandle(HPEN *hPen* **);**

Parameters

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 a 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.

See Also **CRect**, **CSize**

Public Members

Construction/Destruction

CPoint	Constructs a **CPoint**.

Operations

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

operator +	Returns a **CPoint** offset by a size.
operator –	Returns a **CPoint** offset by a negative size.

Operators Returning CSize Values

operator –	Returns the size difference between two points.

Member Functions

CPoint::CPoint

Syntax

CPoint();

CPoint(int *initX*, **int** *initY* **);**

CPoint(POINT *initPt* **);**

CPoint(SIZE *initSize* **);**

Parameters

initX
Sets the x member for the **CPoint**.

initY
Sets the y member for the **CPoint**.

initPt
Windows **POINT** structure used to initialize **CPoint**.

initSize
Sets the x and y member equal to the corresponding values in *cx* and *cy* values in *initSize*.

Remarks

Constructs a **CPoint** object. If no arguments are given, x and y members are not initialized.

CPoint::Offset

Syntax

void Offset(int *xOffset*, **int** *yOffset* **);**

void Offset(POINT *point* **);**

void Offset(SIZE *initSize* **);**

Parameters

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 to offset the **CPoint**.

initSize
Specifies the amount 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**, or **Size**.

Operators

CPoint::operator ==

Syntax

BOOL operator ==(POINT *point* **) const;**

Parameters

point
Contains a **POINT** or **CPoint**.

Remarks

Checks for equality between two points.

Return Value

TRUE if the points are equal; otherwise **FALSE**.

CPoint::operator !=

Syntax

BOOL operator !=(POINT *point* **) const;**

Parameters

point
Contains a **POINT** or **CPoint**.

Remarks

Checks for inequality between two points.

Return Value

TRUE if the points are not equal; otherwise **FALSE**.

CPoint::operator +=

Syntax

void operator +=(SIZE *size* **);**

Parameters

size
Contains a **SIZE** or a **CSize**.

Remarks

Offsets a **CPoint** by a size.

CPoint::operator –=

Syntax

void operator – =(SIZE *size* **);**

Parameters

size
Contains a **SIZE** or a **CSize**.

Remarks

Subtracts a size from the **CPoint**.

CPoint::operator +

Syntax

CPoint operator +(SIZE *size* **) const;**

Parameters

size
Contains a **SIZE** or a **CSize**.

Return Value

A **CPoint** that is offset by a size.

CPoint::operator –

Syntax

CSize operator – (POINT *point* **) const;**

CPoint operator – (SIZE *size* **) const;**

Parameters

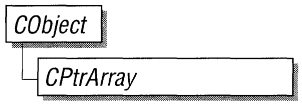
point
Contains a **POINT** or **CPoint**.

size
Contains a **SIZE** or **CSize**.

Return Value

A **CSize** that is the difference between two points, or returns a **CPoint** that is negatively offset by a size.

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>

Public Members

Construction/Destruction

CPtrArray	Constructs an empty array for void pointers.
~CPtrArray	Destroys a **CPtrArray** object.

Bounds

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

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

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

SetAtGrow	Sets the value for a given index, growing the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

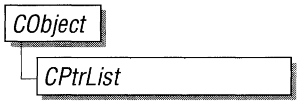
Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

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 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>

Public Members

Construction/Destruction

CPtrList	Constructs an empty list for **void** pointers.
~CPtrList	Destroys a **CPtrList** object.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

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

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

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

InsertBefore	Inserts a new element before a given position.
InsertAfter	Inserts a new element after a given position.

Searching

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

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 a **CRect** and Windows **RECT** structures.

A **CRect** object can be passed as a function parameter wherever a **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. 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 invalid rectangle.

When using overloaded **CRect** operators, the first operator must be a **CRect**; the second can be either a **RECT** or a **CRect**.

See Also **CPoint**, **CSize**

Public Members

Construction/Destruction

CRect	Constructs a **CRect** object.

Operations

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.
IntersectRect	Sets **CRect** equal to the intersection of two rectangles.
UnionRect	Sets **CRect** equal to the union of two rectangles.

Operators

operator LPRECT	Converts a **CRect** to a **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

Syntax **CPoint& BottomRight();**

Remarks Returns a reference to the bottom-right point of **CRect**.

Return Value **POINT&**, a reference to a **POINT**.

CRect::CopyRect

Syntax **void CopyRect(LPRECT** *lpSrcRect* **);**

Parameters *lpSrcRect*
Points to the **RECT** or **CRect** whose dimensions are to be copied.

Remarks Copies the *lpSrcRect* rectangle to the **CRect**.

See Also **::CopyRect**, **CRect::operator =**

CRect::CRect

Syntax

CRect();

CRect(int *l*, **int** *t*, **int** *r*, **int** *b* **);**

CRect(const RECT& *srcRect* **);**

CRect(LPRECT *lpSrcRect* **);**

CRect(POINT *point*, **SIZE** *size* **);**

Parameters

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 dimensions for the **CRect** object.

lpSrcRect
Points to the **RECT** structure with the dimensions 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(LPRECT**) member functions perform a **CopyRect**. The other constructors initialize the member variables of the object directly.

See Also

CRect::SetRect, **CRect::CopyRect**, **CRect::operator =**

CRect::EqualRect

Syntax

BOOL EqualRect(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to a **RECT** or **CRect** that contains the upper-left and lower-right corner coordinates of a rectangle.

Return Value

TRUE if the two rectangles have the same top, left, bottom, and right values; otherwise **FALSE**.

See Also

::EqualRect

CRect::Height

Syntax

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

Syntax

void InflateRect(int *x*, **int** *y* **);**

void InflateRect(SIZE *size* **);**

Parameters

x
Specifies the amount to increase or decrease **CRect**'s width. It must be negative to decrease the width.

y
Specifies the amount to increase or decrease **CRect**'s height. It must be negative to decrease the height.

size
Contains a **SIZE** or **CSize** that specifies x and y amounts to add to the **CRect**'s height and width.

Remarks

InflateRect's parameters are signed values; positive values inflate the **CRect**, and negative values deflate it.

When inflated, **CRect**'s width is increased by two times *x*, and its height is increased by two times *y*.

See Also

::InflateRect

CRect::IntersectRect

Syntax

int IntersectRect(LPRECT *lpRect1*, **LPRECT** *lpRect2* **);**

Parameters

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 **CRect** equal to the intersection of two existing rectangles. The intersection is the largest rectangle contained in both existing rectangles.

Return Value

TRUE if the intersection of the two rectangles is not empty. It is **FALSE** if the intersection is empty.

See Also

::IntersectRect, **CRect::operator &=**, **CRect::operator &**

CRect::IsRectEmpty

Syntax

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

TRUE if **CRect** is empty. **FALSE** if **CRect** is not empty.

See Also

::IsRectEmpty, **CRect::IsRectNull**

CRect::IsRectNull

Syntax

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

TRUE if **CRect**'s top, left, bottom, and right values are all equal to 0; otherwise **FALSE**.

See Also

CRect::IsRectEmpty

CRect::OffsetRect

Syntax

void OffsetRect(int *x*, **int** *y* **);**

void OffsetRect(POINT *point* **);**

void OffsetRect(SIZE *size* **);**

Parameters

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

Syntax

BOOL PtInRect(POINT *point* **) const;**

Parameters

point
Contains a **POINT** or **CPoint**.

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**.

Return Value

TRUE if the point lies within **CRect**; otherwise **FALSE**.

See Also

::PtInRect

CRect::SetRect

Syntax

void SetRect(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

Parameters

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

Syntax	**void SetRectEmpty();**
Remarks	Creates a **NULL** rectangle (all coordinates equal to 0).
See Also	**::SetRectEmpty**

CRect::Size

Syntax	**CSize Size() const;**
Return Value	The **CRect** width and height encapsulated as the *cx* and *cy* member variables of a **CSize** object.

CRect::TopLeft

Syntax	**CPoint& TopLeft();**
Return Value	A reference to the top-left point of **CRect**.

CRect::UnionRect

Syntax

int UnionRect(LPRECT *lpRect1***, LPRECT** *lpRect2* **);**

Parameters

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.

Windows ignores the dimensions of an empty rectangle; that is, a rectangle that has no height or has no width.

Return Value

TRUE if the union is not empty; **FALSE** if the union is empty.

See Also

::UnionRect, **CRect::operator |=**, **CRect::operator |**

CRect::Width

Syntax

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 LPRECT

Syntax **operator LPRECT();**

Remarks Converts a **CRect** to a **LPRECT**, with no need for the AND (**&**) operator.

CRect::operator =

Syntax **void operator =(const RECT&** *srcRect* **);**

Parameters *srcRect*
Refers to a source rectangle.

Remarks Copies the dimensions of *srcRect* to **CRect**.

See Also **CRect::SetRect**, **::CopyRect**

CRect::operator ==

Syntax **BOOL operator ==(const RECT&** *rect* **) const;**

Parameters *rect*
Refers to a source rectangle.

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 **TRUE**; otherwise **FALSE**.

See Also **::EqualRect**

CRect::operator !=

Syntax

BOOL operator !=(const RECT& *rect* **) const;**

Parameters

rect
Refers to a source rectangle.

Remarks

Determines if *rect* is not equal to **CRect** by comparing the coordinates of their upper-left and lower-right corners.

Return Value

TRUE if not equal; otherwise **FALSE**.

See Also

CRect::operator ==

CRect::operator +=

Syntax

void operator +=(POINT *point* **);**

Parameters

point
Contains a **POINT** or **CPoint**.

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 added to **CRect**.

See Also

CRect::OffsetRect

CRect::operator –=

Syntax

void operator – =(POINT *point* **);**

Parameters

point
Contains a **POINT** or **CPoint**.

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 subtracted from **CRect**.

See Also

CRect::OffsetRect

CRect::operator &=

Syntax

void operator &=(const RECT& *rect* **);**

Parameters

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.

See Also

CRect::IntersectRect

CRect::operator |=

Syntax

void operator |=(const RECT& *rect* **);**

Parameters

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.

See Also **CRect::UnionRect**

CRect::operator +

Syntax **CRect operator +(POINT** *point* **) const;**

Parameters

point
Contains a **POINT** or **CPoint**.

Remarks Returns a new **CRect** that is equal to **CRect** displaced by *point.* Moves the **CRect** *point.x* units along the x-axis and *point.y* units along the y-axis. The *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 –

Syntax **CRect operator – (POINT** *point* **) const;**

Parameters

point
Contains a **POINT** or **CPoint**.

Remarks A new **CRect** that is equal to **CRect** displaced by *–point.* Moves the **Rect** *–point.x* units along the x-axis and *–point.y* units along the y-axis. The *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 &

Syntax

CRect operator &(const RECT& *rect2* **) const;**

Parameters

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.

See Also

CRect::IntersectRect

CRect::operator |

Syntax

CRect operator |(const RECT& *rect2* **) const;**

Parameters

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.

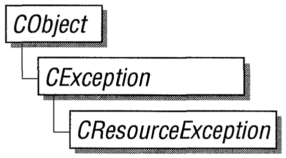
Windows ignores the dimensions of an empty rectangle, that is, a rectangle that has no height or has no width.

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>

Public Members

CResourceException	Constructs a **CResourceException** object.

Member Functions

CResourceException::CResourceException

Syntax

CResourceException();

Remarks

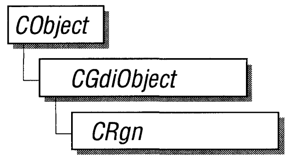
Constructs a **CResourceException** object.

Do not use this constructor directly, but rather call the global function **AfxThrowResourceException**.

See Also

Chapter 5, "Exception Processing," **AfxThrowResourceException**

class CRgn : public CGdiObject



The **CRgn** class encapsulates a Windows graphical design 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** in conjunction 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.

Public Members

Construction/Destruction

CRgn	Constructs a **CRgn** object.

Initialization

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	Initialize a **CRgn** object so that it is equivalent to the union of two specified **CRgn** objects.
CopyRgn	Initializes a **CRgn** object so that it is a copy of a specified **CRgn** object.

Operations

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

Syntax

int CombineRgn(CRgn* *pRgn1*, **CRgn*** *pRgn2*, **int** *nCombineMode* **);**

Parameters

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:

Value	Meaning
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.

Note 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:

Value	Meaning
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

Syntax

int CopyRgn(CRgn* *pRgnSrc* **);**

Parameters

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 **CombineRgn**.

Return Value

Specifies the type of the resulting region. It can be one of the following values:

Value	Meaning
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

Syntax

BOOL CreateEllipticRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

Parameters

x1
Specifies the x-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

y1
Specifies the y-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

x2
Specifies the x-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

y2
Specifies the 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.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreateEllipticRgnIndirect, **::CreateEllipticRgn**

CRgn::CreateEllipticRgnIndirect

Syntax

BOOL CreateEllipticRgnIndirect(LPRECT *lpRect* **);**

Parameters

lpRect
Points to a **RECT** structure or **CRect** object that contains the 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 *lpRect* and is stored in the **CRgn** object.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreateEllipticRgn, **::CreateEllipticRgnIndirect**

CRgn::CreatePolygonRgn

Syntax

BOOL CreatePolygonRgn(LPPOINT *lpPoints*, **int** *nCount*, **int** *wMode* **);**

Parameters

lpPoints
Points to an array of **POINT** structures or an array of **CPoint** objects. Each structure specifies the x- and y-coordinate of one vertex of the polygon.

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.

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.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreatePolyPolygonRgn, **::CreatePolygonRgn**

CRgn::CreatePolyPolygonRgn

Syntax

BOOL CreatePolyPolygonRgn(LPPOINT *lpPoints*, **LPINT** *lpPolyCounts*, **int** *nCount*, **int** *nPolyFillMode* **);**

Parameters

lpPoints
Points to an array of **POINT** structures or an array of **CPoint** objects that define the vertices of the polygons. Each polygon must be explicitly closed because the system does not close them automatically. The polygons are specified consecutively.

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.

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.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreatePolygonRgn, **CDC::SetPolyFillMode**, **::CreatePolyPolygonRgn**

CRgn::CreateRectRgn

Syntax

BOOL CreateRectRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

Parameters

x1
Specifies the x-coordinate of the upper-left corner of the region.

y1
Specifies the y-coordinate of the upper-left corner of the region.

x2
Specifies the x-coordinate of the lower-right corner of the region.

y2
Specifies the y-coordinate of the lower-right corner of the region.

Remarks

Creates a rectangular region that is stored in the **CRgn** object.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreateRectRgnIndirect, **CRgn::CreateRoundRectRgn**, **::CreateRectRgn**

CRgn::CreateRectRgnIndirect

Syntax

BOOL CreateRectRgnIndirect(LPRECT *lpRect* **);**

Parameters

lpRect
Points to a **RECT** structure or **CRect** object that contains the coordinates of the upper-left and lower-right corners of the region.

Remarks

Creates a rectangular region that is stored in the **CRgn** object.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreateRectRgn, **CRgn::CreateRoundRectRgn**, **::CreateRectRgnIndirect**

CRgn::CreateRoundRectRgn

Syntax

BOOL CreateRoundRectRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2*, **int** *x3*, **int** *y3* **);**

Parameters

x1
Specifies the x-coordinate of the upper-left corner of the region.

y1
Specifies the y-coordinate of the upper-left corner of the region.

x2
Specifies the x-coordinate of the lower-right corner of the region.

y2
Specifies the 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.

Return Value

TRUE if the operation succeeded; otherwise **FALSE**.

See Also

CRgn::CreateRectRgn, **CRgn::CreateRectRgnIndirect**, **::CreateRoundRectRgn**

CRgn::CRgn

Syntax

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

Syntax

BOOL EqualRgn(CRgn* *pRgn* **) const;**

Parameters

pRgn
Identifies a region.

Remarks

Checks the given region to determine whether it is equivalent to the region stored in the **CRgn** object.

Return Value

TRUE if the two regions are equivalent; otherwise **FALSE**.

See Also

::EqualRgn

CRgn::FromHandle

Syntax

static CRgn* FromHandle(HRGN *hRgn* **);**

Parameters

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

Syntax

int GetRgnBox(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to a **RECT** structure or **CRect** object to receive the coordinates of the bounding rectangle.

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:

Value	Meaning
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

Syntax

int OffsetRgn(int *x*, **int** *y* **);**

int OffsetRgn(POINT *point* **);**

Parameters

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.

Return Value

Specifies the new region's type. It can be any one of the following values:

Value	Meaning
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

Syntax

BOOL PtInRegion(int *x*, **int** *y* **) const;**

BOOL PtInRegion(POINT *point* **) const;**

Parameters

x
Specifies the x-coordinate of the point to test.

y
Specifies the y-coordinate of the point to test.

point
The x- and y-coordinate of *point* specify the x- and y-coordinate 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

TRUE if the point is in the region; otherwise **FALSE**.

See Also

::PtInRegion

CRgn::RectInRegion

Syntax

BOOL RectInRegion(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to a **RECT** structure or **CRect** object.

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

TRUE if any part of the specified rectangle lies within the boundaries of the region; otherwise **FALSE**.

See Also

::RectInRegion

CRgn::SetRectRgn

Syntax

void SetRectRgn(int *x1*, **int** *y1*, **int** *x2*, **int** *y2* **);**

void SetRectRgn(LPRECT *lpRect* **);**

Parameters

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

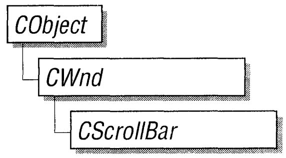
Creates a rectangular region. Unlike **CreateRectRgn**, however, it does not allocate any additional memory from the local Windows application heap. Instead, it uses the space allocated for the region stored in the **CRgn** object. This means that the **CRgn** object must already have been initialized 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** function to avoid calls to the local memory manager.

See Also

CRgn::CreateRectRgn, **::SetRectRgn**

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 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.

See Also **CWnd**, **CButton**, **CComboBox**, **CEdit**, **CListBox**, **CStatic**, **CModalDialog**, **CDialog**

Public Members

Construction/Destruction

CScrollBar	Constructs a **CScrollBar** object.

Initialization

Create	Creates the Windows scroll bar and attaches it to the **CScrollBar** object.

Operations

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.

Member Functions

CScrollBar::CScrollBar

Syntax

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::Create

Syntax

BOOL Create(DWORD *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* **);**

Parameters

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** or **CModalDialog** object. It must not be **NULL**.

nID
The scroll bar's resource 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:

Style	Application
WS_CHILD	Always.
WS_VISIBLE	Usually.
WS_DIABLED	Rarely.
WS_GROUP	To group controls.
WS_TABSTOP	To allow tabbing to reach this scroll bar control.

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

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 **Create**. 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.

Return Value

TRUE if successful; otherwise **FALSE**.

See Also

CScrollBar::CScrollBar

CScrollBar::GetScrollPos

Syntax

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.

See Also

CScrollBar::SetScrollPos, ::GetScrollPos

CScrollBar::GetScrollRange

Syntax

void GetScrollRange(LPINT *lpMinPos*, **LPINT** *lpMaxPos* **) const;**

Parameters

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::SetScrollPos

Syntax

int SetScrollPos(int *nPos*, **BOOL** *bRedraw* = **TRUE);**

Parameters

nPos
Specifies the new position for the scroll bar thumb. 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.

See Also

CScrollBar::GetScrollPos, ::SetScrollPos

CScrollBar::SetScrollRange

Syntax

void SetScrollRange(int *nMinPos*, **int** *nMaxPos*, **BOOL** *bRedraw* = **TRUE);**

Parameters

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 or show 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* to **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.

See Also

CScrollBar::SetScrollPos, CScrollBar::GetScrollRange, ::SetScrollRange

class CSize : public tagSIZE

The **CSize** class is similar to the Windows **SIZE** structure, which implements a relative coordinate or position.

A **SIZE** structure has the following form:

```
typedef struct tagSIZE {
   int cx;
   int cy;
} SIZE;
```

The **cx** and **cy** members of **CSize** are public. In addition, **CSize** implements member functions to manipulate the **SIZE** structure.

Because **CSize** derives from **tagSIZE**, **CSize** objects may be used as **SIZE** structures.

See Also **CRect**, **CPoint**

Public Members

Construction/Destruction

CSize	Constructs a **CSize** object.

Operations

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

operator +	Adds the two sizes.
operator –	Subtracts the two sizes.

Member Functions

CSize::CSize

Syntax

CSize();

CSize(int *initCX*, **int** *initCY* **);**

CSize(SIZE *initSize* **);**

CSize(POINT *initPt* **);**

Parameters

initCX
Sets the **cx** member for the **CSize**.

initCY
Sets the **cy** member for the **CSize**.

initSize
Windows **SIZE** structure used to initialize **CSize**.

initPt
Windows **POINT** structure used to initialize **CSize**.

Remarks

Constructs a **CSize** object. If no arguments are given, **cx** and **cy** members are not initialized.

Operators

CSize::operator ==

Syntax **BOOL operator ==(SIZE** *size* **) const;**

Parameters *size*
A Windows **SIZE** structure.

Remarks Checks for equality between two sizes.

Return Value **TRUE** if the sizes are equal; otherwise **FALSE**.

CSize::operator !=

Syntax **BOOL operator !=(SIZE** *size* **) const;**

Parameters *size*
A Windows **SIZE** structure.

Remarks Checks for inequality between two sizes.

Return Value **TRUE** if the sizes are not equal; otherwise **FALSE**.

CSize::operator +=

Syntax **void operator +=(SIZE** *size* **);**

Parameters *size*
A Windows **SIZE** structure.

Remarks Adds a size to a **CSize**.

CSize::operator –=

Syntax

void operator –=(SIZE *size* **);**

Parameters

size
A Windows **SIZE** structure.

Remarks

Subtracts a size from a **CSize**.

CSize::operator +

Syntax

CSize operator +(SIZE *size* **) const;**

Parameters

size
A Windows **SIZE** structure.

Return Value

Returns a **CSize** that is the sum of two sizes.

CSize::operator –

Syntax

CSize operator –(SIZE *size* **) const;**

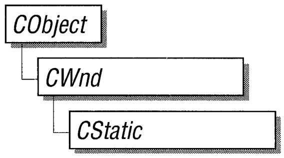
Parameters

size
A Windows **SIZE** structure.

Return Value

Returns a **CSize** that is the difference between two sizes.

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.

You 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 destructed 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.

See Also **CWnd**, **CButton**, **CComboBox**, **CEdit**, **CListBox**, **CScrollBar**, **CModalDialog**, **CDialog**

Public Members

Construction/Destruction

CStatic	Constructs a **CStatic** object.

Initialization

Create	Creates the Windows static control and attaches it to the **CStatic** object.

Member Functions

CStatic::CStatic

Syntax

CStatic();

Remarks

Constructs a **CStatic** object.

See Also

CStatic::Create

CStatic::Create

Syntax

BOOL Create(const char FAR* *lpText*, **DWORD** *dwStyle*, **const RECT&** *rect*, **CWnd*** *pParentWnd*, **UINT** *nID* = **0xffff);**

Parameters

lpText
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** or **CModalDialog** object. It must not be **NULL**.

nID
Specifies the static control's resource ID.

Remarks

You 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:

Style	Application
WS_CHILD	Always.
WS_VISIBLE	Usually.
WS_DIABLED	Rarely.

See **CreateEx** in the **CWnd** base class for a full description of these window styles.

Use any combination of the following static control styles for *dwStyle*:

Style	Meaning
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.

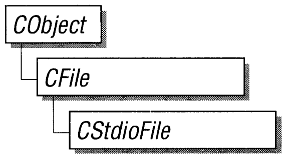
Style	Meaning
SS_LEFT	Designates a simple rectangle and displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
SS_LEFTNOWORDWRAP	Designates 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, Windows 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.

Style	Meaning
SS_WHITEFRAME	Specifies a box with a frame drawn with the same color as window backgrounds. The default is white.
SS_WHITERECT	Specifies a rectangle filled with the color used to fill window backgrounds. The default is white.

Return Value **TRUE** if successful; otherwise **FALSE**.

See Also **CStatic::CStatic**

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.

#include <afx.h>

Comments

The following **CFile** functions are not implemented for **CStdioFile**.

- **Duplicate**
- **LockRange**
- **UnlockRange**

If you call these functions on a **CStdioFile**, you will get a **CNotSupportedException**.

Public Members

Data Members

m_pStream	A data member containing a pointer to an open file.

Construction/Destruction

CStdioFile	Constructs a **CStdioFile** object from a path or file pointer.
~CStdioFile	Destroys the object and closes the file if it is open.

Text Read/Write

ReadString	Reads a single line of text.
WriteString	Writes a single line of text.

Member Functions

CStdioFile::CStdioFile

Syntax

CStdioFile();

CStdioFile(FILE* *pOpenStream* **);**

CStdioFile(const char ****pszFileName*, **UINT *nOpenFlags* **)**
throw(CFileException);

Parameters

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
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::~CStdioFile

Syntax

virtual ~CStdioFile();

Remarks

Usually, this destructor closes the operating-system file associated with the **CStdioFile** object. However, if the **CStdioFile** object was constructed by passing in a handle to an opened file using **CStdioFile(int)**, the operating-system file will not be closed. You must close the operating-system file yourself.

CStdioFile::ReadString

Syntax

virtual char FAR* ReadString(char FAR* *lpsz*, **UINT** *nMax* **)**
throw(CFileException);

Parameters

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

The buffer containing the text data.

Example

```
extern CStdioFile f;
char buf[100];

f.ReadString( buf, 100 );
```

CStdioFile::WriteString

Syntax

virtual void WriteString(const char FAR* *lpsz* **)**
throw(CFileException);

Parameters

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

Syntax

FILE* 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 (starting on page 598) offers useful information on:

- **CString** Exception Cleanup
- **CString** Argument Passing

Comments

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 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>

Public Members

Construction/Destruction

CString	Constructs **CString** objects in various ways.
~CString	Destroys a **CString** object.

The String as an Array

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.
SetAt	Sets a character at a given position.
operator const char* ()	Directly accesses characters stored in a **CString** object.

Assignment/Concatenation

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

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

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

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

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

operator <<	Inserts a **CString** object to an archive or dump context.
operator >>	Extracts a **CString** object from an archive.

Buffer Access

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

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

Syntax

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.

Example

```
CString s( '\\265' );  // Octal ANSI code for '1/2'
s.AnsiToOem();
ASSERT( s == "\\253" ); // Octal oem code for '1/2'
```

See Also

CString::OemToAnsi

CString::Collate

Syntax

int Collate(const char* *psz* **) const;**

Parameters

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

0	The strings are identical.
–1	This **CString** object is less than *psz*.
1	This **CString** object is greater than *psz*.

Example

```
CString s1( "abc" );
CString s2( "abd" );
ASSERT( s1.Collate( s2 ) == -1 );
```

See Also **CString::Compare**, **CString::CompareNoCase**

CString::Compare

Syntax **int Compare(const char*** *psz* **) const;**

Parameters

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

0	The strings are identical.
–1	This **CString** object is less than *psz*.
1	This **CString** object is greater than *psz*.

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
```

See Also **CString::CompareNoCase**, **CString::Collate**

CString::CompareNoCase

Syntax **int CompareNoCase(const char*** *psz* **) const;**

Parameters

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

0	The strings are identical (ignoring case).
–1	This **CString** object is less than *psz* (ignoring case).
1	This **CString** object is greater than *psz* (ignoring case).

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
```

See Also

CString::Compare, **CString::Collate**

CString::CString

Syntax

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);

Parameters

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.

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
```

See Also

CString::operator =, "CString Exception Cleanup" on page 598

CString::~CString

Syntax

~CString();

Remarks

This **CString** destructor releases allocated memory used to store the string's character data.

CString::Empty

Syntax

void Empty();

Remarks

Makes this **CString** object an empty string, and frees memory as appropriate.

Example

```
CString s1( "abc" );
CString s2;
s1.Empty();
ASSERT( s1 == s2 );
```

See Also

CString::IsEmpty, "CString Exception Cleanup" on page 598

CString::Find

Syntax

int Find(char *ch* **) const;**

int Find(const char* *pszSub* **) const;**

Parameters

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.

Example

```
CString s( "abcdef" );
ASSERT( s.Find( 'c' ) == 2 );
ASSERT( s.Find( "de" ) == 3);
```

See Also

CString::ReverseFind, CString::FindOneOf

CString::FindOneOf

Syntax

int FindOneOf(const char* *pszCharSet* **) const;**

Parameters

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.

Example

```
CString s( "abcdef" );
ASSERT( s.FindOneOf( "xd" ) == 3 ); // 'd' is first match
```

See Also

CString::Find

CString::GetAt

Syntax

char GetAt(int *nIndex* **) const;**

Parameters

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 **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.

Example

```
CString s( "abcdef" );
ASSERT( s.GetAt(2) == 'c' );
```

See Also

CString::SetAt, **CString::GetLength**, **CString::operator [ ]**

CString::GetBuffer

Syntax

char* GetBuffer(int *nMinBufLength* **)**
throw(CMemoryException);

Parameters

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 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.

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 than 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();
```

See Also **CString::GetBufferSetLength**, **CString::ReleaseBuffer**

CString::GetBufferSetLength

Syntax

char* GetBufferSetLength(int *nNewLength* **)**
throw(CMemoryException);

Parameters

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 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

Syntax **int GetLength() const;**

Remarks Returns a count of the characters in this **CString** object. The count does not include a null terminator.

Example

```
CString s( "abcdef" );
ASSERT( s.GetLength() == 6 );
```

See Also **CString::IsEmpty**

CString::IsEmpty

Syntax **BOOL IsEmpty() const;**

Remarks Tests a **CString** object for the empty condition.

Return Value **TRUE** if the **CString** object has 0 length; otherwise **FALSE**.

Example

```
CString s;
ASSERT( s.IsEmpty() );
```

See Also **CString::GetLength**

CString::Left

Syntax

CString Left(int *nCount* **) const**
throw(CMemoryException);

Parameters

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 The returned **CString** object may be empty.

Example

```
CString s( "abcdef" );
ASSERT( s.Left(3) == "abc" );
```

See Also **CString::Mid**, **CString::Right**

CString::LoadString

Syntax

BOOL LoadString(UINT *nID* **)**
throw(CMemoryException);

Parameters

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

Syntax

void MakeLower();

Remarks

Converts this **CString** object to a lowercase string.

Example

```
CString s( "ABC" );
s.MakeLower();
ASSERT( s == "abc" );
```

See Also

CString::MakeUpper

CString::MakeReverse

Syntax

void MakeReverse();

Remarks

Reverses the order of the characters in this **CString** object.

Example

```
CString s( "abc" );
s.MakeReverse();
ASSERT( s == "cba" );
```

CString::MakeUpper

Syntax

void MakeUpper();

Remarks

Converts this **CString** object to an uppercase string.

Example

```
CString s( "abc" );
s.MakeUpper();
ASSERT( s == "ABC" );
```

See Also

CString::MakeLower

CString::Mid

Syntax

CString Mid(int *nFirst* **) const**
throw(CMemoryException);

CString Mid(int *nFirst*, **int** *nCount* **) const**
throw(CMemoryException);

Parameters

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 The returned **CString** object may be empty.

Example

```
CString s( "abcdef" );
ASSERT( s.Mid( 2, 3 ) == "cde" );
```

See Also **CString::Left**, **CString::Right**

CString::OemToAnsi

Syntax **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.

Example

```
CString s( '\\253' );  // Octal oem code for '1/2'
s.OemToAnsi();
ASSERT( s == "\\265" ); // Octal ANSI code for '1/2'
```

See Also **CString::AnsiToOem**

CString::ReleaseBuffer

Syntax **void ReleaseBuffer(int** *nNewLength* **= –1);**

Parameters

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.

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
```

See Also **CString::GetBuffer**

CString::ReverseFind

Syntax **int ReverseFind(char** *ch* **) const;**

Parameters

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.

Example

```
CString s( "abcabc" );
ASSERT( s.ReverseFind( 'b' ) == 4 );
```

See Also **CString::Find**, **CString::FindOneOf**

CString::Right

Syntax **CString Right(int** *nCount* **) const**
throw(CMemoryException);

Parameters

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 The returned **CString** object may be empty.

Example

```
CString s( "abcdef" );
ASSERT( s.Right(3) == "def" );
```

See Also

CString::Mid, **CString::Left**

CString::SetAt

Syntax

void SetAt(int *nIndex*, **char** *ch* **);**

Parameters

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 **GetLength**. The Debug version of the Microsoft Foundation classes 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

Syntax

CString SpanExcluding(const char* *pszCharSet* **) const throw(CMemoryException);**

Parameters

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

Syntax

CString SpanIncluding(const char* *pszCharSet* **) const throw(CMemoryException);**

Parameters

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 =

Syntax

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.

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
```

See Also

CString::CString

CString::operator const char* ()

Syntax

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::operators <<, >>

Syntax

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** insert (<<) operator supports diagnostic dumping and storing to an archive. The extract (>>) 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 +

Syntax

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.

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
```

See Also

CString::operator +=

CString::operator +=

Syntax

void operator +=(const CString& *string* **)**
throw(CMemoryException);

void operator +=(char *ch* **)**
throw(CMemoryException);

void 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.

Example

```
CString s( "abc" );
ASSERT( ( s += "def" ) == "abcdef" );
```

See Also

CString::operator +

CString Comparison Operators

Syntax

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 [ ]

Syntax **char operator [ ](int** *nIndex* **) const;**

Remarks You can think of a **CString** object as an array of characters. The subscript ([ ]) operator returns a single character specified by the zero-base 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**.

Example

```
CString s( "abc" );
ASSERT( s[1] == 'b' );
```

See Also **CString::GetAt**, **CString::SetAt**

Application Notes

CString Exception Cleanup

Memory Leaks

If you notice that the Microsoft Foundation 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 isn't 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 **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&**).

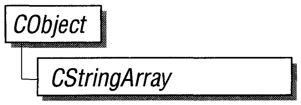
Strings as Function Outputs

Normally you can return **CString** objects from functions since **CStrings** 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**s 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 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>

Public Members

Construction/Destruction

CStringArray	Constructs an empty array for **CString** objects.
~CStringArray	Destroys a **CStringArray** object.

Bounds

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

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

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

SetAtGrow	Sets the value for a given index, growing the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

Insertion/Removal

InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator [ ]	Sets or gets the element at the specified index.

class CStringList : public CObject

CObject
CStringList

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**s is stored to an archive, either with the 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>

Public Members

Construction/Destruction

CStringList	Constructs an empty list for **CString** objects.
~CStringList	Destroys a **CStringList** object.

Head/Tail Access

GetHead	Returns the head element of the list (cannot be empty).
GetTail	Returns the tail element of the list (cannot be empty).

Operations

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

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

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

InsertBefore	Inserts a new element before a given position.
InsertAfter	Inserts a new element after a given position.

Searching

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

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.

See the *Run-Time Library Reference* for more information on the **time_t** data type and the run-time functions that are used by **CTime**.

A companion class, **CTimeSpan**, represents a time interval—the difference between two **CTime** objects.

#include <afx.h>

See Also Run-time functions: **asctime**, **_ftime**, **gmtime**, **localtime**, **strftime**, **time**

Derivation 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.

Public Members

Construction/Destruction

CTime	Constructs **CTime** objects in various ways.
GetCurrentTime	Creates a **CTime** object that represents the current time (static member function).

Extraction

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

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

operator =	Assigns new time values.
operator +, –	Adds and subtracts **CTimeSpan** and **CTime** objects.
operator +=, –=	Adds and subtracts a **CTimeSpan** object to and from this **CTime** object.
operator ==, < , etc.	Compare two absolute times.

Archive/Dump

operator <<	Outputs a **CTime** object to **CArchive** or **CDumpContext**.
operator >>	Inputs a **CTime** object from **CArchive**.

Member Functions

CTime::CTime

Syntax

CTime();

CTime(const CTime& *timeSrc* **);**

CTime(time_t *time* **);**

CTime(int *nYear*, **int** *nMonth*, **int** *nDay*, **int** *nHour*, **int** *nMin*, **int** *nSec* **);**

Parameters

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.

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 zero (illegal) value.

Note Zero is an invalid time. This constructor is provided to allow the definition of **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	1900–2036
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

Syntax

CString Format(const char* *pFormat* **);**

Parameters

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.

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
CString s = t.Format( "%A, %B %d, %Y" );
ASSERT( s == "Tuesday, March 19, 1999" );
```

See Also **CTime::FormatGmt**

CTime::FormatGmt

Syntax **CString FormatGmt(const char*** *pFormat* **);**

Parameters

pFormat
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.

Example See the example for **Format**.

See Also **CTime::Format**

CTime::GetCurrentTime

Syntax **static CTime GetCurrentTime();**

Remarks Returns a **CTime** object that represents the current time.

Example

```
CTime t = CTime::GetCurrentTime();
ASSERT( ( t.GetYear() >= 1999 ) && ( t.GetYear() <= 2000 ) );
```

CTime::GetDay

Syntax

int GetDay() const;

Remarks

Returns the day of the month, based on local time, in the range 1 through 31.

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 );
```

See Also

CTime::GetDayOfWeek

CTime::GetDayOfWeek

Syntax

int GetDayOfWeek() const;

Remarks

Returns the day of the week based on local time. 1 = Sunday, 2 = Monday, ... 7 = Saturday.

CTime::GetGmtTm

Syntax

struct tm* GetGmtTm(struct tm* *ptm* **= NULL) const;**

Parameters

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 are as follows:

Field	Value Stored
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 –1 to 136; the year in the **CTime** interface is in the range 1900 to 2036 (inclusive).

Example

See the example for **GetLocalTm**.

CTime::GetLocalTm

Syntax

struct tm* GetLocalTm(struct tm* *ptm* **) const;**

Parameters

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::GetHour

Syntax **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::GetMinute

Syntax **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

Syntax **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

Syntax

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

Syntax

time_t GetTime() const;

Remarks

Returns a **time_t** value for the given **CTime** object.

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 );
```

See Also

CTime constructors

CTime::GetYear

Syntax

int GetYear();

Remarks

Returns the year, based on local time, in the range 1900 to 2036.

Example

See the example for **GetDay**.

Operators

CTime::operator =

Syntax

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.

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
```

See Also

CTime constructors

CTime::operator +, –

Syntax

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 +=, −=

Syntax

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 ); // one hour exactly
ASSERT( t.GetHour() == 23 );
```

CTime Comparison Operators

Syntax

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 <<, >>

Syntax

friend CDumpContext& operator <<(CDumpContext& *dc*, **CTime** *time* **);**

friend CArchive& operator <<(CArchive& *ar*, **CTime** *time* **);**

friend CArchive& operator >>(CArchive& *ar*, **CTime&** *rtime* **);**

Remarks

The **CTime** insert (<<) operator supports diagnostic dumping and storing to an archive. The extract (>>) 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.

Example

```
CTime t( 1999, 3, 19, 22, 15, 0 ); // 10:15PM March 19, 1999
afxDump << t << "\\n"; // prints 'CTime("Tue Mar 19 22:15:00 1999")'

extern CArchive ar;
if( ar.IsLoading() )
  ar >> t;
else
  ar << t;
```

See Also

CArchive, **CDumpContext**

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 ± 68 years, approximately.

A companion class, **CTime**, represents an absolute time. A **CTimeSpan** is the difference between two **CTime**s.

#include <afx.h>

See Also Run-time functions: **asctime**, **_ftime**, **gmtime**, **localtime**, **strftime**, **time**

Derivation 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.

Public Members

Construction/Destruction

CTimeSpan	Constructs **CTimeSpan** objects in various ways.

Extraction

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

Format	Converts a **CTimeSpan** into a formatted string.

Operators

operator =	Assigns new time-span values.
operator +, –	Adds and subtracts **CTimeSpan** objects.
operator +=, –=	Adds and subtracts a **CTimeSpan** object to and from this **CTimeSpan**.
operator ==, <, etc.	Compare two relative time values.

Archive/Dump

operator <<	Outputs a **CTimeSpan** object to **CArchive** or **CDumpContext**.
operator >>	Inputs a **CTimeSpan** object from **CArchive**.

Member Functions

CTimeSpan::CTimeSpan

Syntax

CTimeSpan();

CTimeSpan(const CTimeSpan& *timeSpanSrc* **);**

CTimeSpan(time_t *time* **);**

CTimeSpan(LONG *lDays*, **int** *nHours*, **int** *nMins*, **int** *nSecs* **);**

Parameters

timeSpanSrc
Indicates a **CTimeSpan** object that already exists.

time
Indicates a **time_t** time value.

lDays, *nHours*, *nMins*, *nSecs*
Indicate days, hours, minutes, and seconds.

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 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

Syntax

CString Format(const char* *pFormat* **);**

Parameters

pFormat
Indicates 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 formatting codes for **Format** are listed below:

Value	Meaning
%**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, min: %M, sec: %S" );
ASSERT( s == "Total days: 3, hours: 01, min: 05, sec: 12" );
```

CTimeSpan::GetDays

Syntax **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

Syntax **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

Syntax

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

Syntax

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

Syntax

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

Syntax **LONG GetTotalMinutes() const;**

Remarks Returns the total number of complete minutes in this **CTimeSpan**.

Example See the example for **GetTotalHours**.

CTimeSpan::GetTotalSeconds

Syntax **LONG GetTotalSeconds() const;**

Remarks Returns the total number of complete seconds in this **CTimeSpan**.

Example See the example for **GetTotalHours**.

Operators

CTimeSpan::operator =

Syntax

const CTimeSpan& operator =(const CTimeSpan& *timeSpanSrc* **);**

Remarks

The overloaded assignment operator copies the source **CTimeSpan** *timeSpanSrc* object into this **CTimeSpan** object.

Example

```
CTimeSpan ts1;
CTimeSpan ts2( 3, 1, 5, 12 ); // 3 days, 1 hour, 5 min, and 12 sec
ts1 = ts2;
ASSERT( ts1 == ts2 );
```

See Also

CTimeSpan constructors

CTimeSpan::operator +, –

Syntax

CTimeSpan operator –(CTimeSpan *timeSpan* **) const;**

CTimeSpan operator +(CTimeSpan *timeSpan* **) const;**

Remarks

These two operators allow you to add and subtract **CTimeSpan**s 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 min, 52 sec
```

CTimeSpan::operator +=, –=

Syntax

const CTimeSpan& operator +=(CTimeSpan *timeSpan* **);**

const CTimeSpan& operator –=(CTimeSpan *timeSpan* **);**

Remarks

These operators allow you to add and subtract a **CTimeSpan** 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

Syntax

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 <<, >>

Syntax

friend CDumpContext& operator <<(CDumpContext& *dc*, **CTimeSpan** *timeSpan* **);**

friend CArchive& operator <<(CArchive& *ar*, **CTimeSpan** *timeSpan* **);**

friend CArchive& operator >>(CArchive& *ar*, **CTimeSpan&** *rtimeSpan* **);**

Remarks

The **CTimeSpan** insert (<<) operator supports diagnostic dumping and storing to an archive. The extract (>>) operator supports loading from an archive.

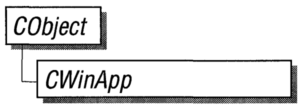
When you send a **CTimeSpan** object to the dump context, the value is displayed in a human-readable format that shows days, hours, minutes, and seconds.

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 CWinApp : public CObject

Use the **CWinApp** class to create a Windows application object. An application object provides member functions for initializing the application (and each instance of it) and for running the application.



Each application that uses the Foundation classes can only contain one **CWinApp** object. 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 Foundation class library. Declare your **CWinApp** object at the global level or statically.

Typically, you will derive an application class from **CWinApp** and override the **InitInstance** member function to create your application's main window object.

The Microsoft Foundation Class Library provides a number of global functions that you can use to access your **CWinApp** object, as shown by the following list. These functions are available from C or C++.

Function	Purpose
AfxGetApp	Obtain a pointer to the **CWinApp** object.
AfxGetInstanceHandle	Obtain a handle to the current application instance.
AfxGetResourceHandle	Obtain a handle to the application's resources.
AfxGetAppName	Obtain a pointer to a string containing the application's name.

Note Most users of the Foundation classes will use the default **WinMain** provided by the Foundation class library. If you provide your own version of **WinMain**, you must call **AfxWinInit** before using a Foundation class to attach your **CWinApp** object to the Microsoft Foundation Class Library.

Public Members

Data Members

m_pszAppName	Specifies the name of the application.
m_hInstance	Corresponds to the *hInstance* parameter passed by Windows to **WinMain**.
m_hPrevInstance	Corresponds to the *hPrevInstance* parameter passed by Windows to **WinMain**.
m_lpCmdLine	Corresponds to the *lpCmdLine* parameter passed by Windows to **WinMain**.
m_nCmdShow	Corresponds to the *nCmdShow* parameter passed by Windows to **WinMain**.
m_pMainWnd	Holds a pointer to the application's main window. See **InitInstance** for an example of how to initialize **m_pMainWnd**.

Construction/Destruction

CWinApp	Constructs a **CWinApp** object.

Operations

LoadCursor	Loads a cursor resource, or a resource handle if the resource has been loaded previously.
LoadStandardCursor	Loads a Windows predefined cursor specified by **IDC_** constants from WINDOWS.H.
LoadOEMCursor	Loads a Windows OEM predefined cursor specified by **OCR_** constants from WINDOWS.H.
LoadIcon	Loads an icon resource, or a resource handle if the resource has been loaded previously.
LoadStandardIcon	Loads a Windows predefined icon specified by **IDI_** constants from WINDOWS.H.
LoadOEMIcon	Loads a Windows OEM predefined icon specified by **OIC_** constants from WINDOWS.H.

Overridables

InitApplication	Performs any application-level initialization when overridden. (Sometimes override.)
InitInstance	Performs Windows instance initialization, such as creating your window objects when overridden. (Always override.)
Run	Runs the default message loop. Override to customize the message loop. (Seldom override.)
OnIdle	Overrride to perform application-specific idle-time processing. (Sometimes override.)
ExitInstance	Override to clean up when your application terminates. (Sometimes override.)
PreTranslateMessage	Filters messages before they are dispatched to the Windows functions **TranslateMessage** and **DispatchMessage**. (Seldom override.)

Protected Members

m_msgCur	Specifies the last window message retrieved by **Run**; only useful if the message is currently being processed.

Member Functions

CWinApp::CWinApp

Syntax

CWinApp(const char* *pszAppName* **= NULL);**

Parameters

pszAppName
A null-terminated string containing the Windows application name for your application. If this argument is not supplied or is **NULL**, **CWinApp** uses the filename of the executable file by default.

Remarks

Constructs a **CWinApp** object and passes *pszAppName* to be stored as the application name. This constructor is invoked by your global application object definition. You can have only one **CWinApp** object in your application. The constructor stores a pointer to the **CWinApp** object in a Microsoft Foundation-defined global variable. This is so the **WinMain** can call the object's member functions to initialize and run the application.

CWinApp::ExitInstance

Syntax

virtual int ExitInstance();

Remarks

Override this function to clean up when your application terminates.

ExitInstance is called by the default **Run** member function.

Return Value

The application's exit code, where 0 indicates no errors and values greater than 0 indicate an error. This value is used as the return value from **WinMain**.

CWinApp::InitApplication

Syntax

virtual BOOL InitApplication();

Remarks

Windows allows several copies of the same program to be running at the same time. Thus, 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. This function is called by the Foundation library version of **WinMain**. Override **InitApplication** if your application needs one-time initialization such as Windows class registration.

Return Value **TRUE** if initialization is successful; otherwise **FALSE**.

See Also **CWinApp::InitInstance**

CWinApp::InitInstance

Syntax **virtual BOOL InitInstance();**

Remarks Windows allows several copies of the same program to be running at the same time. Thus, 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. This function is called by the Foundation library implementation of **WinMain**. Override **InitInstance** to provide initialization for each new instance of your application running under Windows. Typically, you override **InitInstance** to construct your main window object and set **m_pMainWnd** to point to that window, as shown here.

```
BOOL CDerivedApp::InitInstance()
{
    m_pMainWnd = new CDerivedWindow();
    m_pMainWnd->ShowWindow( m_nCmdShow );
    m_pMainWnd->UpdateWindow();

    return TRUE;
}
```

Return Value **TRUE** if initialization is successful; otherwise **FALSE**.

See Also **CWinApp::InitApplication**

CWinApp::LoadCursor

Syntax

HCURSOR LoadCursor(LPSTR *lpCursorName* **);**

HCURSOR LoadCursor(UINT *nIDCursor* **);**

Parameters

lpCursorName
Points to a null-terminated string that contains the name of the cursor resource. You can use a **CString** in place of an **LPSTR**.

nIDCursor
ID number of the resource.

Remarks

Loads the cursor resource named by *lpCursorName* or specified by *nIDCursor* from the current executable file. **LoadCursor** loads the cursor into memory only if it has not been previously loaded.

Use the **LoadStandardCursor** or **LoadOEMCursor** member functions to access the predefined Windows cursors.

Return Value

A handle to a cursor resource. If unsuccessful, returns **NULL**.

See Also

CWinApp::LoadStandardCursor, **CWinApp::LoadOEMCursor**, **::LoadCursor**

CWinApp::LoadIcon

Syntax

HICON LoadIcon(LPSTR *lpIconName* **);**

HICON LoadIcon(UINT *nIDIcon* **);**

Parameters

lpIconName
Points to a null-terminated string that contains the name of the icon resource. You can also use a **CString** for this argument.

nIDIcon
ID number of the resource.

Remarks

Loads the icon resource named by *lpIconName* or specified by *nIDIcon* from the executable file. **LoadIcon** loads the icon only if it has not been previously loaded.

You can use the **LoadStandardIcon** or **LoadOEMIcon** member functions to access the predefined Windows icons.

Return Value

A handle to an icon resource. If unsuccessful, returns **NULL**.

See Also

CWinApp::LoadStandardIcon, **CWinApp::LoadOEMIcon**, **::LoadIcon**

CWinApp::LoadOEMCursor

Syntax

HCURSOR LoadOEMCursor(UINT *nIDCursor* **);**

Parameters

nIDCursor
An **OCR_** manifest constant identifier that specifies a predefined Windows cursor. You must have **#define OEMRESOURCE** in your source file to get access to the **OCR_** constants in WINDOWS.H.

Remarks

Loads the Windows predefined cursor resource specified by *nIDCursor*.

Use **LoadOEMCursor** or the **LoadStandardCursor** member function to access the predefined Windows cursors.

Return Value

A handle to a cursor resource. If unsuccessful, returns **NULL**.

See Also

CWinApp::LoadCursor, **CWinApp::LoadStandardCursor**, **::LoadCursor**

CWinApp::LoadOEMIcon

Syntax

HICON LoadOEMIcon(UINT *nIDIcon* **);**

Parameters

nIDIcon
An **OIC_** manifest constant identifier that specifies a predefined Windows icon. You must have **#define OEMRESOURCE** in your source file to get access to the **OIC_** constants in WINDOWS.H.

Remarks

Loads the Windows predefined icon resource specified by *nIDIcon.*

Use **LoadOEMIcon** or the **LoadStandardIcon** member function to access the predefined Windows icons.

Return Value

A handle to an icon resource. If unsuccessful, returns **NULL**.

See Also

CWinApp::LoadStandardIcon, **CWinApp::LoadIcon**, **::LoadIcon**

CWinApp::LoadStandardCursor

Syntax

HCURSOR LoadStandardCursor(LPSTR *lpCursorName* **);**

Parameters

lpCursorName
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 for *lpCursorName*:

Value	Meaning
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 pointing straight up
IDC_SIZE	Cursor to use when resizing a window
IDC_ICON	Cursor to use when dragging 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-header arrow
IDC_SIZENS	Vertical two-headed arrow

Remarks

Loads the Windows predefined cursor resource specified by *lpCursorName.*

Use **LoadStandardCursor** or the **LoadOEMCursor** member function to access the predefined Windows cursors.

Return Value

A handle to a cursor resource. If unsuccessful, returns **NULL.**

See Also

CWinApp::LoadOEMCursor, CWinApp::LoadCursor, ::LoadCursor

CWinApp::LoadStandardIcon

Syntax

HICON LoadStandardIcon(LPSTR *lpIconName* **);**

Parameters

lpIconName
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 for *lpIconName*:

Value	Meaning
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 specified by *lpIconName.*

Use **LoadStandardIcon** or the **LoadOEMIcon** member function to access the predefined icons used by Windows.

Return Value

A handle to an icon resource (a bitmap defining the icon). If unsuccessful, returns **NULL**.

See Also

CWinApp::LoadOEMIcon, **CWinApp::LoadIcon**, **::LoadIcon**

CWinApp::OnIdle

Syntax

virtual BOOL OnIdle(LONG *lCount* **);**

Parameters

lCount
A counter incremented each time **GetMessage** finds the message queue empty. This count is reset to 0 each time a new message is processed. *lCount* can be used to determine relatively how long the application has been idling without processing a message.

Remarks

Override this member function to perform idle-time processing. **OnIdle** is called when the application's message queue is empty. Use your override to call your own idle-handler members for such tasks as background recalculation in a spreadsheet, background repagination in a word processor, file backup, and the like.

The *lCount* parameter is incremented each time **GetMessage** finds the queue empty and reset to 0 each time a new message is processed. You can call your different idle routines based on this count.

Do not perform lengthy tasks during **OnIdle** because your application cannot process user input until **OnIdle** returns.

Note The default implementation of **OnIdle** performs internal data structure cleanup. Therefore, if you override **OnIdle**, you must explicitly call **CWinApp::OnIdle** in your overridden version to get the default processing.

Return Value

TRUE to receive more idle processing time; **FALSE** if no more idle time is needed.

CWinApp::PreTranslateMessage

Syntax

virtual BOOL PreTranslateMessage(MSG* *pMsg* **);**

Parameters

pMsg
A pointer to a **MSG** structure containing 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 access-key translation, so you must call **PreTranslateMessage** in your overridden version.

Return Value

TRUE if the message was fully processed in **PreTranslateMessage** and should not be passed to the Windows functions **TranslateMessage** and **DispatchMessage**. **FALSE** if the message should be processed in the normal way.

CWinApp::Run

Syntax

virtual int Run();

Remarks

Provides a default message loop. **Run** acquires and dispatches Windows messages until a **WM_QUIT** message is received. If the application's message queue currently contains no messages, **Run** calls **OnIdle** to perform idle-time processing. Incoming messages are passed to **PreTranslateMessage** for special processing, then passed to the Windows function **TranslateMessage** for standard keyboard translation, and finally **DispatchMessage** 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**.

Data Members

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 **AfxGetInstanceHandle**.

CWinApp::m_hPrevInstance

Remarks

Corresponds to the *hPrevInstance* parameter passed by Windows to **WinMain**. The **m_hPrevInstance** data member has the value **NULL** if this is the first instance of the application that is running.

CWinApp::m_lpCmdLine

Remarks

Corresponds to the *lpCmdLine* parameter passed by Windows to **WinMain**. Use **m_lpCmdLine** to access any command-line arguments entered when the application was started.

CWinApp::m_msgCur

Remarks

Corresponds to the last window message retrieved by **Run**; only useful if the message is currently being processed.

CWinApp::m_nCmdShow

Remarks

Corresponds to the *nCmdShow* parameter passed by Windows to **WinMain**. If **m_nCmdShow** is **TRUE**, the first call to **CWnd::ShowWindow** makes the main window visible. You can pass **m_nCmdShow** as an argument when you call **ShowWindow** for your application's main window.

CWinApp::m_pMainWnd

Remarks

Use this data member to store a pointer to your application's main window object. The 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, you must explicitly call the Windows function **PostQuitMessage** to terminate your application.

CWinApp::m_pszAppName

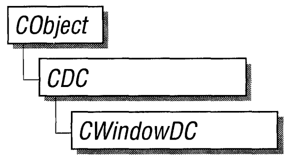
Remarks

Specifies the name of the application (optionally provided to the constructor or extracted from .EXE name if not provided).

Returned by **AfxGetAppName**.

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.



See Also **CDC**

Public Members

Construction/Destruction

CWindowDC	Constructs a **CWindowDC** object.
~CWindowDC	Destroys the **CWindowDC** object.

Protected Members

m_hWnd	The **HWND** to which this **CWindowDC** is attached.

Member Functions

CWindowDC::CWindowDC

Syntax

CWindowDC(CWnd* *pWnd* **)**
throw(CResourceException);

Parameters

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**

CWindowDC::~CWindowDC

Syntax

virtual ~CWindowDC();

Remarks

Destroys a **CWindowDC** object and calls the Windows **ReleaseDC** function.

See Also

CDC, **CClientDC**, **CWnd**, **::ReleaseDC**

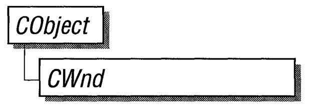
Data Members

CWindowDC::m_hWnd

Remarks

The **HWND** of the **CWnd** pointer used to construct the **CWindowDC** object.

class CWnd : public CObject



The **CWnd** class provides the base functionality of all window classes in the Microsoft Foundation Class Library.

A **CWnd** object is actually 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 the **CreateEx** member function and destroyed by the **CWnd** virtual destructor. The **DestroyWindow** function, one of the few public virtual **CWnd** member functions, 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 the **On***Message* member function to handle that member's particular message in your derived classes.

The **CWnd** class also provides the functionality of a Windows child window.

To create a useful child window for your application, derive a class from **CWnd**. 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.

Construction can be a one-step process in a derived class. Write a constructor for the derived class and call **Create** from within the constructor.

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. If you allocate any memory in the **CWnd** object, override the **CWnd** destructor to dispose of the allocations.

Within the Microsoft Foundation Class Library, further classes are derived from **CWnd** to provide specific window types. Three of these classes, **CFrameWnd**, **CMDIFrameWnd**, and **CMDIChildWnd**, contain further window functionality and are designed for further derivation. The control classes derived from **CWnd**, such as **CDialog** and **CButton**, can be used directly, or can also be used for further derivation of classes.

See Also **CDialog**, **CModalDialog**, **CStatic**, **CButton**, **CEdit**, **CListBox**, **CComboBox**, **CScrollBar**, **CFrameWnd**, **CMDIFrameWnd**, **CMDIChildWnd**

Public Members

Data Members

m_hWnd	Indicates the **HWND** attached to this **CWnd**.
wndTop	Indicates a static **CWnd** to use with the **SetWindowPos** member function to indicate that **CWnd** should be moved to the top of the window list.
wndBottom	Indicates a static **CWnd** to use with the **SetWindowPos** member function to indicate that **CWnd** should be moved to the bottom of the window list.

Construction/Destruction

CWnd	Constructs a **CWnd** object.
~CWnd	Destroys a **CWnd** object and destroys the attached window.
DestroyWindow	Destroys the attached Windows window.

Initialization

Create	Creates and initializes the child window associated with the **CWnd** object.
GetStyle	Returns the current window style.
Attach	Attaches a Windows handle to a **CWnd** object.
Detach	Detaches a Windows handle from a **CWnd** object and returns the handle.
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.
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 **this** is **NULL**.

Message 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 **CWnd** object's application queue, then returns without waiting for the object to process the message.

Window Text Functions

SetWindowText	Sets the **CWnd** text or caption title (if one exists) to the specified text.
GetWindowText	Copies the **CWnd** text or caption title (if it has one) into a buffer.
GetWindowTextLength	Returns the length of the **CWnd** text or caption title.
SetFont	Sets the current font.
GetFont	Retrieves the current font.

CMenu Functions

GetMenu	Retrieves a pointer to the 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.

Child Window Attributes

GetDlgCtrlID	If the **CWnd** is a child window, calling this function returns its ID value.

Window Size and Position

CloseWindow	Minimizes **CWnd**.
OpenIcon	Activates and restores a minimized (iconic) **CWnd**.
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.

Coordinate Mapping Functions

ClientToScreen	Converts the client coordinates of a given **point** or **rect** on the display to screen coordinates.
ScreenToClient	Converts the screen coordinates of a given **point** or **rect** on the display to client coordinates.

Update/Painting Functions

BeginPaint	Prepares **CWnd** for painting and fills a **PAINTSTRUCT** data structure with information about the painting.
EndPaint	Marks the end of painting.
GetDC	Retrieves a display context for the client area.
GetWindowDC	Retrieves the display context for the whole window, including the caption bar, menus, and scroll bars.
ReleaseDC	Releases common 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 update region.
InvalidateRgn	Invalidates the client area within the given region by adding it to the current update region.
ValidateRect	Validates the client area within the given rectangle by removing the rectangle from the update region.
ValidateRgn	Validates the client area within the given region by removing the region from the current update region.
ShowWindow	Shows or hides **CWnd**.
IsWindowVisible	Determines if the window is visible.
ShowOwnedPopups	Shows or hides all pop-up windows associated with the window.

Timer Functions

SetTimer	Installs a system timer that sends a **WM_TIMER** message when triggered.
KillTimer	Kills a system timer.

Window State Functions

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	Assigns the input focus.
SetSysModalWindow	Makes **CWnd** a system-modal window.
GetSysModalWindow	Retrieves the system-modal **CWnd** if there is one.
GetDesktopWindow	Retrieves the Windows desktop window.

Dialog-Box Item Functions

CheckDlgButton	Places a check mark next to or removes a check mark from a button control, or dims the button.
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.
GetDlgItem	Retrieves the handle of a control contained in the specified dialog 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 previous (or next) control within a group of controls.
GetNextDlgTabItem	Retrieves the first control that has the **WS_TABSTOP** style and precedes (or follows) the specified control.
IsDlgButtonChecked	Determines whether a button control has a check mark next to it, and whether a three-state button control is dimmed, checked, or neither.
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.

Scrolling Functions

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 **CWnd**.
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.

Window Access Functions

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 class.
GetNextWindow	Searches for the next (or previous) window in the window-manager's list.
GetTopWindow	Searches for a top-level child window that belongs to the **CWnd**.
GetWindow	Searches for the specified window from the window-manager's list.
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.

Alert Functions

FlashWindow	Flashes the window once.
MessageBox	Creates and displays a window that contains an application-supplied message and caption.

Clipboard Functions

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 **CloseClipboard** Windows function is called.
GetClipboardOwner	Retrieves a pointer to the current owner of the Clipboard.
GetClipboardViewer	Retrieves a pointer to the first window in the chain of Clipboard viewers.

Caret Functions

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.

Message Handlers

(require entry in Message Map)

OnCommand	Called when the user selects an item from a menu, when a control calls its parent's message handler, or when an access keystroke is translated.
OnActivate	Called when **CWnd** is being activated or deactivated.
OnActivateApp	Called when **CWnd** is about to be activated and **CWnd** belongs to a different task than the currently active window.
OnCancelMode	Called to allow **CWnd** to cancel any internal modes, such as mouse capture, if **CWnd** has the focus when a dialog box or message box is displayed.
OnChildActivate	Called whenever the size or position of **CWnd** changes if **CWnd** is a child window.
OnClose	Called as a signal that **CWnd** or an application will terminate.
OnCreate	Called when an application requests that **CWnd** be created.
OnCtlColor	Called when the control or message box is about to be drawn if **CWnd** is the parent of a system-defined control class or a message box.
OnDestroy	Called when **CWnd** is being destroyed.
OnEnable	Called when an application changes the enabled state of **CWnd**.
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 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 being minimized (made 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 **CWnd** has been changed.
OnPaint	Called when Windows or an application makes a request to repaint a portion of the window.
OnPaintIcon	Called when **CWnd** is minimized (iconic) and the icon is to be painted.
OnParentNotify	Called when a **CWnd** 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 (if **CWnd** does not have an icon defined for its class).
OnQueryEndSession	Called when the user chooses to end the Windows session, or when an application calls the **ExitWindows** Windows function.
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.

Nonclient-Area Functions

OnNcActivate	Called when the nonclient area needs to be changed to indicate an active or inactive state.
OnNcCalcSize	Called when the size of the client area needs 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 middle 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**.

System Message Handlers

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.
OnPaletteChanged	Called to allow windows that don't have the input focus and use a color palette to realize their logical palettes and update their client areas.
OnSpoolerStatus	Called from Print Manager whenever a job is added to or removed from the Print Manager queue.
OnSysColorChange	Called for all top-level windows when a change is made in the system color setting.
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.

Input Message Handlers

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. 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.
OnKeyUp	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 **CWnd** has the input focus.

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.

Initialization Message Handlers

OnInitMenu	Called when a menu is about to become active.
OnInitMenuPopup	Called when a pop-up menu is about to become active.

Clipboard Message Handlers

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 **EmptyClipboard** Windows function.
OnDrawClipboard	Called when the contents of the Clipboard change.
OnHScrollClipboard	Called when a Clipboard owner will 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.

Control Message Handlers

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 sorted owner-draw combo or list box.
OnDeleteItem	Called when an owner-draw list box or combo box is destroyed or when items are removed from the control by calls to **CComboBox::DeleteString** or **CComboBox::ResetContent**.
OnDrawItem	Called when a visual aspect of an owner-draw 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, although Windows normally handles this input.

OnMeasureItem	Called for an owner-draw button, 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.

MDI Message Handlers

OnMDIActivate	Called when an MDI child window is activated or deactivated.

Protected Members

Initialization

CreateEx	Creates a Windows overlapped, pop-up, or child window and attaches it to a **CWnd** object.

Operations

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.
GetSuperWndProcAddr	Accesses the original **WndProc** address of a subclassed window, and is used for translating Windows messages in the main message handler.
PreTranslateMessage	Used by **CWinApp** to filter window messages before they are dispatched to **TranslateMessage** and **DispatchMessage**.
WindowProc	Provides a window procedure for a **CWnd**. The default dispatches messages through the message map.
Default	Calls the default window procedure, which provides default processing for any window messages that an application does not process.
DefWindowProc	Calls the default window procedure, which provides default processing for any window messages that an application does not process.

Member Functions

CWnd::ArrangeIconicWindows

Syntax

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, or 0 if there were no icons.

See Also

CWnd::GetDesktopWindow, **CMDIFrameWnd::MDIIconArrange**, **::ArrangeIconicWindows**

CWnd::Attach

Syntax

BOOL Attach(HWND *hWndNew* **);**

Parameters

hWndNew
Specifies a handle to a Windows window.

Remarks

Attaches a Windows window to a **CWnd** object.

Return Value

TRUE if the operation was successful; otherwise **FALSE**.

See Also

CWnd::Detach, **CWnd::~CWnd**, **CWnd::m_hWnd**

CWnd::BeginPaint

Syntax

CDC* BeginPaint(LPPAINTSTRUCT *lpPaint* **);**

Parameters

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 sizing, moving, creating, scrolling, or 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

Syntax **void BringWindowToTop();**

Remarks Brings **CWnd** to the top of a stack of overlapping windows. In addition, **BringWindowToTop** activates pop-up and top-level windows. The **BringWindowToTop** member function should be used to uncover any window that is partially or completely obscured by any overlapping windows.

See Also **::BringWindowToTop**

CWnd::ChangeClipboardChain

Syntax **BOOL ChangeClipboardChain(HWND** *hWndNext* **);**

Parameters

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 **TRUE** if **CWnd** is removed; otherwise **FALSE**.

See Also **CWnd::SetClipboardViewer**, **::ChangeClipboardChain**

CWnd::CheckDlgButton

Syntax **void CheckDlgButton(int** *nIDButton*, **UINT** *nCheck* **);**

Parameters

nIDButton
Specifies the button control 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 is dimmed; if *nCheck* is 1, it is checked; if *nCheck* is 0, the check mark is removed.

Remarks

Places a check mark next to or removes a check mark from a button control, or, for a three-state button, may dim the button.

See Also

CWnd::IsDlgButtonChecked, **::CheckDlgButton**

CWnd::CheckRadioButton

Syntax

void CheckRadioButton(int *nIDFirstButton*, **int** *nIDLastButton*,
  int *nIDCheckButton* **);**

Parameters

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

Checks the radio button specified by *nIDCheckButton* and removes the check mark from all other radio buttons in the group of buttons specified by *nIDFirstButton* and *nIDLastButton.* Checking a radio button turns the radio button on or off.

See Also

CWnd::GetCheckedRadioButton, **::CheckRadioButton**

CWnd::ChildWindowFromPoint

Syntax

CWnd* ChildWindowFromPoint(POINT *point* **) const;**

Parameters

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.

The **CWnd*** that is returned may be temporary, and should not be stored beyond its immediate use.

See Also **CWnd::WindowFromPoint**, **::ChildWindowFromPoint**

CWnd::ClientToScreen

Syntax

void ClientToScreen(LPPOINT *lpPoint* **) const;**

void ClientToScreen(LPRECT *lpRect* **) const;**

Parameters

lpPoint
Points to a **POINT** structure or **CPoint** that contains the client coordinates to be converted.

lpRect
Points to a **RECT** structure or **CRect** 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 **CPoint** or **CRect** 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::CloseWindow

Syntax

void CloseWindow();

Remarks

Minimizes **CWnd**. If **CWnd** is an overlapped window, it is minimized by removing the client area and caption of the open window from the display screen and moving its icon into the icon area of the screen.

This member function has no effect if **CWnd** is a pop-up or child window.

See Also

CWnd::OpenIcon, **::CloseWindow**

CWnd::Create

Syntax

BOOL Create(const char FAR* *lpClassName***,**
const char FAR* *lpWindowName***, DWORD** *dwStyle***, const RECT&** *rect***,**
const CWnd* *pParentWnd***, UINT** *nID* **);**

Parameters

lpClassName
Points to a null-terminated character string that names the Windows class (a **WNDCLASS struct**). The class name can be any name registered with the **AfxRegisterWndClass** function or any of the predefined control-class names. If **NULL**, uses the default **CWnd** attributes. See **CreateEx** for a description of the possible values.

lpWindowName
Points to a null-terminated character string that contains the window name.

dwStyle
Specifies the window style attributes. See **CreateEx** 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.

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

TRUE if initialization is successful; otherwise **FALSE**.

See Also

CWnd::CWnd, **CWnd::CreateEx**

CWnd::CreateCaret

Syntax

void CreateCaret(CBitmap* *pBitmap* **);**

Parameters

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 using the **CBitmap::CreateBitmap** member function, **CreateDIBitmap** Windows function, or the **CBitmap::LoadBitmap** 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 losing the input focus or becoming inactive.

See Also

CBitmap::CreateBitmap, **::CreateDIBitmap**, **::DestroyCaret**, **CBitmap::LoadBitmap**, **CWnd::ShowCaret**, **::CreateCaret**

CWnd::CreateEx

Syntax

Protected:
BOOL CreateEx(DWORD *dwExStyle*, **const char FAR*** *lpClassName*, **const char FAR*** *lpWindowName*, **DWORD** *dwStyle*, **int** *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*, **HWND** *hwndParent*, **HMENU** *nIDorHMenu* **);**

Parameters

dwExStyle
Specifies the extended style of the **CWnd** being created. It may be one of the following values:

Style	Meaning
WS_EX_DLGMODALFRAME	Designates a window with a double border that may optionally be created with a title bar by specifying the **WS_CAPTION** style flag in *dwStyle*.
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.
WS_EX_TOPMOST	Specifies that a window created with this style should be placed above all nontopmost windows and stay above them even when **CWnd** is deactivated. An application can use the **SetWindowPos** member function to add or remove this attribute.

lpClassName
Points to a null-terminated character string that names the Windows class (a **WNDCLASS struct**). The class name can be any name registered with the **AfxRegisterWndClass** function or any of the predefined control-class names. It must not be **NULL**.

lpWindowName
Points to a null-terminated character string that contains the window name.

dwStyle
Specifies the Windows style of **CWnd**.

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.

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 **OnMessage** 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 *lpWindowName* parameter is displayed in the title bar.

The *dwStyle* parameter can be any combination of the following window styles.

Style	Meaning
DS_LOCALEDIT	Specifies that edit controls in the dialog box will use memory in the application's data segment. By default, all edit controls in dialog boxes use memory outside the application's data segment. This feature may be suppressed by adding the **DS_LOCALEDIT** flag to the Style command for the dialog box. If this flag is not used, **EM_GETHANDLE** and **EM_SETHANDLE** messages must not be used since the storage for the control is not in the application's data segment. This feature does not affect edit controls created outside of dialog boxes.
DS_MODALFRAME	Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and Control menu by specifying the **WS_CAPTION** and **WS_SYSMENU** styles.
DS_NOIDLEMSG	Suppresses **WM_ENTERIDLE** messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.
DS_SYSMODAL	Creates a system-modal dialog box.
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_CHILDWINDOW	Same as the **WS_CHILD** style.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.

Style	Meaning
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, it is possible, when drawing within the client area of a child window, 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 by using 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). Only dialog boxes use this style.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Same as the **WS_MINIMIZE** style.
WS_MAXIMIZE	Creates a window of maximum size.
WS_MAXIMIZEBOX	Creates a window that has a maximize box.
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 box.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a caption and a border.

Style	Meaning
WS_OVERLAPPEDWINDOW	Creates an overlapped window having 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 that has 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_SIZEBOX	Same as the **WS_THICKFRAME** style.
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. Only dialog boxes use this style.
WS_THICKFRAME	Creates a window with a thick frame that can be used to size the window.
WS_TILED	Same as the **WS_OVERLAPPED** style.
WS_TILEDWINDOW	Same as the **WS_OVERLAPPEDWINDOW** style.
WS_VISIBLE	Creates a window that is initially visible. This applies to overlapped and pop-up windows.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

Return Value

TRUE if the **CWnd** window is created; otherwise **FALSE**.

See Also

::CreateWindowEx

CWnd::CreateGrayCaret

Syntax

void CreateGrayCaret(int *nWidth*, **int** *nHeight* **);**

Parameters

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 **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 losing the input focus or becoming inactive.

The system's window-border width or height can be retrieved by using 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.

See Also

::DestroyCaret, **::GetSystemMetrics**, **CWnd::ShowCaret**, **::CreateCaret**

CWnd::CreateSolidCaret

Syntax

void CreateSolidCaret(int *nWidth*, **int** *nHeight* **);**

Parameters

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 **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 losing the input focus or becoming inactive.

The system's window-border width or height can be retrieved by using 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.

See Also

::DestroyCaret, **::GetSystemMetrics**, **CWnd::ShowCaret**, **::CreateCaret**

CWnd::CWnd

Syntax **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::~CWnd**

CWnd::~CWnd

Syntax **virtual ~CWnd();**

Remarks Destroys a **CWnd** object and destroys the attached window.

See Also **CWnd::CWnd**, **CWnd::DestroyWindow**

CWnd::Default

Syntax **protected: LONG 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 is used to ensure that every message is processed. All **CWnd On***Message* member functions call this member function.

Return Value Depends on the message that was passed to this function.

See Also **::DefDlgProc**, **CWnd::DefWindowProc**, **::DefWindowProc**

CWnd::DefWindowProc

Syntax

Protected:
virtual LONG DefWindowProc(UINT *message*, **UINT** *wParam*,
LONG *lParam* **);**

Parameters

message
Specifies the Windows message to be processed.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of 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 is used to ensure that every message is processed. It should be called with the same parameters as those received by the window procedure.

The source code for the **DefWindowProc** function is provided on the Windows Software Development Kit disks.

Return Value

Dependent on the message that was passed to this function.

See Also

::DefDlgProc, **CWnd::Default**, **::DefWindowProc**

CWnd::DeleteTempMap

Syntax

static void DeleteTempMap();

Remarks

Called automatically by the idle time handler of **CWinApp**, and deletes any temporary **CWnd** objects created by **FromHandle**.

See Also

CWnd::FromHandle

CWnd::DestroyWindow

Syntax

virtual BOOL DestroyWindow();

Remarks

Destroys the Windows window attached to **CWnd**. The **DestroyWindow** member function sends appropriate messages to **CWnd** to deactivate it and remove the input focus. It also destroys **CWnd**'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 **CWnd** 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 **CWnd** itself.

The **DestroyWindow** member function also destroys modeless dialog boxes created by the **CreateDialog** Windows function.

If the **CWnd** being destroyed is a child window and does not have the **WS_NOPARENTNOTIFY** style set, then the **WM_PARENTNOTIFY** message is sent to the parent.

Return Value

Specifies whether the window is destroyed. It is **TRUE** if the window is destroyed; otherwise **FALSE**.

See Also

::CreateDialog, **CWnd::OnDestroy**, **CWnd::Detach**, **::DestroyWindow**

CWnd::Detach

Syntax

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

Syntax

int DlgDirList(const char FAR* *lpPathSpec*, **int** *nIDListBox*, **int** *nIDStaticPath*, **UINT** *nFileType* **);**

Parameters

lpPathSpec
Points to a path string, must be a **CString** or a null-terminated character string.

nIDListBox
Specifies the identifier of a list box control. 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 for displaying 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:

Value	Meaning
0x0000	Read/write data files with no additional attributes.
0x0001	Read-only files.
0x0002	Hidden files.
0x0004	System files.
0x0010	Subdirectories.
0x0020	Archives.

Value	Meaning
0x2000	**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 function.
0x4000	Drives.
0x8000	Exclusive bit. If the exclusive bit is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to normal files.

Remarks

Fills a list box control with a file or directory listing. It fills the list box specified by *nIDListBox* with the names of all files matching the path given by *lpPathSpec*.

The **DlgDirList** member function shows subdirectories enclosed in square brackets ([ ]), and shows drives in the form [-*x*-], where *x* is the drive letter.

The *lpPathSpec* parameter has the following form:

```
[drive:] [ [\u]directory[\idirectory]...\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 character. The wildcard characters are a question mark (**?**), meaning match any character, and an asterisk (*), meaning match any number of characters.

If you specify a zero-length string for *lpPathSpec* or 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.

DlgDirList sends **LB_RESETCONTENT** and **LB_DIR** messages to the list box.

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::DlgDirListComboBox, ::DlgDirList

CWnd::DlgDirListComboBox

Syntax

int DlgDirListComboBox(const char FAR* *lpPathSpec*, **int** *nIDComboBox*, **int** *nIDStaticPath*, **UINT** *nFileType* **);**

Parameters

lpPathSpec
Points to a path string, must be a **CString** or a null-terminated character string.

nIDComboBox
Specifies the identifier of a combo box control 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 for displaying 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:

Value	Meaning
0x0000	Read/write data files with no additional attributes.
0x0001	Read-only files.
0x0002	Hidden files.
0x0004	System files.
0x0010	Subdirectories.
0x0020	Archives.
0x2000	**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 function.
0x4000	Drives.
0x8000	Exclusive bit. If the exclusive bit is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to normal files.

Remarks

Fills the list box of a combo box control with a file or directory listing. It fills the list box of the combo box specified by *nIDComboBox* with the names of all files matching the path given by *lpPathSpec*.

The **DlgDirListComboBox** member function shows subdirectories enclosed in square brackets ([ ]), and shows drives in the form [-*x*-], where *x* is the drive letter.

The *lpPathSpec* parameter has the following form:

```
[drive:] [ [\u]directory[\idirectory]...\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 character. The wildcard characters are a question mark (?), meaning match any character, and an asterisk (*), meaning 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.

DlgDirListComboBox sends **CB_RESETCONTENT** and **CB_DIR** messages to the combo box.

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

Syntax

BOOL DlgDirSelect(LPSTR *lpString*, **int** *nIDListBox* **);**

Parameters

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 control 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 the current selection is a directory name or drive letter, **DlgDirSelect** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new path. If there is no selection, *lpString* does not change.

DlgDirSelect sends **LB_GETCURSEL** and **LB_GETTEXT** messages to the list box.

The **DlgDirSelect** member function 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. If it is, this function will not return a 0 value and *lpString* will remain unchanged.

Return Value

Specifies the status of the current list box selection. It is **TRUE** if the current selection is a directory name; otherwise **FALSE**.

See Also

CWnd::DlgDirList, **CWnd::DlgDirListComboBox**, **CWnd::DlgDirSelectComboBox**, **::DlgDirSelect**

CWnd::DlgDirSelectComboBox

Syntax

BOOL DlgDirSelectComboBox(LPSTR *lpString*, **int** *nIDComboBox* **);**

Parameters

lpString
Points to a buffer that is to receive the selected path.

nIDComboBox
Specifies the integer ID of the combo box control 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 the current selection is a directory name or drive letter, **DlgDirSelectComboBox** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new path. 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.

The **DlgDirSelectComboBox** member function does not allow more than one filename to be returned from a combo box.

Return Value Specifies the status of the current combo box selection. It is **TRUE** if the current selection is a directory name; otherwise **FALSE**.

See Also **CWnd::DlgDirListComboBox, ::DlgDirSelectComboBox**

CWnd::DrawMenuBar

Syntax **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::EnableWindow

Syntax **BOOL EnableWindow(BOOL** *bEnable* **= TRUE);**

Parameters

bEnable
Specifies whether the given window is to be enabled or disabled. If this parameter is **TRUE**, the **CWnd** will be enabled. If this parameter is **FALSE**, the **CWnd** 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.

CWnd 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.

Initially, all windows are enabled by default. **EnableWindow** must be used to disable **CWnd** explicitly.

Return Value

Indicates the state before the **EnableWindow** member function was called. The return value is **TRUE** if **CWnd** was previously enabled. The return value is **FALSE** if **CWnd** was previously disabled or an error occurred.

See Also

::EnableWindow

CWnd::EndPaint

Syntax

void EndPaint(LPPAINTSTRUCT *lpPaint* **);**

Parameters

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

Syntax

static CWnd* FindWindow(const char FAR* *lpClassName***,**
const char FAR* *lpWindowName* **);**

Parameters

lpClassName
Points to a null-terminated string that specifies the window's class name. If *lpClassName* is **NULL**, all class names match (a **WNDCLASS struct**).

lpWindowName
Points to a null-terminated string that specifies the window name (the window's text caption). If *lpWindowName* is **NULL**, all window names match.

Remarks

Returns the **CWnd** whose class is given by *lpClassName* and whose window name, or caption, is given by *lpWindowName.* 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

Syntax

BOOL FlashWindow(BOOL *bInvert* **);**

Parameters

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. Flashing the **CWnd** means changing the appearance of its caption bar as if the **CWnd** were changing from inactive to active status, or vice versa. (An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.)

Typically, a **CWnd** is flashed to inform the user that it requires attention, but that it does not currently have the input focus.

FlashWindow flashes the window only once; for successive flashing, create a system timer and repeatedly call **FlashWindow**.

The *bInvert* parameter should be **FALSE** only when **CWnd** 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 **TRUE** for iconic windows. If **CWnd** is iconic, **FlashWindow** will simply flash the icon; *bInvert* is ignored for iconic windows.

Return Value

Specifies the state before the call to the **FlashWindow** member function. It is **TRUE** if **CWnd** was active before the call; otherwise **FALSE**.

See Also

::FlashWindow

CWnd::FromHandle

Syntax

static CWnd* FromHandle(HWND *hWnd* **);**

Parameters

hWnd
A **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, a temporary **CWnd** object is created and attached.

Return Value

The pointer may be temporary, and should not be stored beyond immediate use.

See Also

CWnd::DeleteTempMap

CWnd::GetActiveWindow

Syntax

static CWnd* GetActiveWindow();

Remarks

Retrieves a pointer to the active **CWnd**. The active **CWnd** 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 beyond immediate use.

See Also

CWnd::SetActiveWindow, **::GetActiveWindow**

CWnd::GetCapture

Syntax

static CWnd* GetCapture();

Remarks

Retrieves the **CWnd** that has the mouse capture. Only one window has the mouse capture at any given time. This window receives mouse input whether or not the cursor is within its borders.

The **CWnd** receives the mouse capture when the **SetCapture** member function is called.

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

Syntax

static CPoint GetCaretPos();

Remarks

Retrieves the client coordinates of the caret's current position, and copies them to a **CPoint** structure.

The caret position is given in the client coordinates of the **CWnd** window.

Return Value

CPoint containing the coordinates of the caret's position.

See Also

::GetCaretPos

CWnd::GetCheckedRadioButton

Syntax

int GetCheckedRadioButton(int *nIDFirstButton*, **int** *nIDLastButton* **);**

Parameters

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

This function retrieves the ID of the currently checked radio button in the specified group.

Return Value

ID of the checked radio button.

See Also

CWnd::CheckRadioButton

CWnd::GetClientRect

Syntax

void GetClientRect(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to a **RECT** structure or a **CRect** to receive the client coordinates.

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

Syntax

static CWnd* GetClipboardOwner();

Remarks

Retrieves the current owner of the Clipboard.

The Clipboard can still contain data even if the Clipboard is not currently owned.

Return Value

Identifies the **CWnd** that owns the Clipboard. It is **NULL** if the Clipboard is not owned.

The returned pointer may be temporary and should not be stored for later use.

See Also

CWnd::GetClipboardViewer, **::GetClipboardOwner**

CWnd::GetClipboardViewer

Syntax

static CWnd* GetClipboardViewer();

Remarks

Retrieves the first window in the Clipboard-viewer chain.

Return Value

Identifies the window currently responsible for displaying the Clipboard. It is **NULL** if there is no viewer.

The returned pointer may be temporary, and should not be stored for later use.

See Also

CWnd::GetClipboardOwner, **::GetClipboardViewer**

CWnd::GetCurrentMessage

Syntax

Protected:
static const MSG* GetCurrentMessage();

Return Value

Returns a pointer to the message the window is currently processing. Should only be called when in an **OnMessage** handler.

CWnd::GetDC

Syntax

CDC* GetDC();

Remarks

Retrieves a pointer to a display context for the client area. The display context can be used in subsequent GDI functions to draw in the client area.

Retrieves a common, class, or private display context depending on the class style specified for the **CWnd**. For common display 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.

Unless the display context belongs to a window class, the **ReleaseDC** member function must be called to release the context after painting. Since only five common display contexts are available at any given time, failure to release a display context can prevent other applications from accessing a display context.

A display context belonging to the **CWnd** class is returned by the **GetDC** member function if **CS_CLASSDC**, **CS_OWNDC**, or **CS_PARENTDC** were specified as a style in the **WNDCLASS** structure when the class was registered.

Return Value

Identifies the display context for the **CWnd** client area if the function is successful. The return value is **NULL** if the function is unsuccessful. The pointer may be temporary, and should not be stored for later use.

See Also

CWnd::ReleaseDC, **::GetDC**, **CClientDC**

CWnd::GetDesktopWindow

Syntax

static CWnd* 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

Syntax

int GetDlgCtrlID() const;

Remarks

Returns the **CWnd**'s ID value if **CWnd** is a child window.

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. If the function fails, the return value is **NULL**.

See Also

::GetDlgCtrlID

CWnd::GetDlgItem

Syntax

CWnd* GetDlgItem(int *nID* **) const;**

Parameters

nID
Specifies the integer ID of the item to be retrieved.

Remarks

Retrieves a pointer to the specified control in a dialog box.

Can be used with any parent-child pair, not just a dialog box, as long as the child window has a unique ID (as specified by the *nID* parameter in the **Create** member function that created the child window).

The pointer returned is usually cast to the type of control identified by *nID*.

Return Value

A pointer to the given control. 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**, **::GetDlgItem**

CWnd::GetDlgItemInt

Syntax

UINT GetDlgItemInt(int *nID*, **BOOL*** *lpTrans* = **NULL,**
  BOOL *bSigned* = **TRUE) const;**

Parameters

nID
  Specifies the integer identifier of the dialog-box item 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

Translates the text of the specified control in the given dialog box into an integer value.

Retrieves the text of the control identified by *nID*. It translates the text by stripping any extra spaces at the beginning of the text and converting decimal digits, stopping the translation when it reaches the end of the text or encounters any non-numeric 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.

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.

Zero 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 given 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

Syntax

int GetDlgItemText(int *nID*, **LPSTR** *lpStr*, **int** *nMaxCount* **) const;**

Parameters

nID
  Specifies the integer identifier of the dialog-box item whose caption or text is to be retrieved.

lpStr
  Points to the buffer to receive the 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 caption 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. The value is 0 if no text is copied.

See Also

CWnd::GetDlgItem, **CWnd::GetDlgItemInt**, **::GetDlgItemText**, **WM_GETTEXT**

CWnd::GetFocus

Syntax **static CWnd* 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 or an error occurred.

The pointer may be temporary, and should not be stored for later use.

See Also **CWnd::GetActiveWindow**, **CWnd::GetCapture**, **CWnd::SetFocus**, **::GetFocus**

CWnd::GetFont

Syntax **CFont* GetFont();**

Remarks Gets the current font.

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

Syntax **CWnd* GetLastActivePopup() const;**

Remarks 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 **CWnd** itself if any of the following conditions are met:

- **CWnd** itself was most recently active.
- **CWnd** does not own any pop-up windows.
- **CWnd** 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

Syntax

CMenu* GetMenu() const;

Remarks

Retrieves a pointer to the **CWnd**'s menu. 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

Syntax

CWnd* GetNextDlgGroupItem(CWnd* *pWndCtl*,
BOOL *bPrevious* **= FALSE) const;**

Parameters

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 this parameter is **TRUE**, the function searches for the previous control in the group. If this parameter is **FALSE**, the function 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.

The returned pointer may be temporary, and should not be stored for later use.

See Also

CWnd::GetNextDlgTabItem, **::GetNextDlgGroupItem**

CWnd::GetNextDlgTabItem

Syntax

CWnd* GetNextDlgTabItem(CWnd* *pWndCtl*,
    BOOL *bPrevious* **= FALSE) const;**

Parameters

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 this parameter is **TRUE**, the function searches for the previous control in the dialog box. If this parameter is **FALSE**, the function searches for the next control in the dialog box.

Remarks

Retrieves a pointer to the first control that was created with the **WS_TABSTOP** style and precedes (or follows) the specified control.

Return Value

The return value is the previous (or next) control that has the **WS_TABSTOP** style.

The returned pointer may be temporary, and should not be stored for later use.

See Also

CWnd::GetNextDlgGroupItem, **::GetNextDlgTabItem**

CWnd::GetNextWindow

Syntax

CWnd* GetNextWindow(UINT *nFlag* **= GW_HWNDNEXT) const;**

Parameters

nFlag

Specifies whether the function returns a pointer to the next window or the previous window. It can be either of the following values:

Value	Meaning
GW_HWNDNEXT	Returns the window that follows the **CWnd** object on the window-manager's list.
GW_HWNDPREV	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.

The returned pointer may be temporary, and should not be stored for later use.

See Also

::GetNextWindow

CWnd::GetParent

Syntax

CWnd* GetParent() const;

Remarks

Retrieves the parent window (if any).

Return Value

Identifies the parent window. The value is **NULL** if the **CWnd** has no parent window.

The returned pointer may be temporary, and should not be stored for later use.

See Also

::GetParent

CWnd::GetSafeHwnd

Syntax

HWND GetSafeHwnd() const;

Return Value

Returns **m_hWnd**, or **NULL** if **this** is **NULL**.

CWnd::GetScrollPos

Syntax

int GetScrollPos(int *nBar* **) const;**

Parameters

nBar
Specifies the scroll bar to examine. The parameter can take one of the following values:

Value	Meaning
SB_CTL	Retrieves the position of a scroll-bar control.
SB_HORZ	Retrieves the position of the **CWnd** horizontal scroll bar.
SB_VERT	Retrieves the position of the **CWnd** vertical scroll bar.

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 50 to 100 and the thumb is in the middle of the bar, the current position is 75.

Return Value

Specifies the current position of the scroll box.

See Also

::GetScrollPos

CWnd::GetScrollRange

Syntax

void GetScrollRange(int *nBar*, **LPINT** *lpMinPos*, **LPINT** *lpMaxPos* **) const;**

Parameters

nBar
Specifies the scroll bar to examine. The parameter can take one of the following values:

Value	Meaning
SB_CTL	Retrieves the position of a scroll-bar control.
SB_HORZ	Retrieves the position of the **CWnd** horizontal scroll bar.
SB_VERT	Retrieves the position of the **CWnd** 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 standard scroll bars or is not a scroll-bar control, 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

Syntax

DWORD GetStyle() const;

Return Value

The window's style.

See Also

::GetWindowLong, CWnd::CreateEx

CWnd::GetSuperWndProcAddr

Syntax

protected: virtual FARPROC* GetSuperWndProcAddr();

Return Value

The original **WndProc** address of a subclassed window.

CWnd::GetSysModalWindow

Syntax

static CWnd* GetSysModalWindow();

Remarks

Returns the system-modal window, if there is one.

Return Value

Identifies the system-modal window, if one is present. If no such window is present, the return value is **NULL**.

The returned pointer may be temporary, and should not be stored for later use.

See Also

::GetSysModalWindow, CWnd::SetSysModalWindow

CWnd::GetSystemMenu

Syntax

CMenu* GetSystemMenu(BOOL *bRevert* **) const;**

Parameters

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 **GetSystemMenu** member 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**. Menu items on the Control 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 automatically dims items on the standard Control menu, depending on the situation. **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

Syntax

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

Syntax

BOOL GetUpdateRect(LPRECT *lpRect*, **BOOL** *bErase* = **FALSE);**

Parameters

lpRect
Points to a **CRect** or **RECT** structure that is to receive the client coordinates of the update enclosing the update region.

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 **TRUE** if the update region is not empty; otherwise **FALSE**.

See Also

CWnd::BeginPaint, **::GetUpdateRect**

CWnd::GetUpdateRgn

Syntax

int GetUpdateRgn(CRgn* *pRgn*, **BOOL** *bErase* **= FALSE);**

Parameters

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:

Value	Meaning
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

CWnd::BeginPaint, **::GetUpdateRgn**

CWnd::GetWindow

Syntax

CWnd* GetWindow(UINT *nCmd* **) const;**

Parameters

nCmd

Specifies the relationship between **CWnd** and the returned window. It can take one of the following values:

Value	Meaning
GW_CHILD	Identifies **CWnd**'s 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 **CWnd**'s owner.

Remarks

Searches the window manager's list for a window. The window-manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows. The *nCmd* parameter specifies the relationship between **CWnd** and the returned window.

Return Value

Identifies a window. The value is **NULL** if the function reaches the end of the window-manager's list or if *nCmd* is invalid.

The returned pointer may be temporary, and should not be stored for later use.

See Also

CWnd::GetDlgItem, **::GetWindow**

CWnd::GetWindowDC

Syntax

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.

Assigns default attributes to the display context each time it retrieves the context. Previous attributes are lost.

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, the value is **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::GetWindowRect

Syntax

void GetWindowRect(LPRECT *lpRect* **) const;**

Parameters

lpRect
Points to a **CRect** 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

Syntax

int GetWindowText(LPSTR *lpString*, **int** *nMaxCount* **) const;**

Parameters

lpString
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.

Remarks

Copies **CWnd**'s caption title (if it has one) into the buffer pointed to by *lpString*. If the **CWnd** object is a control, the **GetWindowText** member function copies the text within the control instead of copying the caption.

Return Value

Specifies the length of the copied string. It is 0 if **CWnd** has no caption or if the caption is empty.

See Also

CWnd::SetWindowText, **CWnd::GetWindowText**, **WM_GETTEXT**

CWnd::GetWindowTextLength

Syntax

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.

Return Value

Specifies the text length, not including any null-termination character. The value is 0 if no such text exists.

See Also

::GetWindowTextLength

CWnd::HideCaret

Syntax

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

Syntax

BOOL HiliteMenuItem(CMenu* *pMenu*, **UINT** *nIDHiliteItem*, **UINT** *nHilite* **);**

Parameters

pMenu
Identifies the top-level menu that contains the item to be highlighted.

nIDHiliteItem
Specifies the integer identifier of the menu item or the offset of the menu item in the menu, 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:

Value	Meaning
MF_BYCOMMAND	Interprets *nIDHiliteItem* as the menu-item ID (the default interpretation).
MF_BYPOSITION	Interprets *nIDHiliteItem* as an offset.
MF_HILITE	Highlights the item. If this value is not given, highlighting is removed from the item.
MF_UNHILITE	Removes highlighting from the item.

Remarks
Highlights or removes the highlighting from a top-level (menu-bar) menu item.

Return Value
Specifies whether the menu item was highlighted. **TRUE** if the item was highlighted; otherwise **FALSE**.

See Also
CMenu::ModifyMenu, **::HiliteMenuItem**

CWnd::Invalidate

Syntax
void Invalidate(BOOL *bErase* **= FALSE);**

Parameters
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 using 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 **WM_PAINT** 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

Syntax **void InvalidateRect(LPRECT** *lpRect*, **BOOL** *bErase* = **FALSE);**

Parameters

lpRect
Points to a **CRect** 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 the region is validated by using 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 **WM_PAINT** 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

Syntax

void InvalidateRgn(CRgn* *pRgn*, **BOOL** *bErase* = **FALSE);**

Parameters

pRgn
Identifies the region to be added to the update region. The region is assumed to have client coordinates.

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 **WM_PAINT** is next sent, or the region is validated by using 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 **WM_PAINT** 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 using one of the region functions.

See Also

CWnd::BeginPaint, **CWnd::ValidateRect**, **CWnd::ValidateRgn**, **::InvalidateRgn**

CWnd::IsChild

Syntax

BOOL IsChild(CWnd* *pWnd* **) const;**

Parameters

pWnd
Identifies the window to be checked.

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 **TRUE** if the window identified by *pWnd* is a child window of **CWnd**; otherwise **FALSE**.

See Also

::IsChild

CWnd::IsDlgButtonChecked

Syntax

UINT IsDlgButtonChecked(int *nIDButton* **) const;**

Parameters

nIDButton
Specifies the integer identifier of the button control.

Remarks

If the **CWnd** object is a button control, the **IsDlgButtonChecked** member function determines whether it has a check mark next to it. If it is a three-state button control, it determines if it is dimmed, checked, or neither.

Return Value

Nonzero if the given control is checked, and 0 if it is not checked. For three-state buttons, the return value is 2 if the button is dimmed, 1 if the button is checked, and 0 if it is unchecked.

See Also

::IsDlgButtonChecked

CWnd::IsIconic

Syntax **BOOL IsIconic() const;**

Remarks Specifies whether **CWnd** is minimized (iconic).

Return Value Specifies whether the **CWnd** object is minimized. It is **TRUE** if **CWnd** is minimized; otherwise **FALSE**.

See Also **::IsIconic**

CWnd::IsWindowEnabled

Syntax **BOOL IsWindowEnabled() const;**

Remarks Specifies whether **CWnd** is enabled for mouse and keyboard input.

Return Value Specifies whether **CWnd** is enabled. The value is **TRUE** if it is enabled; otherwise **FALSE**.

See Also **::IsWindowEnabled**

CWnd::IsWindowVisible

Syntax **BOOL IsWindowVisible() const;**

Remarks Returns **TRUE** any time an application has made **CWnd** visible by using the **ShowWindow** member function (even if **CWnd** is completely covered by another child or pop-up window, the return value is **TRUE**).

Return Value Specifies whether a given window exists on the screen. It is **TRUE** if it exists on the screen; otherwise **FALSE**.

See Also **CWnd::ShowWindow**, **::IsWindowVisible**

CWnd::IsZoomed

Syntax **BOOL IsZoomed() const;**

Remarks Determines whether **CWnd** has been maximized.

Return Value Specifies whether **CWnd** is maximized. The value is **TRUE** if it is maximized; otherwise **FALSE**.

See Also **::IsZoomed**

CWnd::KillTimer

Syntax **BOOL KillTimer(int** *nIDEvent* **);**

Parameters *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 **TRUE** if the event was killed. It is **FALSE** if the **KillTimer** member function could not find the specified timer event.

See Also **CWnd::SetTimer**, **::KillTimer**

CWnd::MessageBox

Syntax **int MessageBox(const char FAR*** *lpText*, **const char FAR*** *lpCaption* = **NULL**, **UINT** *nType* = **MB_OK);**

Parameters *lpText*
Points to a **CString** or null-terminated string containing the message to be displayed.

lpCaption
Points to a **CString** or null-terminated string to be used for the message-box caption. If *lpCaption* is **NULL**, the default caption "Error" is used.

nType
Specifies the contents of the message box. It can be a combination of the following values:

Value	Meaning
MB_ABORTRETRYIGNORE	Message box contains three push buttons: Abort, Retry, and Ignore.
MB_APPLMODAL	The user must respond to the message box before continuing work in the **CWnd**. However, the user can move to the windows of other applications and work in those windows. **MB_APPLMODAL** is the default if **MB_SYSTEMMODAL** is not specified.
MB_DEFBUTTON1	First button is the default. Note that the first button is always the default unless **MB_DEFBUTTON2** or **MB_DEFBUTTON3** is specified.
MB_DEFBUTTON2	Second button is the default.
MB_DEFBUTTON3	Third button is the default.
MB_ICONEXCLAMATION	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION	An icon consisting of a lowercase "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.
MB_OK	Message box contains one push button: OK.
MB_OKCANCEL	Message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	Message box contains two push buttons: Retry and Cancel.

Value	Meaning
MB_SYSTEMMODAL	All applications are suspended until the user responds to the message box. Unless the application specifies **MB_ICONSTOP**, the message box does not become modal until after it is created; consequently, the parent window and other windows continue to receive messages resulting from its activation. System-modal message boxes are used to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory).
MB_YESNO	Message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	Message box contains three push buttons: Yes, No, and Cancel.

Remarks

Creates and displays a window that contains an application-supplied message and caption, plus a combination of the predefined icons and push buttons described in the preceding list.

When a system-modal message box is created to indicate that the system is low on memory, do not take the strings passed as *lpText* and *lpCaption* from a resource file, since an attempt to load the resource may fail.

When an application calls the **MessageBox** member function and specifies the **MB_ICONSTOP** and **MB_SYSTEMMODAL** flags for *nType*, Windows will display the resulting message box regardless of available memory. When these flags are specified, Windows limits the length of the message-box text to one line.

Return Value

Specifies the outcome of the function. It is 0 if there is not enough memory to create the message box. Otherwise, it is one of the following menu-item values returned by the message box:

Value	Meaning
IDABORT	Abort button pressed
IDCANCEL	Cancel button pressed
IDIGNORE	Ignore button pressed
IDNO	No button pressed

Value	Meaning
IDOK	OK button pressed
IDRETRY	Retry button pressed
IDYES	Yes button pressed

If a message box has a Cancel button, the **IDCANCEL** value will be returned if either the ESC key or the Cancel button is pressed. If the message box has no Cancel button, pressing the ESC key has no effect.

See Also **::MessageBox**

CWnd::MoveWindow

Syntax

void MoveWindow(int *x*, **int** *y*, **int** *nWidth*, **int** *nHeight*,
  BOOL *bRepaint* = **TRUE);**

void MoveWindow(LPRECT *lpRect*, **BOOL** *bRepaint* = **TRUE);**

Parameters

x
  Specifies the new position of the left side of the window.

y
  Specifies the new position of the top of the window.

nWidth
  Specifies the new width of the window.

nHeight
  Specifies the new height of the window.

bRepaint
  Specifies whether the window is to be repainted. If this parameter is **TRUE**, the window is repainted. This is set by default.

lpRect
  The **CRect** or **RECT** structure specifying 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 **WM_GETMINMAXINFO**. This 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 will be replaced by the minimum or maximum values specified in the **WM_GETMINMAXINFO** message.

See Also **CWnd::SetWindowPos**, **::MoveWindow**

CWnd::OnActivate

Syntax

afx_msg void OnActivate(UINT *nState*, **CWnd*** *pWndOther*,
BOOL *bMinimized* **);**

Parameters

nState
Indicates the minimized state of the window being activated or deactivated. A nonzero value indicates that the **CWnd** object is minimized.

pWndOther
Pointer to the **CWnd** being activated or deactivated. The pointer can be **NULL**, and it may be temporary.

bMinimized
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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ACTIVATE** message.

See Also **WM_MOUSEACTIVATE**, **WM_NCACTIVATE**, **CWnd::Default**, **WM_ACTIVATE**

CWnd::OnActivateApp

Syntax

afx_msg void OnActivateApp(BOOL *bActive*, **HANDLE** *hTask* **);**

Parameters

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 when **CWnd** is about to be activated and **CWnd** belongs to a different task than the currently active window. **OnActivateApp** is called for all top-level windows of the task being activated, and for all top-level windows of the task being deactivated.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ACTIVATEAPP** message.

See Also

CWnd::Default, **WM_ACTIVATEAPP**

CWnd::OnAskCbFormatName

Syntax

afx_msg void OnAskCbFormatName(UINT *nMaxCount*, **LPSTR** *lpString* **);**

Parameters

nMaxCount
Specifies the maximum number of bytes to copy.

lpString
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) and 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ASKCBFORMATNAME** message.

See Also **CWnd::Default, WM_ASKCBFORMAINAME**

CWnd::OnCancelMode

Syntax **afx_msg void OnCancelMode();**

Remarks 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.

Calls the **Default** member function, which responds by calling the **ReleaseCapture** Windows function. The **Default** member function does not cancel any other modes.

Override this member function in your derived class to handle the **WM_CANCELMODE** message.

See Also **CWnd::Default, ::ReleaseCapture, WM_CANCELMODE**

CWnd::OnChangeCbChain

Syntax **afx_msg void OnChangeCbChain(HWND** *hWndRemove*, **HWND** *hWndAfter* **);**

Parameters

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CHANGECBCHAIN** message.

See Also **CWnd::ChangeClipboardChain**, **::SendMessage**, **CWnd::Default**

CWnd::OnChar

Syntax **afx_msg void OnChar(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

nChar
Contains the value of the key.

nRepCnt
Contains the repeat count.

nFlags
Contains 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).
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 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 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 CONTROL 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*.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CHAR** message.

See Also

CWnd::Default, **WM_CHAR**, **WM_KEYDOWN**, **WM_KEYUP**

CWnd::OnCharToItem

Syntax

afx_msg int OnCharToItem(UINT *nChar*, **CWnd*** *pListBox*, **UINT** *nIndex* **);**

Parameters

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

A list box with the **LBS_WANTKEYBOARDINPUT** style sends its owner a **WM_CHARTOITEM** message in response to a **WM_CHAR** message. **WM_CHARTOITEM** is handled by default by **OnCharToItem**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CHARTOITEM** 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 will perform the default action in response to the keystroke. A return value of 0 or greater specifies the index of an item in the list box and indicates that the list box will perform the default action for the keystroke on the given item.

See Also

CWnd::Default, WM_CHAR, WM_CHARTOITEM

CWnd::OnChildActivate

Syntax

afx_msg void OnChildActivate();

Remarks

If the **CWnd** object is a child window, **OnChildActivate** is called whenever the size or position of the window changes.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CHILDACTIVATE** message.

See Also

CWnd::SetWindowPos, CWnd::OnClose, WM_CHILDACTIVATE, CWnd::Default

CWnd::OnClose

Syntax

afx_msg void OnClose();

Remarks

Called as a signal that the **CWnd** or an application is to terminate. An application can prompt the user for confirmation and destroy the **CWnd** object by calling the **DestroyWindow** member function only if the user confirms the choice.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CLOSE** message.

See Also

CWnd::DestroyWindow, ::PostQuitMessage, WM_CLOSE, CWnd::Default

CWnd::OnCommand

Syntax

virtual BOOL OnCommand(UINT *wParam*, **LONG** *lParam* **);**

Parameters

wParam
Identifies the menu item or control.

lParam
Specifies additional information. If the message is from a menu, the low-order word and the high-order word are both 0. If the message is from an accelerator, the low-order word is 0 and the high-order word is 1. If the message is from a control, the low-order word is the handle of the window sending the message and the high-order word is the notification code.

Remarks

Called when the user selects an item from a **CWnd** menu, when a child control sends a notification message to **CWnd**, or when an access keystroke is translated.

Access keystrokes that are defined to select items from the Control menu are translated to **WM_SYSCOMMAND** messages.

If an access keystroke that corresponds to a menu item occurs when the **CWnd** is minimized, **OnCommand** is not called. However, if an access keystroke that does not match any of the items on the **CWnd** menu or on the Control menu occurs, **OnCommand** is called, even if **CWnd** is minimized.

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 **TRUE** if it processes this message; otherwise **FALSE**.

See Also

WM_SYSCOMMAND, WM_COMMAND

CWnd::OnCompacting

Syntax

afx_msg void OnCompacting(UINT *nCpuTime* **);**

Parameters

nCpuTime
Specifies the ratio of CPU time currently spent by Windows compacting memory. For example, 8000h represents 50 percent of CPU time.

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_COMPACTING** message.

See Also

::GetNumTasks, CWnd::Default, WM_COMPACTING

CWnd::OnCompareItem

Syntax

afx_msg int OnCompareItem(LPCOMPAREITEMSTRUCT *lpCompareItemStruct* **);**

Parameters

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

Override this member function in your derived class to handle the **WM_COMPAREITEM** message.

Use the overridden member function to specify the relative position of a new item in a 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.

The *lpCompareItemStruct* parameter is 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. **OnCompareItem** should return a value indicating which of the items should appear before the other. Typically, Windows makes this call several times until it determines the exact position for the new item.

A **COMPAREITEMSTRUCT** data structure has this form:

```
typedef struct tagCOMPAREITEMSTRUCT {
    WORD   CtlType;
    WORD   CtlID;
    HWND   hwndItem;
    WORD   itemID1;
    DWORD  itemData1;
    WORD   itemID2;
    DWORD  itemData2;
} COMPAREITEMSTRUCT;
```

Members

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.

This message-handler member function calls the **Default** member function.

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.

See Also

COMPAREITEMSTRUCT, **WM_COMPAREITEM**, **CWnd::Default**

CWnd::OnCreate

Syntax

afx_msg int OnCreate(LPCREATESTRUCT *lpCreateStruct* **);**

Parameters

lpCreateStruct
Points to a **CREATESTRUCT** structure containing information about the **CWnd** object being created.

Remarks

Called when an application requests that the **CWnd** object be created by calling the **Create** or **CreateEx** member function. The new **CWnd** object receives this call after the **CWnd** object 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.

A **CREATESTRUCT** structure has the following form:

```
typedef struct tagCREATESTRUCT {
   LPSTR    lpCreateParams;
   HANDLE   hInstance;
   HANDLE   hMenu;
   HWND     hwndParent;
   int      cy;
   int      cx;
   int      y;
   int      x;
   LONG     style;
   LPSTR    lpszName;
   LPSTR    lpszClass;
   DWORD    dwExStyle;
} CREATESTRUCT;
```

Members

lpCreateParams
Points to data to be used for creating 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, the 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, the 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 struct**).

dwExStyle
Specifies the extended style for the new window.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CREATE** message.

Return Value

OnCreate must return 0 to continue the creation of the **CWnd** object. If the application returns –1, the **CWnd** will be destroyed.

See Also

CWnd::CreateEx, CWnd::OnNcCreate, WM_CREATE, CWnd::Default, CWnd::FromHandle

CWnd::OnCtlColor

Syntax

afx_msg HBRUSH OnCtlColor(CDC* *pDC*, **CWnd*** *pWnd*,
UINT *nCtlColor* **);**

Parameters

pDC
Contains a pointer to the display context for the child window. May be temporary.

pWnd
Contains a pointer to the child **CWnd**. May be temporary.

nCtlColor
Contains one of the following values, specifying the type of control:

Value	Meaning
CTLCOLOR_BTN	Button control
CTLCOLOR_DLG	Dialog box
CTLCOLOR_EDIT	Edit control

Value	Meaning
CTLCOLOR_LISTBOX	List box control
CTLCOLOR_MSGBOX	Message box
CTLCOLOR_SCROLLBAR	Scroll-bar control
CTLCOLOR_STATIC	Static control

Remarks

Called when a child system-defined control class or a message box is about to be drawn. The following controls call **OnCtlColor**:

Combo boxes	Buttons
Edit controls	Static controls
List boxes	Scroll bars

To change the background color of a single-line edit control, you must set the brush handle in both the **CTLCOLOR_EDIT** and **CTLCOLOR_MSGBOX** message codes, as well as calling the **SetBkColor** function in response to the **CTLCOLOR_EDIT** code.

The return value from the function has no effect on a button with the **BS_PUSHBUTTON** or **BS_DEFPUSHBUTTON** style.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_CTLCOLOR** message.

Return Value

OnCtlColor must return a handle to the brush that is to be used for painting the control background, or it must return **NULL**.

See Also

CDC::SetBkColor, **WM_CTLCOLOR**, **CWnd::Default**

CWnd::OnDeadChar

Syntax

afx_msg void OnDeadChar(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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, 0 otherwise).
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 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 CONTROL 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*.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_DEADCHAR** message.

See Also **CWnd::Default**, **WM_DEADCHAR**

CWnd::OnDeleteItem

Syntax **afx_msg void OnDeleteItem(LPDELETEITEMSTRUCT** *lpDeleteItemStruct* **);**

Parameters

lpDeleteItemStruct
Specifies a long pointer to a **DELETEITEMSTRUCT** data structure that contains information about the deleted list box item.

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 are removed by **CComboBox::DeleteString**, **CListBox::DeleteString**, **CComboBox::ResetContent**, or **CListBox::ResetContent**.

A **DELETEITEMSTRUCT** data structure has this form:

```
typedef struct tagDELETEITEMSTRUCT {
    WORD  CtlType
    WORD  CtlID;
    WORD  itemID;
    HWND  hwndItem;
    DWORD itemData;
} DELETEITEMSTRUCT;
```

Members

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**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_DELETEITEM** message.

See Also **CComboBox::DeleteString**, **CListBox::DeleteString**, **CComboBox::ResetContent**, **CListBox::ResetContent**, **CWnd::Default**, **WM_DELETEITEM**

CWnd::OnDestroy

Syntax **afx_msg void OnDestroy();**

Remarks Called to inform the **CWnd** 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** is the main window (**CWinApp**'s **m_pMainWnd**) then **OnDestroy** calls **PostQuitMessage**.

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**, **::PostQuitMessage**, **CWnd::SetClipboardViewer**, **WM_DESTROY**

CWnd::OnDestroyClipboard

Syntax **afx_msg void OnDestroyClipboard();**

Remarks Called for the Clipboard owner when the Clipboard is emptied through a call to the **EmptyClipboard** Windows function.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_DESTROYCLIPBOARD** message.

See Also **::EmptyClipboard, CWnd::Default, WM_DESTROYCLIPBOARD**

CWnd::OnDevModeChange

Syntax **afx_msg void OnDevModeChange(LPSTR** *lpDeviceName* **);**

Parameters *lpDeviceName*
Points to the device name specified in the Windows initialization file, WIN.INI.

Remarks Called for all top-level **CWnd**s when the user changes device-mode settings.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_DEVMODECHANGE** message.

See Also **CWnd::Default, WM_DEVMODECHANGE**

CWnd::OnDrawClipboard

Syntax **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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_DRAWCLIPBOARD** message.

See Also

::SendMessage, CWnd::SetClipboardViewer, WM_CHANGECBCHAIN, CWnd::Default

CWnd::OnDrawItem

Syntax

afx_msg void OnDrawItem(LPDRAWITEMSTRUCT *lpDrawItemStruct* **);**

Parameters

lpDrawItemStruct
Specifies a long pointer to a **DRAWITEMSTRUCT** data structure that contains 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.

A **DRAWITEMSTRUCT** structure has this form:

```
typedef struct tagDRAWITEMSTRUCT {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemAction;
    WORD   itemState;
    HWND   hwndItem;
    HDC    hDC;
    RECT   rcItem;
    DWORD  itemData;
} DRAWITEMSTRUCT;
```

Members

CtlType
Is the control type. The values for control types are as follows:

Value	Meaning
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 can be –1. This 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. This indicates to the user whether the list box or combo box has 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:

Value	Meaning
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. The **itemState** member 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:

Value	Meaning
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
For combo boxes, list boxes and buttons, this member specifies the window handle of the control; for menus, it contains the handle of the menu (**HMENU**) containing the item.

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 does not clip menu items. When drawing menu items, the owner must ensure that the owner does 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
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

This message-handler member function calls **Default**. Override this member function in your derived class to handle the **WM_DRAWITEM** message.

See Also **DRAWITEMSTRUCT**, **CWnd::Default**, **WM_DRAWITEM**, **CWnd::FromHandle**, **::FromHandle**

CWnd::OnEnable

Syntax **afx_msg void OnEnable(BOOL** *bEnable* **);**

Parameters *bEnable*
Specifies whether the **CWnd** 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 **CWnd**. It is sent to the **CWnd** whose enabled state is changing. **OnEnable** is called before the **EnableWindow** member function returns, but after the window enabled state (**WS_DISABLE** style bit) has changed.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ENABLE** message.

See Also **CWnd::EnableWindow**, **CWnd::Default**, **WM_ENABLE**

CWnd::OnEndSession

Syntax **afx_msg void OnEndSession(BOOL** *bEnding* **);**

Parameters

bEnding
Specifies whether or not the session is being ended. It is **TRUE** if the session is being ended; otherwise, it is **FALSE**.

Remarks Called after the **CWnd** has returned **TRUE** from a **OnQueryEndSession** member function call. The **OnEndSession** call informs the **CWnd** 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**.

CWnd does not need to call the **DestroyWindow** member function or **PostQuitMessage** Windows function when the session is ending.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ENDSESSION** message.

See Also **CWnd::DestroyWindow**, **::ExitWindows**, **::PostQuitMessage**, **WM_QUERYENDSESSION**, **CWnd::Default**, **WM_ENDSESSION**

CWnd::OnEnterIdle

Syntax

afx_msg void OnEnterIdle(UINT *nWhy*, **CWnd*** *pWho* **);**

Parameters

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:

Value	Description
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 containing 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ENTERIDLE** message.

See Also

CWnd::Default, WM_ENTERIDLE

CWnd::OnEraseBkgnd

Syntax

afx_msg BOOL OnEraseBkgnd(CDC* *pDC* **);**

Parameters

pDC
Specifies the device-context object.

Remarks

Called when the **CWnd** background needs erasing (for example, when resized). It is called to prepare an invalidated region for painting.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ERASEBKGND** message.

The **Default** and **DefWindowProc** member functions erase the background using the class background brush specified by the **hbrbackground** member of the class structure.

If the **hbrbackground** member is **NULL**, **OnEraseBkgnd** should erase the background color. **OnEraseBkgnd** should align the origin of the intended brush with the **CWnd** coordinates by first calling the **UnrealizeObject** Windows function for the brush, and then selecting the brush.

Windows assumes the background should be computed by using 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

OnEraseBkgnd should return **TRUE** to erase the background; otherwise, it should return **FALSE**.

See Also

::UnrealizeObject, **WM_ICONERASEBKGND**, **CWnd::Default**, **WM_ERASEBKGND**, **CBrush::FromHandle**

CWnd::OnFontChange

Syntax

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** Window function) should send the **WM_FONTCHANGE** message to all top-level windows.

To send the **WM_FONTCHANGE** message to all top-level windows, an application can use the **SendMessage** Windows function to send the **WM_FONTCHANGE** message with the *hWnd* parameter set to 0xFFFF.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_FONTCHANGE** message.

See Also **::AddFontResource**, **::RemoveFontResource**, **::SendMessage**, **CWnd::Default**, **WM_FONTCHANGE**

CWnd::OnGetDlgCode

Syntax **afx_msg UINT OnGetDlgCode();**

Remarks Normally, Windows handles all DIRECTION-key and TAB-key input to a **CWnd** control. When **OnGetDlgCode** is called, a **CWnd** control can choose a particular type of input to process itself.

Although the **Default** and **DefWindowProc** member functions always return 0 in response to the **WM_GETDLGCODE** message, the **OnGetDlgCode** functions for the predefined control classes return a code appropriate for each class.

OnGetDlgCode's returned values are useful only with user-defined dialog controls or standard controls modified by subclassing.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_GETDLGCODE** message.

Return Value One or more of the following values, indicating which type of input the application processes:

Value	Meaning
DLGC_DEFPUSHBUTTON	Default push button.
DLGC_HASSETSEL	**EM_SETSEL** messages.
DLGC_PUSHBUTTON	Pushbutton.
DLGC_RADIOBUTTON	Radio button.
DLGC_WANTALLKEYS	All keyboard input.
DLGC_WANTARROWS	DIRECTION keys.
DLGC_WANTCHARS	**WM_CHAR** messages.

Value	Meaning
DLGC_WANTMESSAGE	All keyboard input (the application passes this message on to control).
DLGC_WANTTAB	TAB key.

See Also **CWnd::Default**, **WM_GETDLGCODE**

CWnd::OnGetMinMaxInfo

Syntax **afx_msg void OnGetMinMaxInfo(LPPOINT** *lpPoints* **);**

Parameters

lpPoints
Points to an array of five **POINT** structures that contain the following information:

Value	Meaning
apt[0]	This value is reserved for internal use.
apt[1]	Specifies the maximized width (*point*.**x**) and the maximized height (*point*.**y**) of the **CWnd**.
apt[2]	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**).
apt[3]	Specifies the minimum tracking width (*point*.**x**) and the minimum tracking height (*point*.**y**) of the **CWnd**.
apt[4]	Specifies the maximum tracking width (*point*.**x**) and the maximum tracking height (*point*.**y**) of the **CWnd**.

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**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_GETMINMAXINFO** message.

See Also

CWnd::Default, **WM_GETMINMAXINFO**

CWnd::OnHScroll

Syntax

afx_msg void OnHScroll(UINT *nSBCode*, **UINT** *nPos*, **CWnd*** *pScrollBar* **);**

Parameters

nSBCode
Specifies a scroll-bar code that indicates the user's scrolling request. This parameter can be one of the following values:

Value	Description
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
Specifies the scroll box position if the scroll-bar code is **SB_THUMBPOSITION**; otherwise, not used.

pScrollBar
If the control is a scroll bar, contains a pointer to the control. If the user clicked a pop-up window's scroll bar, this parameter is not used.

Remarks

Called when the user clicks a window's horizontal scroll bar.

The **SB_THUMBTRACK** notification 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 by using the **SetScrollPos** member function.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_HSCROLL** message.

See Also **CWnd::SetScrollPos, WM_VSCROLL, WM_HSCROLL, CWnd::Default**

CWnd::OnHScrollClipboard

Syntax

afx_msg void OnHScrollClipboard(CWnd* *pClipAppWnd*, **UINT** *nSBCode*, **UINT** *nPos* **);**

Parameters

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:

Value	Description
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.

The Clipboard owner should use the **Invalidate** or **InvalidateRect** member functions, or repaint as desired. The scroll-bar position should also be reset.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_HSCROLLCLIPBOARD** message.

See Also

CWnd::Invalidate, CWnd::OnVScrollClipboard, CWnd::InvalidateRect, CWnd::Default, WM_HSCROLLCLIPBOARD

CWnd::OnIconEraseBkgnd

Syntax

afx_msg void OnIconEraseBkgnd(CDC* *pDC* **);**

Parameters

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** 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; otherwise, the **WM_ERASEBKGND** message is sent instead.

The **DefWindowProc** member function fills the icon background with the background brush of the parent window.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_ICONERASEBKGND** message.

See Also

CWnd::Default, WM_ERASEBKGND, WM_ICONERASEBKGND

CWnd::OnInitMenu

Syntax

afx_msg void OnInitMenu(CMenu* *pMenu* **);**

Parameters

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 in order 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_INITMENU** message.

See Also

CWnd::OnInitMenuPopup, **CWnd::Default**, **WM_INITMENU**

CWnd::OnInitMenuPopup

Syntax

afx_msg void OnInitMenuPopup(CMenu* *pPopupMenu*, **UINT** *nIndex*, **BOOL** *bSysMenu* **);**

Parameters

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 system 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_INITMENUPOPUP** message.

See Also **CWnd::OnInitMenu**, **CWnd::Default**, **WM_INITMENUPOPUP**

CWnd::OnKeyDown

Syntax **afx_msg void OnKeyDown(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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, 0 otherwise).
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 indicating the previous key state can be used to determine whether the **OnKeyDown** 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 CONTROL keys on the main section of the keyboard; the INSERT, DELETE, 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*.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_KEYDOWN** message.

See Also

WM_CHAR, **WM_KEYUP**, **CWnd::Default**, **WM_KEYDOWN**

CWnd::OnKeyUp

Syntax

afx_msg void OnKeyUp(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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; 0 otherwise).
9–10	Not used.

Value	Description
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 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 CONTROL keys on the main section of the keyboard; the INSERT, DELETE, 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*.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_KEYUP** message.

See Also

WM_CHAR, **WM_KEYUP**, **CWnd::Default**, **WM_KEYDOWN**

CWnd::OnKillFocus

Syntax

afx_msg void OnKillFocus(CWnd* *pNewWnd* **);**

Parameters

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_KILLFOCUS** message.

See Also

CWnd::SetFocus, **CWnd::Default**, **WM_KILLFOCUS**

CWnd::OnLButtonDblClk

Syntax

afx_msg void OnLButtonDblClk(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags
Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_LBUTTONDBLCLK** message.

See Also

CWnd::OnLButtonDown, **CWnd::OnLButtonUp**, **CWnd::Default**, **WM_LBUTTONDBLCLK**

CWnd::OnLButtonDown

Syntax

afx_msg void OnLButtonDown(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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 **CWnd**.

Remarks

Called when the user presses the left mouse button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_LBUTTONDOWN** message.

See Also

CWnd::OnLButtonDblClk, **CWnd::OnLButtonUp**, **WM_LBUTTONDOWN**, **CWnd::Default**

CWnd::OnLButtonUp

Syntax

afx_msg void OnLButtonUp(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags
Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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 **CWnd**.

Remarks

Called when the user releases the left mouse button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_LBUTTONUP** message.

See Also

CWnd::OnLButtonDblClk, **CWnd::OnLButtonDown**, **WM_LBUTTONUP**, **CWnd::Default**

CWnd::OnMButtonDblClk

Syntax

afx_msg void OnMButtonDblClk(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags
Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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 **CWnd**.

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 Library windows. Windows generates a **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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MBUTTONDBLCLK** message.

See Also

CWnd::Default, **CWnd::OnMButtonDown**, **CWnd::OnMButtonUp**, **WM_MBUTTONDBLCLK**

CWnd::OnMButtonDown

Syntax

afx_msg void OnMButtonDown(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags
Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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 **CWnd**.

Remarks

Called when the user presses the middle mouse button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MBUTTONDOWN** message.

See Also

CWnd::OnMButtonDblClk, **CWnd::OnMButtonUp**, **CWnd::Default**, **WM_MBUTTONDOWN**

CWnd::OnMButtonUp

Syntax

afx_msg void OnMButtonUp(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags
Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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 **CWnd**.

Remarks

OnMButtonUp is called when the user releases the middle mouse button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MBUTTONUP** message.

See Also

CWnd::OnMButtonDblClk, **CWnd::OnMButtonDown**, **CWnd::Default**, **WM_MBUTTONUP**

CWnd::OnMDIActivate

Syntax

afx_msg void OnMDIActivate(BOOL *bActivate*, **CWnd*** *pActivatedWnd*, **CWnd*** *pDeactivatedWnd* **);**

Parameters

bActivate
When the client window calls a child window's **OnMDIActivate** member function, *bActivate* is **TRUE** if the child is being activated and **FALSE** if it is being deactivated.

pActivatedWnd
When the application calls its MDI client window's **OnMDIActivate** member function, *pActivatedWnd* contains a pointer to the MDI child window to be activated. When received by an MDI child window, *pActivatedWnd* contains a pointer to the child window being activated. This pointer may be temporary, and should not be stored for later use.

pDeactivatedWnd
When received by an MDI child window, *pDeactivatedWnd* contains a pointer to the child window being deactivated. This pointer may be temporary, and should not be stored for later use.

Remarks

An application calls the multiple document interface (MDI) **CMDIFrameWnd::MDIActivate** member function to activate a different MDI child window. The **OnMDIActivate** member function is 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 a **WM_NCACTIVATE** message to draw an active window frame and caption bar, but it does not receive another **OnMDIActivate** call.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MDIACTIVATE** message.

See Also

CMDIFrameWnd::MDIGetActive, CMDIFrameWnd::MDINext, CMDIFrameWnd::MDIActivate, WM_MDIACTIVATE, CWnd::Default

CWnd::OnMeasureItem

Syntax

afx_msg void OnMeasureItem
(LPMEASUREITEMSTRUCT *lpMeasureItemStruct* **);**

Parameters

lpMeasureItemStruct
Specifies a long pointer to a **MEASUREITEMSTRUCT** data structure that contains the dimensions of the owner-draw control.

Remarks

Called for the owner of an owner-draw button, combo box, list box, or menu item when the control is created. When the owner receives the call, the owner should 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, this function is called for the owner for each item in the control; otherwise, this function is called once.

Windows calls **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 requiring these values should occur in the main function of the application or library.

A **MEASUREITEMSTRUCT** data structure has the following form:

```
typedef struct tagMEASUREITEMSTRUCT {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemWidth;
    WORD   itemHeight;
    DWORD  itemData
} MEASUREITEMSTRUCT;
```

Members

CtlType
Is the control type. The values for control types are as follows:

Value	Meaning
ODT_BUTTON	Owner-draw button.
ODT_COMBOBOX	Owner-draw combo box.
ODT_LISTBOX	Owner-draw list box.
ODT_MENU	Owner-draw menu.

CtlID
Is the control ID for a combo box, list box, or button. This member is not used for a menu.

itemID
Is 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 returning from the message.

itemHeight
Specifies the height of an individual item in a list box or a menu. Before returning 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

Comments

Failure to fill out the proper members in the **MEASUREITEMSTRUCT** structure will cause improper operation of the control.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MEASUREITEM** message.

See Also

WM_INITDIALOG, **CWnd::Default**, **WM_MEASUREITEM**

CWnd::OnMenuChar

Syntax

afx_msg LONG OnMenuChar(UINT *nChar*, **UINT** *nFlags*, **CMenu*** *pMenu* **);**

Parameters

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MENUCHAR** message.

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 menu 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 accelerators are used to select bitmaps placed in a menu.

See Also

CWnd::Default, **WM_MENUCHAR**

CWnd::OnMenuSelect

Syntax

afx_msg void OnMenuSelect(UINT *nItemID*, **UINT** *nFlags*,
  HMENU *hSysMenu* **);**

Parameters

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:

Flag	Description
MF_BITMAP	Item is a bitmap.
MF_CHECKED	Item is checked.
MF_DISABLED	Item is disabled.

Flag	Description
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
Identifies the menu associated with the message, if *nFlags* contains **MF_SYSMENU**.

Remarks

If the **CWnd** is associated with a menu, **OnMenuSelect** is called when the user selects a menu item.

If *nFlags* contains –1 and *hSysMenu* contains 0, Windows has closed the menu because the user pressed ESC or clicked outside the menu.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MENUSELECT** message.

See Also

CWnd::Default, **WM_MENUSELECT**, **CMenu::FromHandle**

CWnd::OnMouseActivate

Syntax

afx_msg int OnMouseActivate(CWnd* *pFrameWnd*, **UINT** *nHitTest*, **UINT** *message* **);**

Parameters

pFrameWnd
Specifies a pointer to the topmost 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 message number.

Remarks

Called when the cursor is in an inactive window and the user presses a mouse button.

If the child window passes the message to the **Default** or **DefWindowProc** member function, **Default** or **DefWindowProc** passes this message to the **CWnd** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MOUSEACTIVATE** message.

Return Value

Specifies whether to activate the **CWnd** and whether to discard the mouse event. It must be one of the following values:

Value	Meaning
MA_ACTIVATE	Activate **CWnd**.
MA_NOACTIVATE	Do not activate **CWnd**.
MA_ACTIVATEANDEAT	Activate **CWnd** and discard the mouse event.
MA_NOACTIVATEANDEAT	Do not activate **CWnd** and discard the mouse event.

See Also

CWnd::OnNcHitTest, **CWnd::Default**, **WM_MOUSEACTIVATE**

CWnd::OnMouseMove

Syntax

afx_msg void OnMouseMove(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags
Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if 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 mouse cursor moves. If the mouse is not captured, the **WM_MOUSEMOVE** is sent to the **CWnd** beneath the mouse pointer; otherwise, the message goes to the mouse-capture window.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MOUSEMOVE** message.

See Also

CWnd::SetCapture, **CWnd::OnNCHitTest**, **WM_MOUSEMOVE**, **CWnd::Default**

CWnd::OnMove

Syntax

afx_msg void OnMove(int *x*, **int** *y* **);**

Parameters

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 **CWnd** has been moved.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_MOVE** message.

See Also

CWnd::Default, **WM_MOVE**

CWnd::OnNcActivate

Syntax

afx_msg BOOL OnNcActivate(BOOL *bActive* **);**

Parameters

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** member function draws the appropriate caption bar.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCACTIVATE** message.

Return Value

TRUE if activation should proceed; **FALSE** if activation should be aborted.

See Also

CWnd::Default, **WM_NCACTIVATE**

CWnd::OnNcCalcSize

Syntax **afx_msg void OnNcCalcSize(LPRECT** *lpRect* **);**

Parameters *lpRect*
Points to a **RECT** data structure that contains the screen coordinates of the **CWnd** rectangle (including client area, borders, caption, scroll bars, and so on).

Remarks Called when the size of the client area needs to be calculated.

The **Default** member function 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 *lpRect.*

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCCALCSIZE** message.

See Also **CWnd::Default**, **WM_NCCALCSIZE**

CWnd::OnNcCreate

Syntax **afx_msg BOOL OnNcCreate(LPCREATESTRUCT** *lpCreateStruct* **);**

Parameters *lpCreateStruct*
Points to the **CREATESTRUCT** data structure for **CWnd**.

Remarks Called prior to the **WM_CREATE** message when the **CWnd** is first created.

By default, scroll bars are initialized (the scroll-bar position and range are set) and the **CWnd** text is set. Memory used internally to create and maintain the window is allocated.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCCREATE** message.

Return Value **TRUE** if the nonclient area is created. It is **FALSE** if an error occurs; the **Create** function will return **NULL** in this case.

See Also **CWnd::CreateEx, WM_NCCREATE, CWnd::Default**

CWnd::OnNcDestroy

Syntax **afx_msg void OnNcDestroy();**

Remarks Called when the nonclient area is being destroyed. The **DestroyWindow** member function sends **WM_NCDESTROY**.

The **Default** member function frees any memory internally allocated for the Windows window.

The **OnNcDestroy** member function is the last member called when the Windows window is being destroyed. **OnNcDestroy** can call **delete this** to free the **CWnd** object that was dynamically allocated with **new**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCDESTROY** message.

See Also **CWnd::DestroyWindow, CWnd::OnNcCreate, WM_NCDESTROY, CWnd::Default**

CWnd::OnNcHitTest

Syntax **afx_msg UINT OnNcHitTest(CPoint** *point* **);**

Parameters

point
Contains the x- and y-coordinates of the cursor. These coordinates are always screen coordinates.

Remarks Called for the **CWnd** that contains the cursor (or the **CWnd** that used the **SetCapture** member function to capture the mouse input) every time the mouse is moved.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCHITTEST** message.

Return Value

One of the following values:

Value	Meaning
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 caption area.
HTCLIENT	In a client area.
HTERROR	Same as **HTNOWHERE** except that the **DefWindowProc** member 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.
HTMENU	In a menu area.
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	Same as **HTGROWBOX**.
HTSYSMENU	In a Control-menu box (close box in child windows).
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::Default, **CWnd::GetCapture**, **WM_NCHITTEST**

CWnd::OnNcLButtonDblClk

Syntax

afx_msg void OnNcLButtonDblClk(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCLBUTTONDBLCLK** message.

See Also

CWnd::Default, **WM_NCLBUTTONDBLCLK**, **CWnd::OnNcHitTest**

CWnd::OnNcLButtonDown

Syntax

afx_msg void OnNcLButtonDown(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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 **CWnd**.

If appropriate, the **WM_SYSCOMMAND** is sent.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCLBUTTONDOWN** message.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcLButtonDblClk**, **CWnd::OnNcLButtonUp**, **CWnd::OnSysCommand**, **WM_NCLBUTTONDOWN**, **CWnd::Default**

CWnd::OnNcLButtonUp

Syntax **afx_msg void OnNcLButtonUp(UINT** *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCLBUTTONUP** message.

See Also **CWnd::OnNcHitTest**, **CWnd::OnNcLButtonDown**, **CWnd::OnSysCommand**, **CWnd::Default**, **WM_NCLBUTTONUP**

CWnd::OnNcMButtonDblClk

Syntax

afx_msg void OnNcMButtonDblClk(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
: Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
: Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCMBUTTONDBLCLK** message.

See Also

CWnd::OnNcHitTest, **CWnd::OnNcMButtonDown**, **CWnd::OnNcMButtonUp**, **WM_NCMBUTTONDBLCLK**, **CWnd::Default**

CWnd::OnNcMButtonDown

Syntax

afx_msg void OnNcMButtonDown(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
: Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
: Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCMBUTTONDOWN** message.

See Also

CWnd::OnNcHitTest, CWnd::OnNcMButtonUp, WM_NCMBUTTONDOWN, CWnd::Default

CWnd::OnNcMButtonUp

Syntax

afx_msg void OnNcMButtonUp(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCMBUTTONUP** message.

See Also

CWnd::OnNcHitTest, CWnd::OnNcMButtonDblClk, CWnd::OnNcMButtonDown, WM_NCMBUTTONUP, CWnd::Default

CWnd::OnNcMouseMove

Syntax

afx_msg void OnNcMouseMove(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCMOUSEMOVE** message.

See Also

CWnd::OnNcHitTest, **CWnd::OnSysCommand**, **WM_NCMOUSEMOVE**, **CWnd::Default**

CWnd::OnNcPaint

Syntax

afx_msg void OnNcPaint();

Remarks

Called when the nonclient area needs to be painted.

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCPAINT** message.

See Also

CWnd::Default, **WM_NCPAINT**

CWnd::OnNcRButtonDblClk

Syntax

afx_msg void OnNcRButtonDblClk(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCRBUTTONDBLCLK** message.

See Also

CWnd::OnNcHitTest, **CWnd::OnNcRButtonDown**, **CWnd::OnNcRButtonUp**, **CWnd::Default**, **WM_NCRBUTTONDBLCLK**

CWnd::OnNcRButtonDown

Syntax

afx_msg void OnNcRButtonDown(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCRBUTTONDOWN** message.

See Also

CWnd::OnNcHitTest, **CWnd::OnNcRButtonDblClk**, **CWnd::OnNcRButtonUp**, **CWnd::Default**, **WM_NCRBUTTONDOWN**

CWnd::OnNcRButtonUp

Syntax

afx_msg void OnNcRButtonUp(UINT *nHitTest*, **CPoint** *point* **);**

Parameters

nHitTest
Specifies the hit-test code. A hit test is a test that determines the location of the cursor.

point
Specifies a **CPoint** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_NCRBUTTONUP** message.

See Also

CWnd::OnNcHitTest, **CWnd::OnNcRButtonDblClk**, **CWnd::OnNcRButtonDown**, **CWnd::Default**, **WM_NCRBUTTONUP**

CWnd::OnPaint

Syntax

afx_msg void OnPaint();

Remarks

Called when Windows or an application makes a request to repaint a portion of the **CWnd**.

The **WM_PAINT** message is sent to this member function when the **UpdateWindow** member function is called, or when the update region is not empty and there are no pending messages.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_PAINT** message.

See Also

CWnd::UpdateWindow, **CPaintDC**, **CWnd::Default**, **WM_PAINT**

CWnd::OnPaintClipboard

Syntax

afx_msg void OnPaintClipboard(CWnd* *pClipAppWnd*,
HANDLE *hPaintStruct* **);**

Parameters

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 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 by using the **GlobalUnlock** Windows function before it exits.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_PAINTCLIPBOARD** message.

See Also **::GlobalLock**, **::GlobalUnlock**, **CWnd::OnSizeClipboard**, **WM_PAINTCLIPBOARD**, **CWnd::Default**

CWnd::OnPaintIcon

Syntax **afx_msg void OnPaintIcon();**

Remarks Called by a minimized (iconic) **CWnd** when the icon is to be painted.

A window receives this call only if a class icon is defined for the window; otherwise, the **OnPaint** member function is called instead. **OnPaintIcon** permits Windows to paint the icon with the class icon.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_PAINTICON** message.

See Also **WM_PAINT**, **WM_PAINTICON**, **CWnd::Default**

CWnd::OnPaletteChanged

Syntax **afx_msg void OnPaletteChanged(CWnd*** *pFocusWnd* **);**

Parameters

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 after the window with input focus has realized its logical palette, thereby changing the system palette. This call allows windows without the input focus that use a color palette to realize their logical palettes and update their client areas.

To avoid creating a loop, **CWnd** shouldn't realize its palette unless it determines that *pFocusWnd* does not contain a pointer to itself.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_PALETTECHANGED** message.

See Also **::RealizePalette, WM_PALETTECHANGED, CWnd::Default**

CWnd::OnParentNotify

Syntax **afx_msg void OnParentNotify(UINT** *message*, **LONG** *lParam* **);**

Parameters

message
Specifies the event for which the parent is being notified. It can be any of these values:

Value	Description
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
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 if *message* is **WM_CREATE** or **WM_DESTROY**; 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 **CWnd** returns. When the child window is being destroyed, the system calls **OnParentNotify** before any processing takes place to destroy **CWnd**.

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_PARENTNOTIFY** message.

See Also **CWnd::CreateEx**, **CWnd::OnCreate**, **CWnd::OnDestroy**, **CWnd::OnLButtonDown**, **CWnd::OnMButtonDown**, **CWnd::OnRButtonDown**, **WM_PARENTNOTIFY**, **CWnd::Default**

CWnd::OnQueryDragIcon

Syntax **afx_msg HCURSOR OnQueryDragIcon();**

Remarks Called by a minimized (iconic) **CWnd** if it is about to be dragged by the user and it 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 an icon handle, the system converts the icon to a black-and-white cursor.

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_QUERYDRAGICON** message.

Return Value Returns a cursor or icon handle. If **NULL** is returned, the system displays the default cursor. The default return value is **NULL**.

See Also **CWinApp::LoadCursor, CWinApp::LoadIcon, WM_QUERYDRAGICON, CWnd::Default**

CWnd::OnQueryEndSession

Syntax **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 **FALSE**, the Windows session is not ended. Windows stops calling **OnQueryEndSession** as soon as one application returns **FALSE**, and sends the **WM_ENDSESSION** message with a parameter value of **FALSE** for any applications that have already returned **TRUE**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_QUERYENDSESSION** message.

Return Value An application should return **TRUE** if it can be conveniently shut down; otherwise **FALSE**.

See Also **CWnd::Default, ::ExitWindows, CWnd::OnEndSession, WM_QUERYENDSESSION**

CWnd::OnQueryNewPalette

Syntax

afx_msg BOOL OnQueryNewPalette();

Remarks

Called when the **CWnd** is about to receive the input focus. If the **CWnd** realizes its logical palette when it receives the input focus, it should return **TRUE**; otherwise, it should return **FALSE**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_QUERYNEWPALETTE** message.

Return Value

TRUE if the **CWnd** realizes its logical palette; otherwise **FALSE**.

See Also

CWnd::Default, **WM_QUERYNEWPALETTE**

CWnd::OnQueryOpen

Syntax

afx_msg BOOL OnQueryOpen();

Remarks

Called when the **CWnd** is iconized and the user requests that the **CWnd** be opened into a window.

While in **OnQueryOpen**, **CWnd** should not perform any action that would cause an activation or focus change (for example, creating a dialog box).

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_QUERYOPEN** message.

Return Value

TRUE if the icon can be opened, or **FALSE** to prevent the icon from being opened.

See Also

CWnd::Default, **WM_QUERYOPEN**

CWnd::OnRButtonDblClk

Syntax

afx_msg void OnRButtonDblClk(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if SHIFT key is down.

point

Specifies the x- and y-coordinates of the cursor. These coordinates are always relative to the upper-left corner.

Remarks

Called when the user double-clicks the right mouse button.

Only windows that have the **CS_DBLCLKS** class 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 presses the right mouse button again within the system's double-click time limit. Double-clicking the right mouse button actually generates four events: a **WM_RBUTTONDOWN** and **WM_RBUTTONUP** message, the **OnRButtonDblClk** call, and another **WM_RBUTTONUP** message when the button is released.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_RBUTTONDBLCLK** message.

See Also

CWnd::OnRButtonDown, **CWnd::OnRButtonUp**, **WM_RBUTTONDBLCLK**, **CWnd::Default**

CWnd::OnRButtonDown

Syntax

afx_msg void OnRButtonDown(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if SHIFT key is down.

point

Specifies the x- and y-coordinates of the cursor. These coordinates are always relative to the upper-left corner.

Remarks

Called when the user presses the right mouse button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_RBUTTONDOWN** message.

See Also

CWnd::OnRButtonDblClk, CWnd::OnRButtonUp, WM_RBUTTONDOWN, CWnd::Default

CWnd::OnRButtonUp

Syntax

afx_msg void OnRButtonUp(UINT *nFlags*, **CPoint** *point* **);**

Parameters

nFlags

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_RBUTTON	Set if right mouse button is down.
MK_SHIFT	Set if SHIFT key is down.

point

Specifies the x- and y-coordinates of the cursor. These coordinates are always relative to the upper-left corner.

Remarks

Called when the user releases the right mouse button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_RBUTTONUP** message.

See Also

CWnd::OnRButtonDblClk, **CWnd::OnRButtonDown**, **WM_RBUTTONUP**, **CWnd::Default**

CWnd::OnRenderAllFormats

Syntax

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_RENDERALLFORMATS** message.

See Also

::CloseClipboard, CWnd::OpenClipboard, ::SetClipboardData, CWnd::OnRenderFormat, WM_RENDERALLFORMATS, CWnd::Default

CWnd::OnRenderFormat

Syntax

afx_msg void OnRenderFormat(UINT *nFormat* **);**

Parameters

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**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_RENDERFORMAT** message.

See Also

::CloseClipboard, CWnd::OpenClipboard, ::SetClipboardData, WM_RENDERFORMAT, CWnd::Default

CWnd::OnSetCursor

Syntax

afx_msg BOOL OnSetCursor(CWnd* *pWnd*, **UINT** *nHitTest*, **UINT** *message* **);**

Parameters

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

OnSetCursor is called if mouse input is not captured and the mouse causes cursor movement within the **CWnd**.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SETCURSOR** message.

By default, **OnSetCursor** 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.

By default, **OnSetCursor** also 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

The return value is ignored by Windows, but is used by the **Default** member function when it calls the parent to determine if the parent handled the message. **TRUE** means that the message was handled; otherwise **FALSE**.

See Also

CWnd::OnNcHitTest, **CWnd::Default**, **WM_SETCURSOR**

CWnd::OnSetFocus

Syntax

afx_msg void OnSetFocus(CWnd* *pOldWnd* **);**

Parameters

pOldWnd
Contains the **CWnd** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SETFOCUS** message.

See Also

CWnd::Default, **WM_SETFOCUS**

CWnd::OnShowWindow

Syntax

afx_msg void OnShowWindow(BOOL *bShow*, **UINT** *nStatus* **);**

Parameters

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 values:

Value	Description
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** 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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SHOWWINDOW** message.

See Also **CWnd::Default**, **WM_SHOWWINDOW**

CWnd::OnSize

Syntax

afx_msg void OnSize(UINT *nType*, **int** *cx*, **int** *cy* **);**

Parameters

nType
Specifies the type of resizing requested. This parameter can be one of the following values:

Value	Description
SIZEFULLSCREEN	Window has been maximized.
SIZEICONIC	Window has been minimized.
SIZENORMAL	Window has been resized, but neither **SIZEICONIC** nor **SIZEFULLSCREEN** applies.
SIZEZOOMHIDE	Message is sent to all pop-up windows when some other window is maximized.
SIZEZOOMSHOW	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 size has changed.

If the **SetScrollPos** or **MoveWindow** member function is called for a child window from **OnSize**, the *bRedraw* parameter should be nonzero to cause the **CWnd** to be repainted.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SIZE** message.

See Also **CWnd::MoveWindow**, **CWnd::SetScrollPos**, **CWnd::Default**, **WM_SIZE**

CWnd::OnSizeClipboard

Syntax **afx_msg void OnSizeClipboard(CWnd*** *pClipAppWnd*, **HANDLE** *hRect* **);**

Parameters

pClipAppWnd
Identifies the Clipboard-application window. The pointer may be temporary and should not be stored.

hRect
Identifies a handle to 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 Clipboard-viewer window's client area 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 by using the **GlobalUnlock** Windows function before it yields or returns control.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SIZECLIPBOARD** message.

See Also **::GlobalLock**, **::GlobalUnlock**, **::SetClipboardData**, **CWnd::SetClipboardViewer**, **CWnd::Default**, **WM_SIZECLIPBOARD**

CWnd::OnSpoolerStatus

Syntax

afx_msg void OnSpoolerStatus(UINT *nStatus*, **UINT** *nJobs* **);**

Parameters

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.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SPOOLERSTATUS** message.

See Also

CWnd::Default, **WM_SPOOLERSTATUS**

CWnd::OnSysChar

Syntax

afx_msg void OnSysChar(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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	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, 0 otherwise).
9–10	Not used.

Value	Description
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).

Remarks

Called if **CWnd** has the input focus and the **WM_SYSKEYUP** or **WM_SYSKEYDOWN** message is received. 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 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.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, 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*.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SYSCHAR** message.

See Also

::TranslateAccelerator, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**, **CWnd::Default**, **WM_SYSCHAR**

CWnd::OnSysColorChange

Syntax

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 by using the new system colors.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SYSCOLORCHANGE** message.

See Also

::SetSysColors, **WM_PAINT**, **CWnd::Default**, **WM_SYSCOLORCHANGE**

CWnd::OnSysCommand

Syntax

afx_msg void OnSysCommand(UINT *nID*, **LONG** *lParam* **);**

Parameters

nID
Specifies the type of system command requested. This parameter can be one of the following values:

Value	Description
SC_CLOSE	Close **CWnd**.
SC_HOTKEY	Activate the **CWnd** associated with the application-specified hot key.
SC_HSCROLL	Scroll horizontally.
SC_KEYMENU	Retrieve a menu through a keystroke.

Value	Description
SC_MAXIMIZE (or **SC_ZOOM**)	Maximize **CWnd**.
SC_MINIMIZE (or **SC_ICON**)	Minimize **CWnd**.
SC_MOUSEMENU	Retrieve a menu through a mouse click.
SC_MOVE	Move **CWnd**.
SC_NEXTWINDOW	Move to the next window.
SC_PREVWINDOW	Move to the previous window.
SC_RESTORE	Checkpoint (save the previous coordinates).
SC_SCREENSAVE	Executes the screen-save application specified in the Desktop section of the Control Panel.
SC_SIZE	Size **CWnd**.
SC_TASKLIST	Executes or activates the **CWnd** Task Manager application.
SC_VSCROLL	Scroll vertically.

lParam
Contains the cursor coordinates if a Control-menu command is chosen with the mouse. 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 Minimize button.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SYSCOMMAND** message.

By default, **OnSysCommand** carries out the Control-menu request for the predefined actions specified above.

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**.

Access (sometimes called "accelerator") keystrokes that are defined to select items from the Control menu are translated into **OnSysCommand** calls; all other access keystrokes are translated into **WM_COMMAND** messages.

See Also **CWnd::Default**, **WM_SYSCOMMAND**

CWnd::OnSysDeadChar

Syntax **afx_msg void OnSysDeadChar(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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, 0 otherwise).
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 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 **CWnd** has the input focus when the **OnSysKeyUp** or **OnSysKeyDown** member functions are called. It specifies the character value of a dead key.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SYSDEADCHAR** message.

See Also **CWnd::OnSysKeyDown**, **CWnd::OnSysKeyUp**, **CWnd::Default**, **WM_SYSDEADCHAR**, **CWnd::OnDeadChar**

CWnd::OnSysKeyDown

Syntax **afx_msg void OnSysKeyDown(UINT** *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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	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, 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** 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** 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 CONTROL keys on the main section of the keyboard; the INSERT, DELETE, 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*.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SYSKEYDOWN** message.

See Also

::TranslateAccelerator, **WM_SYSKEYUP**, **CWnd::Default**, **WM_SYSKEYDOWN**

CWnd::OnSysKeyUp

Syntax

afx_msg void OnSysKeyUp(UINT *nChar*, **UINT** *nRepCnt*, **UINT** *nFlags* **);**

Parameters

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	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, 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** 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** 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 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 CONTROL keys on the main section of the keyboard; the INSERT, DELETE, 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-USA Enhanced 102-key keyboards, the right ALT key is handled as a CONTROL-ALT key. 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**

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_SYSKEYUP** message.

See Also **::TranslateAccelerator**, **WM_SYSKEYDOWN**, **CWnd::Default**, **WM_SYSKEYUP**

CWnd::OnTimeChange

Syntax **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 the *hWnd* parameter set to 0xFFFF.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_TIMECHANGE** message.

See Also **::SendMessage**, **CWnd::Default**, **WM_TIMECHANGE**

CWnd::OnTimer

Syntax **afx_msg void OnTimer(UINT** *nIDEvent* **);**

Parameters

nIDEvent
Specifies the identifier of the timer.

Remarks Called after each interval specified in the **SetTimer** member function used to install a timer.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_TIMER** message.

See Also **CWnd::SetTimer**, **WM_TIMER**, **CWnd::Default**

CWnd::OnVKeyToItem

Syntax **afx_msg int OnVKeyToItem(UINT** *nKey*, **CWnd*** *pListBox*, **UINT** *nIndex* **);**

Parameters

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.

nIndex
Specifies the current caret position.

Remarks If the **CWnd** 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 message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_VKEYTOITEM** message.

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 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 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::Default**

CWnd::OnVScroll

Syntax

afx_msg void OnVScroll(UINT *nSBCode*, **UINT** *nPos*, **CWnd*** *pScrollBar* **);**

Parameters

nSBCode

Contains a scroll-bar code that specifies the user's scrolling request. This parameter can be one of the following values:

Value	Description
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 scroll-box position if the scroll-bar code is **SB_THUMBPOSITION**; otherwise, not used.

pScrollBar
If the message is sent by a scroll-bar control, *pScrollBar* identifies the control. If the message is sent as a result of the user clicking a pop-up window's scroll bar, *pScrollBar* is not used. The pointer may be temporary.

Remarks

Called when the user clicks a 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 **CWnd**, it must also reset the position of the scroll box by using the **SetScrollPos** member function.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_VSCROLL** message.

See Also

CWnd::SetScrollPos, CWnd::OnHScroll, WM_VSCROLL, CWnd::Default

CWnd::OnVScrollClipboard

Syntax

afx_msg void OnVScrollClipboard(CWnd* *pClipAppWnd*, **UINT** *nSBCode*, **UINT** *nPos* **);**

Parameters

pClipAppWnd
Specifies a pointer to a Clipboard-viewer window. The pointer may be temporary.

nSBCode
Specifies one of the following scroll-bar codes:

Value	Description
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.

Value	Description
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, *nPos* is not used.

Remarks

If the **CWnd** owns the Clipboard, the **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. **OnVScrollClipboard** should scroll the Clipboard image, invalidate the appropriate section, and update the scroll-bar values.

The Clipboard owner should use the **Invalidate** or **InvalidateRect** member function or repaint as desired. The scroll-bar position should also be reset.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_VSCROLLCLIPBOARD** message.

See Also

CWnd::Invalidate, **CWnd::OnHScrollClipboard**, **CWnd::InvalidateRect**, **WM_VSCROLLCLIPBOARD**, **CWnd::Default**

CWnd::OnWinIniChange

Syntax

afx_msg void OnWinIniChange(LPSTR *lpSection* **);**

Parameters

lpSection
Points to a string that specifies the name of the section that has changed (the string does not include the square brackets).

Remarks

Called after a change has been made to the Windows initialization file, WIN.INI.

To send the **WM_WININICHANGE** message to all top-level windows, an application can use the **SendMessage** Windows function with the *hWnd* parameter set to 0xFFFF.

If an application changes many different sections in WIN.INI at the same time, the application should send one **WM_WININICHANGE** message with *lpSection* 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 *lpSection* set to **NULL**, the application should check all sections in WIN.INI that affect the application.

This message-handler member function calls the **Default** member function. Override this member function in your derived class to handle the **WM_WININICHANGE** message.

See Also **::SendMessage**, **::SystemParametersInfo**, **WM_WININICHANGE**, **CWnd::Default**

CWnd::OpenClipboard

Syntax **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 **TRUE** if the Clipboard is opened via **CWnd**, or **FALSE** if another application or window has the Clipboard opened.

See Also **::CloseClipboard**, **::EmptyClipboard**, **::OpenClipboard**

CWnd::OpenIcon

Syntax **BOOL OpenIcon();**

Remarks Activates and displays a minimized (iconic) window. Windows restores the window to its original size and position.

Return Value **TRUE** if the function is successful; otherwise **FALSE**.

See Also **::OpenIcon, CWnd::CloseWindow**

CWnd::PostMessage

Syntax **BOOL PostMessage(UINT** *message*, **UINT** *wParam* **= 0, LONG** *lParam* **= 0);**

Parameters

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 current **CWnd** object'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 functions.

An application should never use the **PostMessage** member function to send a message to a control; it should use **SendDlgItemMessage**.

The Windows **PostMessage** function can be used to access another application.

Return Value

TRUE if the message is posted; otherwise **FALSE**.

See Also

::GetMessage, **::PeekMessage**, **::PostMessage**, **::PostAppMessage**, **CWnd::SendMessage**, **CWnd::SendDlgItemMessage**

CWnd::PreTranslateMessage

Syntax

Protected:
  virtual BOOL PreTranslateMessage(MSG* *pMsg* **);**

Parameters

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 by the **DispatchMessage** Windows function.

Return Value

TRUE if the message is translated and should not be dispatched; **FALSE** if the message was not translated and should be dispatched.

See Also

::TranslateMessage, **::IsDialogMessage**, **CWinApp::PreTranslateMessage**

CWnd::ReleaseDC

Syntax

int ReleaseDC(CDC* *pDC* **);**

Parameters

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. It only frees common and window device contexts. It has no effect on class or private device contexts.

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 that retrieves a common device context.

Return Value

Specifies whether the device context is released. It is 1 if the device context is released; otherwise 0.

See Also

CWnd::GetDC, **CWnd::GetWindowDC**, **::ReleaseDC**

CWnd::ScreenToClient

Syntax

void ScreenToClient(LPPOINT *lpPoint* **) const;**

void ScreenToClient(LPRECT *lpRect* **) const;**

Parameters

lpPoint
Points to a **CPoint** or **POINT** structure that contains the screen coordinates to be converted.

lpRect
Points to a **CRect** or **RECT** structure that contains the screen coordinates to be converted.

Remarks

Converts the screen coordinates of a given point or rect on the display to client coordinates.

The **ScreenToClient** member function uses **CWnd** and the screen coordinates given in *lpPoint* or *lpRect* to compute client coordinates, and then replaces the screen coordinates with the client coordinates. The new coordinates are relative to the upper-left corner of the **CWnd** client area.

The **ScreenToClient** formula assumes the given point is in screen coordinates.

See Also

CWnd::ClientToScreen ::ScreenToClient

CWnd::ScrollWindow

Syntax

void ScrollWindow(int *xAmount*, **int** *yAmount*, **LPRECT** *lpRect* = **NULL**,
LPRECT *lpClipRect* = **NULL);**

Parameters

xAmount
Specifies the amount (in device units) to scroll in the x-axis direction.

yAmount
Specifies the amount (in device units) to scroll in the y-axis direction.

lpRect
Points to a **CRect** or **RECT** structure that specifies the portion of the client area to be scrolled. If *lpRect* is **NULL**, the entire client area is scrolled.

lpClipRect
Points to a **CRect** or **RECT** structure that specifies the clipping rectangle to be scrolled. Only bits inside this rectangle are scrolled. If *lpClipRect* is **NULL**, the entire window is scrolled.

Remarks

Scrolls the current **CWnd** object by moving the contents of the window's client area the number of units specified by *xAmount* along the screen's x-axis and the number of units specified by *yAmount* along the y-axis. The scroll moves right if *xAmount* is positive and left if it is negative. The scroll moves down if *yAmount* is positive and up if it is negative.

If the caret is in the **CWnd** being scrolled, **ScrollWindow** automatically hides the caret to prevent it from being erased, 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 **WM_PAINT** message will be sent, 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::SendDlgItemMessage

Syntax

LONG SendDlgItemMessage(int *nID*, **UINT** *message*, **UINT** *wParam* **= 0,**
LONG *lParam* **= 0);**

Parameters

nID
Specifies the integer identifier of the dialog item that is to receive the message.

message
Specifies the message value.

wParam
Specifies additional message information. The content of this parameter depends on the message being sent.

lParam
Specifies additional message information. The content of this parameter depends on the message being sent.

Remarks

Sends a message to the specified 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 identifier is not valid.

See Also **CWnd::SendMessage**, **::SendDlgItemMessage**

CWnd::SendMessage

Syntax

LONG SendMessage(UINT *message*, **UINT** *wParam* **= 0, LONG** *lParam* **= 0);**

Parameters

message
Specifies the message to be sent.

wParam
Specifies additional message information. The content of this parameter depends on the message being sent.

lParam
Specifies additional message information. The content of this parameter depends on the message being sent.

Remarks

Sends a message to a window or windows. The **SendMessage** member function calls the window procedure for the current **CWnd** object 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 **CWnd** message queue and returns immediately.

Return Value

The result returned by the invoked window procedure; its value depends on the message being sent.

See Also

::InSendMessage, **CWnd::PostMessage**, **CWnd::SendDlgItemMessage**, **::SendMessage**

CWnd::SetActiveWindow

Syntax

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 identity of the window that was previously active.

The returned pointer may be temporary, and should not be stored.

See Also

::SetActiveWindow, **CWnd::GetActiveWindow**

CWnd::SetCapture

Syntax

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** Windows 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.

See Also

::ReleaseCapture, **::SetCapture**, **CWnd::GetCapture**

CWnd::SetCaretPos

Syntax

static void SetCaretPos(POINT *point* **);**

Parameters

point
Specifies the new logical x- and y-coordinates of the caret.

Remarks

Moves the caret to the position given by logical coordinates specified by *point*. Logical coordinates are relative to the client area and are affected by the mapping mode, so the exact position in pixels depends on this mapping mode.

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 **CWnd** should not move the caret if it does not own the caret.

See Also **CWnd::GetCaretPos**, **::SetCaretPos**

CWnd::SetClipboardViewer

Syntax **HWND SetClipboardViewer();**

Remarks Adds **CWnd** to the Clipboard-viewer chain and returns a handle to the next window in the chain.

A **CWnd** 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 **CWnd**. Since the handle to the next window in the Clipboard-viewer chain has not yet been returned, the application should be careful to not pass on the **WM_DRAWCLIPBOARD** message that it receives during the call to **SetClipboardViewer**.

If an application wishes to remove itself from the Clipboard-viewer chain, it must call the **ChangeClipboardChain** member function.

Return Value A handle to the next window in the Clipboard-viewer chain. This handle should be saved in static memory and used to pass on Clipboard-viewer chain messages.

See Also **CWnd::ChangeClipboardChain**, **::SetClipboardViewer**

CWnd::SetDlgItemInt

Syntax

void SetDlgItemInt(int *nID*, **UINT** *nValue*, **BOOL** *bSigned* = **TRUE);**

Parameters

nID
Specifies the integer ID of the control to be modified.

nValue
Specifies the value to be set.

bSigned
Specifies whether or not the integer value is signed.

Remarks

Sets the text of a control to the string that represents the integer value given by *nValue*. The **SetDlgItemInt** member function converts *nValue* to a string that consists of decimal digits, and then copies the string to the control.

If *bSigned* is **TRUE**, *nValue* is assumed to be signed. If *nValue* is signed and less than 0, the function places a minus sign before the first digit in the string.

See Also

CWnd::GetDlgItemInt, **::SetDlgItemInt**, **WM_SETTEXT**

CWnd::SetDlgItemText

Syntax

void SetDlgItemText(int *nID*, **const char FAR*** *lpString* **);**

Parameters

nID
Specifies the integer ID of the control whose text is to be set.

lpString
Points to a **CString** or null-terminated string that is to be copied to the control.

Remarks

Sets the caption or text of a control owned by **CWnd**.

See Also

::SetDlgItemText, **WM_SETTEXT**, **CWnd::GetDlgItemText**

CWnd::SetFocus

Syntax

CWnd* SetFocus();

Remarks

Claims the input focus. The input focus directs all subsequent keyboard input to **CWnd**. The window, if any, that previously had the input focus loses it.

The **SetFocus** member function sends a **WM_KILLFOCUS** message to the **CWnd** that loses the input focus and a **WM_SETFOCUS** message to the **CWnd** that receives the input focus. It also activates either the **CWnd** or its parent.

If the current **CWnd** 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

Syntax

void SetFont(CFont* *pFont*, **BOOL** *bRedraw* = **TRUE);**

Parameters

pFont
Specifies the new font.

bRedraw
If **TRUE**, redraw the **CWnd**; otherwise **FALSE**.

Remarks

Sets the **CWnd** current font to **CFont**. If *bRedraw* is **TRUE**, **CWnd** will also be redrawn.

See Also

CWnd::GetFont, **WM_SETFONT**

CWnd::SetMenu

Syntax **BOOL SetMenu(CMenu*** *pMenu* **);**

Parameters

pMenu
Identifies the new menu. If this parameter is **NULL**, the current menu is removed.

Remarks Sets the current menu to the specified menu.

SetMenu will not destroy a previous menu. An application should call the **CMenu::DestroyMenu** member function to accomplish this task.

Return Value **TRUE** if the menu is changed; otherwise **FALSE**.

See Also **CMenu::DestroyMenu**, **CMenu::LoadMenu**, **::SetMenu**, **CWnd::GetMenu**

CWnd::SetParent

Syntax **CWnd* SetParent(CWnd*** *pWndNewParent* **);**

Parameters

pWndNewParent
Identifies the new parent window.

Remarks Changes the parent window of a child window.

If the **CWnd** is visible, Windows performs the appropriate redrawing and repainting.

Return Value A pointer to the previous parent window object. The returned pointer may be temporary.

See Also **::SetParent**, **CWnd::GetParent**

CWnd::SetRedraw

Syntax

void SetRedraw(BOOL *bRedraw* **= TRUE);**

Parameters

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, 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

Syntax

int SetScrollPos(int *nBar*, **int** *nPos*, **BOOL** *bRedraw* **= TRUE);**

Parameters

nBar
Specifies the scroll bar to be set. It can be one of the following values:

Value	Meaning
SB_CTL	Sets the position of a scroll-bar control. In this case, the **CWnd** must be a scroll-bar control.
SB_HORZ	Sets the **CWnd** horizontal scroll-bar position.
SB_VERT	Sets the **CWnd** vertical scroll-bar position.

nPos
Specifies the new position. 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 **FALSE**, it is not redrawn.

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.

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**

CWnd::SetScrollRange

Syntax

void SetScrollRange(int *nBar*, **int** *nMinPos*, **int** *nMaxPos*,
  BOOL *bRedraw* = **TRUE);**

Parameters

nBar
  Specifies the scroll bar to be set. It can be one of the following values:

Value	Meaning
SB_CTL	Sets the position of a scroll-bar control. In this case, the **CWnd** must be a scroll-bar control.
SB_HORZ	Sets the **CWnd** horizontal scroll-bar position.
SB_VERT	Sets the **CWnd** vertical scroll-bar position.

nMinPos
  Specifies the minimum scrolling position.

nMaxPos
  Specifies the maximum scrolling position.

bRedraw
  Specifies whether or not 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.

Remarks

Sets minimum and maximum position values for the given scroll bar. It can also be used to hide or show standard scroll bars by setting *nMinPos* and *nMaxPos* to 0.

An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

If **SetScrollRange** immediately follows the **SetScrollPos** member function, the *bRedraw* parameter in the **SetScrollPos** member function should be set to **FALSE** to prevent the scroll bar from being drawn twice.

The difference between the values specified by *nMinPos* and *nMaxPos* must not be greater than 32,767.

See Also **CWnd::SetScrollPos**, **::SetScrollRange**, **CWnd::GetScrollRange**

CWnd::SetSysModalWindow

Syntax **CWnd* SetSysModalWindow();**

Remarks Makes the **CWnd** a system-modal window.

If another window is made the active window (for example, the system-modal window creates a dialog box that becomes the active window), the active window becomes the system-modal window. When the original window becomes active again, it is system modal. To end the system-modal state, destroy the system-modal window.

Return Value A pointer to the window object that was previously the system-modal window. The returned pointer may be temporary.

See Also **::SetSysModalWindow**, **CWnd::GetSysModalWindow**

CWnd::SetTimer

Syntax

UINT SetTimer(int *nIDEvent*, **UINT** *nElapse*,
  UINT (FAR PASCAL EXPORT* *lpfnTimer*)**(HWND, UINT, int, DWORD));**

Parameters

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 handled by the **CWnd**.

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-supplied **TimerProc** callback function.

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.

The *lpfnTimer* callback function need not be named `TimerProc`, but it must be defined as follows, and return 0.

```
UINT FAR PASCAL EXPORT TimerProc(
        HWND hWnd,              //handle of CWnd that called SetTimer
        UINT nMsg,              //WM_TIMER
        int nIDEvent            //timer identification
        DWORD dwTime            //system time
);
```

Return Value

The timer identifier to use in **KillTimer** if the function is successful; otherwise 0.

See Also

WM_TIMER, **CWnd::KillTimer**, **::SetTimer**, **CWnd::FromHandle**

CWnd::SetWindowPos

Syntax

void SetWindowPos(const CWnd* *pWndInsertAfter*, **int** *x*, **int** *y*, **int** *cx*, **int** *cy*, **UINT** *nFlags* **);**

Parameters

pWndInsertAfter
Identifies a **CWnd** in the window-manager's list that will precede the positioned window.

x
Specifies the x-coordinate of the new upper-left corner.

y
Specifies the y-coordinate of the new upper-left corner.

cx
Specifies the new window's width.

cy
Specifies the new window's height.

nFlags
Specifies sizing and positioning options. It can be a combination of the following values:

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the **CWnd** class description) around the window.
SWP_HIDEWINDOW	Hides the **CWnd**.
SWP_NOACTIVATE	Does not activate the **CWnd**.
SWP_NOMOVE	Retains current position (ignores the *x* and *y* parameters).
SWP_NOREDRAW	Does not redraw changes.
SWP_NOSIZE	Retains current size (ignores the *cx* and *cy* parameters).
SWP_NOZORDER	Retains current ordering (ignores *pWndInsertAfter*).
SWP_SHOWWINDOW	Displays the **CWnd**.

Remarks

Changes the size, position, and ordering of child, pop-up, and top-level windows. Child, pop-up, and top-level windows are ordered according to their appearance on the screen; the topmost window receives the highest rank, and it is the first window in the list. This ordering is recorded in a window list.

If the **SWP_NOACTIVATE** flag is not specified, the *pWndInsertAfter* parameter is ignored and **CWnd** is activated and placed at the top of the Z order, in front of any other windows.

If the **SWP_NOZORDER** flag is not specified, Windows places **CWnd** in the position following the window identified by *pWndInsertAfter*. If *pWndInsertAfter* is **&wndTop**, Windows places **CWnd** at the top of the list. If *pWndInsertAfter* is set to **&wndBottom**, Windows places **CWnd** at the bottom of the list.

If the **SWP_SHOWWINDOW** or the **SWP_HIDEWINDOW** flag is set, **CWnd** cannot be moved or sized.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

See Also **::DeferWindowsPos**, **::SetWindowPos**

CWnd::SetWindowText

Syntax **void SetWindowText(const char FAR*** *lpString* **);**

Parameters

lpString
Points to a **CString** or null-terminated string.

Remarks Sets the caption title (if one exists) to the specified text. If the **CWnd** is a control, the **SetWindowText** member function sets the text within the control instead of within the caption. This function sends a **WM_SETTEXT** message to **CWnd**.

See Also **CWnd::GetWindowText**, **::SetWindowText**

CWnd::ShowCaret

Syntax **void ShowCaret();**

Remarks Shows the caret on the display 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 in a row. If the caret is not owned by **CWnd**, the caret is not shown.

Hiding the caret is accumulative. If the **HideCaret** member function has been called five times in a row, **ShowCaret** must be called five times to show the caret.

The caret is a shared resource. The **CWnd** should show the caret only when it has the input focus or is active.

See Also

CWnd::HideCaret, **::ShowCaret**

CWnd::ShowOwnedPopups

Syntax

void ShowOwnedPopups(BOOL *bShow* **= TRUE);**

Parameters

bShow
Specifies whether pop-up windows are hidden. It is **TRUE** if all hidden pop-up windows should be shown; it is **FALSE** if all visible pop-up windows should be hidden.

Remarks

Shows or hides all pop-up windows associated with the current **CWnd** object.

See Also

::ShowOwnedPopups

CWnd::ShowScrollBar

Syntax

void ShowScrollBar(UINT *nBar*, **BOOL** *bShow* **= TRUE);**

Parameters

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 values:

Value	Meaning
SB_BOTH	Specifies the **CWnd**'s horizontal and vertical scroll bars.
SB_CTL	Specifies that the **CWnd** is a scroll-bar control.
SB_HORZ	Specifies the **CWnd**'s horizontal scroll bar.
SB_VERT	Specifies the **CWnd**'s vertical scroll bar.

bShow
Specifies whether or not Windows hides the scroll bar. If *bShow* is **FALSE**, the scroll bar is hidden. Otherwise, the scroll bar is displayed.

Remarks

Displays or hides a scroll bar, depending on the value of *bShow*. If *bShow* is **TRUE**, the scroll bar is displayed; if **FALSE**, the scroll bar is hidden.

See Also

::ShowScrollBar

CWnd::ShowWindow

Syntax

BOOL ShowWindow(int *nCmdShow* **);**

Parameters

nCmdShow
Specifies how the **CWnd** is to be shown. It must be one of the following values:

Value	Meaning
SW_HIDE	Hides **CWnd** and passes activation to another window.
SW_MINIMIZE	Minimizes **CWnd** and activates the top-level window in the window-manager's list.
SW_RESTORE	Same as **SW_SHOWNORMAL**.
SW_SHOW	Activates **CWnd** and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates **CWnd** and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates **CWnd** and displays it as a minimized (iconic) window.
SW_SHOWMINNOACTIVE	Displays **CWnd** as minimized (iconic) window. The window that is currently active remains active.
SW_SHOWNA	Displays **CWnd** in its current state. The window that is currently active remains active.

Value	Meaning
SW_SHOWNOACTIVATE	Displays **CWnd** in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays **CWnd**. If **CWnd** is minimized or maximized, Windows restores it to its original size and position.

Remarks Displays or removes the **CWnd**, as specified by *nCmdShow*.

ShowWindow must be called only once per program 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 Specifies the previous state of the **CWnd**. It is **TRUE** if the **CWnd** was previously visible; **FALSE** if the **CWnd** was previously hidden.

See Also **::ShowWindow**

CWnd::UpdateWindow

Syntax **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::ValidateRect

Syntax

void ValidateRect(LPRECT *lpRect* **);**

Parameters

lpRect
Points to a **CRect** or **RECT** structure that contains the rectangle (in client coordinates) to be removed from the update region.

Remarks

Validates the client area within the given rectangle by removing the rectangle from the update region of the given window. If *lpRect* is **NULL**, the entire window is validated.

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 sent.

Windows continues to send **WM_PAINT** messages until the current update region is validated.

See Also

CWnd::BeginPaint, ::ValidateRect, CWnd::ValidateRgn

CWnd::ValidateRgn

Syntax

void ValidateRgn(CRgn* *pRgn* **);**

Parameters

pRgn
Identifies a region that defines the area to be removed from the update region.

Remarks

Validates the client area within the given region by removing the region from the current update region of the given window. If *pRgn* is **NULL**, the entire window is validated.

The given region must have been created previously by a region function. The region coordinates are assumed to be client coordinates.

See Also

::ValidateRgn

CWnd::WindowFromPoint

Syntax

static CWnd* WindowFromPoint(POINT *point* **);**

Parameters

point
Specifies a **CPoint** or **POINT** data structure that defines the point to be checked.

Remarks

Identifies the window that contains the given point; *point* must specify the screen coordinates of a point on the screen.

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.

See Also

::WindowFromPoint, **CWnd::ChildWindowFromPoint**

CWnd::WindowProc

Syntax

protected: virtual LONG WindowProc(UINT *message*, **UINT** *wParam*, **LONG** *lParam* **);**

Parameters

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**

CWnd::wndBottom

Remarks

This is a special static **CWnd** that has an **HWND** of 1. It is only used with the *pWndInsertAfter* parameter of the **SetWindowPos** member function to indicate that the **CWnd** being operated on should be moved to the bottom of the window list. **wndBottom** is a public variable of type **static const CWnd**.

See Also

CWnd::SetWindowPos

CWnd::wndTop

Remarks

This is a special static **CWnd** that has an **HWND** of 0. It is only used with the *pWndInsertAfter* parameter of the **SetWindowPos** member function to indicate that the **CWnd** being operated on should be moved to the top of the window list. **wndTop** is a public variable of type **static const CWnd**.

See Also

CWnd::SetWindowPos

class CWordArray : public CObject

CObject
CWordArray

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 the 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>

Public Members

Construction/Destruction

CWordArray	Constructs an empty array for words.
~CWordArray	Destroys a **CWordArray** object.

Bounds

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

FreeExtra	Frees all unused memory above the current upper bound.
RemoveAll	Removes all the elements from this array.

Element Access

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

SetAtGrow	Sets the value for a given index, growing the array if necessary.
Add	Adds an element to the end of the array; grows the array if necessary.

Insertion/Removal

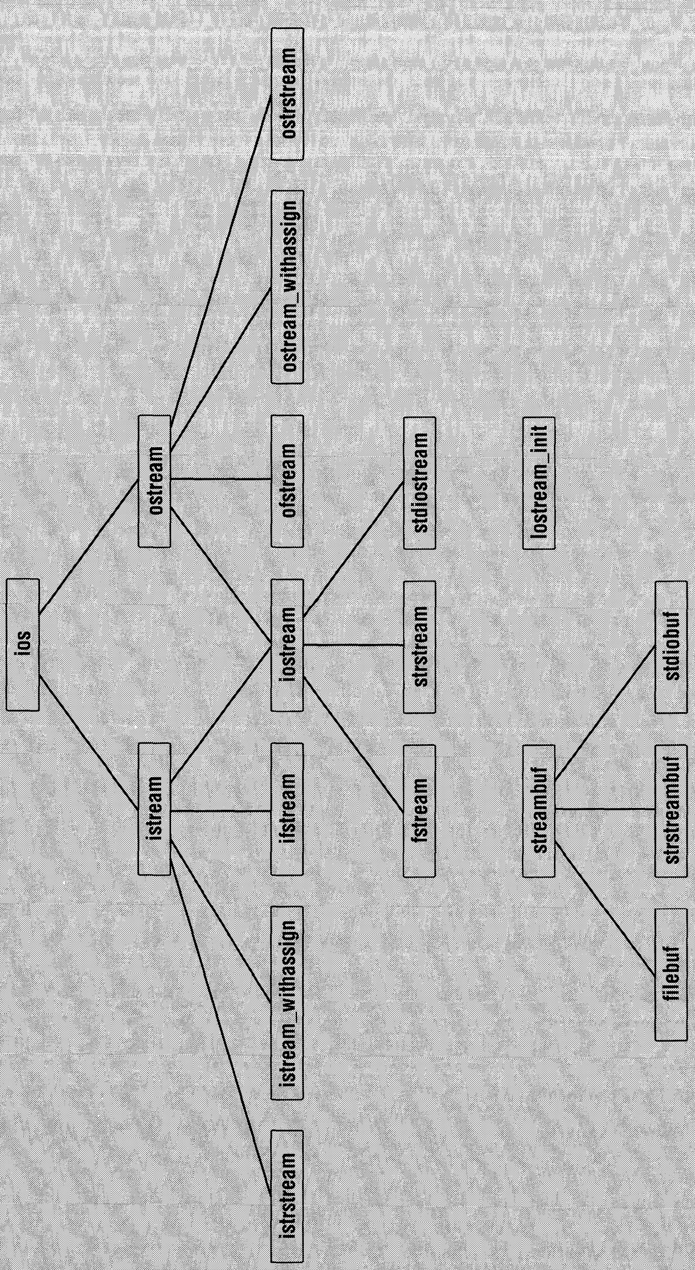
InsertAt	Inserts an element (or all the elements in another array) at a specified index.
RemoveAt	Removes an element at a specific index.

Operators

operator [ ]	Sets or gets the element at the specified index.

The Microsoft iostream Class Library Reference

Part 3


ios
istream
ostream
iostream
istrstream
istream_withassign
ifstream
ofstream
ostream_withassign
ostrstream
fstream
strstream
stdiostream
Iostream_init
streambuf
filebuf
strstreambuf
stdiobuf

iostream Class List

Abstract Stream Base Class

ios	Stream base class.

Input Stream Classes

istream	General-purpose input stream class and base class for other input streams.
ifstream	Input file stream class.
istream_withassign	Input stream class for **cin**.
istrstream	Input string stream class.

Output Stream Classes

ostream	General-purpose output stream class and base class for other output streams.
ofstream	Output file stream class.
ostream_withassign	Output stream class for **cout**, **cerr**, and **clog**.
ostrstream	Output string stream class.

Input/Output Stream Classes

iostream	General-purpose input/output stream class and base class for other input/output streams.
fstream	Input/output file stream class.
strstream	Input/output string stream class.
stdiostream	Input/output class for standard I/O files.

Stream Buffer Classes

streambuf	Abstract stream buffer base class.
filebuf	Stream buffer class for disk files.
strstreambuf	Stream buffer class for strings.
stdiobuf	Stream buffer class for standard I/O files.

Predefined Stream Initializer Class

Iostream_init	Predefined stream initializer class.

class filebuf : public streambuf

The **filebuf** class is a derived class of **streambuf** that is specialized for buffered disk file I/O. The buffering is managed entirely within the Microsoft iostream Class Library. **filebuf** member functions call the run-time low-level I/O routines (the functions declared in IO.H) such as **_sopen**, **_read**, and **_write**.

The file stream classes, **ofstream**, **ifstream**, and **fstream**, use **filebuf** member functions to fetch and store characters. Some of these member functions are virtual functions defined for the **streambuf** class.

The reserve area, put area, and get area were introduced in the **streambuf** class description. The put area and the get area are always the same for **filebuf** objects. Also, the get pointer and put pointers are tied; when one moves so does the other.

#include <fstream.h>

See Also **ifstream**, **ofstream**, **streambuf**, **strstreambuf**, **stdiobuf**

Public Members

Construction/Destruction

filebuf	Constructs a **filebuf** object.
~filebuf	Destroys a **filebuf** object.

Operations

open	Opens a file and attaches it to the **filebuf** object.
close	Flushes any waiting output and closes the attached file.
setmode	Sets the file's mode to binary or text.
attach	Attaches the **filebuf** object to an open file.

Status/Information

fd	Returns the stream's file descriptor.
is_open	Tests whether the file is open.

Member Functions

filebuf::attach

Syntax

filebuf* attach(filedesc *fd* **);**

Parameters

fd
A file descriptor as returned by a call to the run-time function **_open** or **_sopen**. **filedesc** is a **typedef** equivalent to **int**.

Remarks

Attaches this **filebuf** object to the open file specified by *fd*.

Return Value

NULL when the stream is already attached to a file; otherwise it returns its own address.

filebuf::close

Syntax

filebuf* close();

Remarks

This function flushes any waiting output, closes the file, and disconnects the file from the **filebuf** object. If an error occurs, the function returns **NULL** and leaves the **filebuf** object in a closed state. If there is no error, the function returns the address of the **filebuf** object and clears its error state.

See Also

filebuf::open

filebuf::fd

Syntax

filedesc fd() const;

Remarks

Returns the file descriptor associated with the **filebuf** object; **filedesc** is a **typedef** equivalent to **int**. Its value is supplied by the underlying file system. The function returns **EOF** if the object is not attached to a file.

See Also

filebuf::attach

filebuf::filebuf

Syntax

filebuf();

filebuf(filedesc *fd* **);**

filebuf(filedesc *fd*, **char*** *pr*, **int** *nLength* **);**

Parameters

fd
A file descriptor as returned by a call to the run-time function **_sopen**. **filedesc** is a **typedef** equivalent to **int**.

pr
Pointer to a previously allocated reserve area of length *nLength.*

nLength
The length (in bytes) of the reserve area.

Remarks

The three **filebuf** constructors are described as follows:

Constructor	Description
filebuf()	Constructs a **filebuf** object without attaching it to a file.
filebuf(filedesc)	Constructs a **filebuf** object and attaches it to an open file.
filebuf(filedesc, char*, int)	Constructs a **filebuf** object, attaches it to an open file, and initializes it to use a specified reserve area.

filebuf::~filebuf

Syntax

~filebuf();

Remarks

Closes the attached file only if that file was opened by the **open** member function.

filebuf::is_open

Syntax

int is_open() const;

Remarks

Returns a nonzero value if this **filebuf** object is attached to an open disk file identified by a file descriptor; otherwise 0.

See Also

filebuf::open

filebuf::open

Syntax

filebuf* open(const char* *szName*, **int** *nMode*, **int** *nProt* = **filebuf::openprot);**

Parameters

szName
The name of the file to be opened during construction.

nMode
An integer containing mode bits defined as **ios** enumerators that can be combined with the OR (|) operator. See the **ofstream** constructor for a list of the enumerators.

nProt
The file protection specification; defaults to the static integer **filebuf::openprot** that is equivalent to **filebuf::sh_compat**. The possible *nProt* values are as follows:

Value	Meaning
filebuf::sh_compat	Compatibility share mode.
filebuf::sh_none	Exclusive mode—no sharing.

Value	Meaning
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The **filebuf::sh_read** and **filebuf::sh_write** modes can be combined with the logical OR (|) operator.

Remarks

Opens a disk file and attaches it with this **filebuf** object. If the file is already open, or if there is an error while opening the file, the function returns **NULL**; otherwise it returns the **filebuf** address.

See Also

filebuf::is_open, **filebuf::close**, **filebuf::filebuf**

filebuf::setmode

Syntax

int setmode(int *nMode* **= filebuf::text);**

Parameters

nMode

An integer that must be one of the static **filebuf** constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return–linefeed pairs).
filebuf::binary	Binary mode (no translation).

Remarks

This function sets the binary/text mode of the stream's **filebuf** object.

Return Value

The previous mode if there is no error; otherwise 0.

See Also

ios binary manipulator, **ios text** manipulator

class fstream : public iostream

Description

The **fstream** class is an **iostream** derivative specialized for combined disk file input and output. Its constructors automatically create and attach a **filebuf** buffer object.

The **filebuf** class documentation describes the get and put areas and their associated pointers. Although the **filebuf** object's get and put pointers are theoretically independent, the get area and the put area cannot both be active at the same time. Therefore, when the stream's mode changes from input to output, the get area is emptied and the put area is reinitialized. When the mode changes from output to input, the put area is flushed and the get area is reinitialized.

#include <fstream.h>

See Also

ifstream, **ofstream**, **strstream**, **stdiostream**, **filebuf**

Public Members

Construction/Destruction

fstream	Constructs an **fstream** object.
~fstream	Destroys an **fstream** object.

Operations

open	Opens a file and attaches it to the **filebuf** object and thus to the stream.
close	Flushes any waiting output and closes the stream's file.
setbuf	Attaches the specified reserve area to the stream's **filebuf** object.
setmode	Sets the stream's mode to binary or text.
attach	Attaches the stream (through the **filebuf** object) to an open file.

Status/Information

rdbuf	Gets the stream's **filebuf** object.
fd	Returns the file descriptor associated with the stream.
is_open	Tests whether the stream's file is open.

Member Functions

fstream::attach

Syntax

void attach(filedesc *fd* **);**

Parameters

fd

A file descriptor as returned by a call to the run-time function **_open** or **_sopen**; **filedesc** is a **typedef** equivalent to **int**.

Remarks

Attaches this stream to the open file specified by *fd*. The function fails when the stream is already attached to a file. In that case, the function sets **ios::failbit** in the stream's error state.

See Also

filebuf::attach, **fstream::fd**

fstream::close

Syntax

void close();

Remarks

Calls the **close** member function for the associated **filebuf** object. This function, in turn, flushes any waiting output, closes the file, and disconnects the file from the **filebuf** object. The **filebuf** object is not destroyed.

The stream's error state is cleared unless the call to **filebuf::close** fails.

See Also

filebuf::close, **fstream::open**, **fstream::is_open**

fstream::fd

Syntax

filedesc fd() const;

Remarks

Returns the file descriptor associated with the stream. **filedesc** is a **typedef** equivalent to **int**. Its value is supplied by the underlying file system.

See Also

filebuf::fd, fstream::attach

fstream::fstream

Syntax

fstream();

fstream(const char* *szName*, **int** *nMode*, **int** *nProt* = **filebuf::openprot);**

fstream(filedesc *fd* **);**

fstream(filedesc *fd*, **char*** *pch*, **int** *nLength* **);**

Parameters

szName
The name of the file to be opened during construction.

nMode
An integer that contains mode bits defined as **ios** enumerators that can be combined with the bitwise-OR (|) operator:

Value	Meaning
ios::app	The function performs a seek to the end of file. When new bytes are written to the file, they are always appended to the end, even if the position is moved with the **ostream::seekp** function.
ios::ate	The function performs a seek to the end of file. When the first new byte is written to the file, it is appended to the end, but when subsequent bytes are written, they are written to the current position.

Value	Meaning
ios::in	The file is opened for input. The original file (if it exists), will not be truncated.
ios::out	The file is opened for output.
ios::trunc	If the file already exists, its contents are discarded. This mode is implied if **ios::out** is specified and **ios::ate**, **ios::app**, and **ios:in** are not specified.
ios::nocreate	If the file does not already exist, the function fails.
ios::noreplace	If the file already exists, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

Note that there is no **ios::in** or **ios::out** default mode for **fstream** objects. You must specify both modes if your **fstream** object must both read and write files.

nProt

The file protection specification; defaults to the static integer **filebuf::openprot** that is equivalent to **filebuf::sh_compat**. The possible *nProt* values are as follows:

Value	Meaning
filebuf::sh_compat	Compatibility share mode.
filebuf::sh_none	Exclusive mode—no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The **filebuf::sh_read** and **filebuf::sh_write** modes can be combined with the logical OR (|) operator.

fd

A file descriptor as returned by a call to the run-time function **_open** or **_sopen**. **filedesc** is a **typedef** equivalent to **int**.

pch

Pointer to a previously allocated reserve area of length *nLength*. A **NULL** value (or *nLength* = 0) indicates that the stream will be unbuffered.

nLength

The length (in bytes) of the reserve area (0 = unbuffered).

Remarks

The four **fstream** constructors are described as follows:

Constructor	Description
fstream()	Constructs an **fstream** object without opening a file.
fstream(const char*, int, int)	Contructs an **fstream** object, opening the specified file.
fstream(filedesc)	Constructs an **fstream** object that is attached to an open file.
fstream(filedesc, char*, int)	Constructs an **fstream** object that is associated with a **filebuf** object. The **filebuf** object is attached to an open file and to a specified reserve area.

All **fstream** constructors construct a **filebuf** object. The first three use an internally allocated reserve area, but the fourth uses a user-allocated area. The user-allocated area is not automatically released during destruction.

fstream::~fstream

Syntax

~fstream();

Remarks

Flushes the buffer, then destroys an **fstream** object, along with its associated **filebuf** object. The file is closed only if it was opened by the constructor or by the **open** member function.

The **filebuf** destructor releases the reserve buffer only if it was internally allocated.

fstream::is_open

Syntax

int is_open() const;

Remarks

Returns a nonzero value if this stream is attached to an open disk file identified by a file descriptor; otherwise 0.

See Also

filebuf::is_open, **fstream::open**, **fstream::close**

fstream::open

Syntax

void open(const char* *szName*, **int** *nMode*, **int** *nProt* = **filebuf::openprot);**

Parameters

szName
The name of the file to be opened during construction.

nMode
An integer containing mode bits defined as **ios** enumerators that can be combined with the OR (|) operator. See the **fstream** constructor for a list of the enumerators. The **ios::out** mode is implied.

nProt
The file protection specification; defaults to the static integer **filebuf::openprot**. See the **fstream** constructor for a list of the other allowed values.

Remarks

Opens a disk file and attaches it to the stream's **filebuf** object. If the **filebuf** object is already attached to an open file, or if a **filebuf** call fails, the **ios::failbit** is set. If the file is not found, then the **ios::failbit** is set only if the **ios::nocreate** mode was used.

See Also

filebuf::open, **fstream::fstream**, **fstream::close**, **fstream::is_open**

fstream::rdbuf

Syntax

filebuf* rdbuf() const;

Remarks

Returns a pointer to the **filebuf** buffer object that is associated with this stream. (Note that this is not the character buffer; the **filebuf** object contains a pointer to the character area.)

fstream::setbuf

Syntax

streambuf* setbuf(char* *pch*, **int** *nLength* **);**

Parameters

pch
A pointer to a previously allocated reserve area of length *nLength.* A **NULL** value indicates an unbuffered stream.

nLength
The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Remarks

Attaches the specified reserve area to the stream's **filebuf** object. If the file is open and a buffer has already been allocated, the function returns **NULL**; otherwise it returns a pointer to the **filebuf** cast as a **streambuf**. The reserve area will not be released by the destructor.

fstream::setmode

Syntax

int setmode(int *nMode* **= filebuf::text);**

Parameters

nMode

An integer that must be one of the static **filebuf** constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return–linefeed pairs).
filebuf::binary	Binary mode (no translation).

Remarks

This function sets the binary/text mode of the stream's **filebuf** object. It may be called only after the file is opened.

Return Value

The previous mode; –1 if the parameter is invalid, the file is not open, or the mode cannnot be changed.

See Also

ios binary manipulator, **ios text** manipulator

class ifstream : public istream

Description

The **ifstream** class is an **istream** derivative specialized for disk file input. Its constructors automatically create and attach a **filebuf** buffer object.

The **filebuf** class documentation describes the get and put areas and their associated pointers. Only the get area and the get pointer are active for the **ifstream** class.

#include <fstream.h>

See Also

filebuf, **streambuf**, **ofstream**, **fstream**

Public Members

Construction/Destruction

ifstream	Constructs an **ifstream** object.
~ifstream	Destroys an **ifstream** object.

Operations

open	Opens a file and attaches it to the **filebuf** object and thus to the stream.
close	Closes the stream's file.
setbuf	Associates the specified reserve area to the stream's **filebuf** object.
setmode	Sets the stream's mode to binary or text.
attach	Attaches the stream (through the **filebuf** object) to an open file.

Status/Information

rdbuf	Gets the stream's **filebuf** object.
fd	Returns the file descriptor associated with the stream.
is_open	Tests whether the stream's file is open.

Member Functions

ifstream::attach

Syntax

void attach(filedesc *fd* **);**

Parameters

fd

A file descriptor as returned by a call to the run-time function **_open** or **_sopen**; **filedesc** is a **typedef** equivalent to **int**.

Remarks

Attaches this stream to the open file specified by *fd*. The function fails when the stream is already attached to a file. In that case, the function sets **ios::failbit** in the stream's error state.

See Also

filebuf::attach, **ifstream::fd**

ifstream::close

Syntax

void close();

Remarks

Calls the **close** member function for the associated **filebuf** object. This function, in turn, closes the file and disconnects the file from the **filebuf** object. The **filebuf** object is not destroyed.

The stream's error state is cleared unless the call to **filebuf::close** fails.

See Also

filebuf::close, **ifstream::open**, **ifstream::is_open**

ifstream::fd

Syntax

filedesc fd() const;

Remarks

Returns the file descriptor associated with the stream; **filedesc** is a **typedef** equivalent to **int**. Its value is supplied by the underlying file system.

See Also

filebuf::fd, ifstream::attach

ifstream::ifstream

Syntax

ifstream();

ifstream(const char* *szName*, **int** *nMode* = **ios::in,**
int *nProt* = **filebuf::openprot);**

ifstream(filedesc *fd* **);**

ifstream(filedesc *fd*, **char*** *pch*, **int** *nLength* **);**

Parameters

szName
The name of the file to be opened during construction.

nMode
An integer that contains mode bits defined as **ios** enumerators that can be combined with the bitwise-OR (|) operator:

Value	Meaning
ios::in	The file is opened for input (default).
ios::nocreate	If the file does not already exist, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

Note that the **ios::nocreate** flag is necessary if you intend to test for the file's existence (the usual case).

nProt
The file protection specification; defaults to the static integer **filebuf::openprot** that is equivalent to **filebuf::sh_compat**. The possible *nProt* values are as follows:

Value	Meaning
filebuf::sh_compat	Compatibility share mode.
filebuf::sh_none	Exclusive mode—no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The **filebuf::sh_read** and **filebuf::sh_write** modes can be combined with the logical OR (|) operator.

fd

A file descriptor as returned by a call to the run-time function **_open** or **_sopen**; **filedesc** is a **typedef** equivalent to **int**.

pch

Pointer to a previously allocated reserve area of length *nLength.* A **NULL** value (or *nLength* = 0) indicates that the stream will be unbuffered.

nLength

The length (in bytes) of the reserve area (0 = unbuffered).

Remarks

The four **ifstream** constructors are described as follows:

Constructor	Description
ifstream()	Constructs an **ifstream** object without opening a file.
ifstream(const char*, int, int)	Contructs an **ifstream** object, opening the specified file.
ifstream(filedesc)	Constructs an **ifstream** object that is attached to an open file.
ifstream(filedesc, char*, int)	Constructs an **ifstream** object that is associated with a **filebuf** object. The **filebuf** object is attached to an open file and to a specified reserve area.

All **ifstream** constructors construct a **filebuf** object. The first three use an internally allocated reserve area, but the fourth uses a user-allocated area.

ifstream::~ifstream

Syntax

~ifstream();

Remarks

Destroys an **ifstream** object along with its associated **filebuf** object. The file is closed only if was opened by the constructor or by the **open** member function.

The **filebuf** destructor releases the reserve buffer only if it was internally allocated.

ifstream::is_open

Syntax

int is_open() const;

Remarks

Returns a nonzero value if this stream is attached to an open disk file identified by a file descriptor; otherwise 0.

See Also

filebuf::is_open, **ifstream::open**, **ifstream::close**

ifstream::open

Syntax

void open(const char* *szName*, **int** *nMode* = **ios::in,**
  int *nProt* = **filebuf::openprot);**

Parameters

szName
  The name of the file to be opened during construction.

nMode
  An integer containing bits defined as **ios** enumerators that can be combined with the OR (|) operator. See the **ifstream** constructor for a list of the enumerators. The **ios::in** mode is implied.

nProt
  The file protection specification; defaults to the static integer **filebuf::openprot**. See the **ifstream** constructor for a list of the other allowed values.

Remarks

Opens a disk file and attaches it to the stream's **filebuf** object. If the **filebuf** object is already attached to an open file, or if a **filebuf** call fails, the **ios::failbit** is set. If the file is not found, then the **ios::failbit** is set only if the **ios::nocreate** mode was used.

See Also

filebuf::open, **ifstream::ifstream**, **ifstream::close**, **ifstream::is_open**

ifstream::rdbuf

Syntax

filebuf* rdbuf() const;

Remarks

Returns a pointer to the **filebuf** buffer object that is associated with this stream. (Note that this is not the character buffer; the **filebuf** object contains a pointer to the character area.)

ifstream::setbuf

Syntax

streambuf* setbuf(char* *pch*, **int** *nLength* **);**

Parameters

pch
A pointer to a previously allocated reserve area of length *nLength.* A **NULL** value indicates an unbuffered stream.

nLength
The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Remarks

Attaches the specified reserve area to the stream's **filebuf** object. If the file is open and a buffer has already been allocated, the function returns **NULL**; otherwise it returns a pointer to the **filebuf**, which is cast as a **streambuf**. The reserve area will not be released by the destructor.

ifstream::setmode

Syntax

int setmode(int *nMode* **= filebuf::text);**

Parameters

nMode

An integer that must be one of the static **filebuf** constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return–linefeed pairs).
filebuf::binary	Binary mode (no translation).

Remarks

This function sets the binary/text mode of the stream's **filebuf** object. It may be called only after the file is opened.

Return Value

The previous mode; –1 if the parameter is invalid, the file is not open, or the mode cannnot be changed.

See Also

ios binary manipulator, **ios text** manipulator

class ios

As the iostream class hierarchy diagram shows, the **ios** class is the base class for all the input/output stream classes. While **ios** is not technically an abstract base class, you will not usually construct **ios** objects, nor will you derive classes directly from **ios**. Instead, you will use the derived classes **istream** and **ostream** or other derived classes.

Even though you will not use **ios** directly, you will be using many of the inherited member functions and data members described here. Remember that these inherited member function descriptions are not duplicated for derived classes.

#include <iostream.h>

See Also **istream**, **ostream**

Public Members

Data Members (static)

basefield	Mask for obtaining the conversion base flags (**dec**, **oct**, or **hex**).
adjustfield	Mask for obtaining the field padding flags (**left**, **right**, or **internal**).
floatfield	Mask for obtaining the numeric format (**scientific** or **fixed**).

Construction/Destruction

ios	Constructor for use in derived classes.
~ios	Virtual destructor.

Flag and Format Access Functions

flags	Sets or reads the stream's format flags.
setf	Manipulates the stream's format flags.
unsetf	Clears the stream's format flags.
fill	Sets or reads the stream's fill character.
precision	Sets or reads the stream's floating-point format display precision.
width	Sets or reads the stream's output field width.

Status-Testing Functions

good	Indicates good stream status.
bad	Indicates a serious I/O error.
eof	Indicates end of file.
fail	Indicates a serious I/O error or a possibly recoverable I/O formatting error.
rdstate	Returns the stream's error flags.
clear	Sets or clears the stream's error flags.

User-Defined Format Flags

bitalloc	Provides a mask for an unused format bit in the stream's private flags variable (static function).
xalloc	Provides an index to an unused word in an array reserved for special-purpose stream state variables (static function).
iword	Converts the index provided by **xalloc** to a reference (valid only until the next **xalloc**).
pword	Converts the index provided by **xalloc** to a pointer (valid only until the next **xalloc**).

Other Functions

delbuf	Controls the connection of **streambuf** deletion with **ios** destruction.
rdbuf	Gets the stream's **streambuf** object.
sync_with_stdio	Synchronizes the predefined objects **cin**, **cout**, **cerr**, and **clog** with the standard I/O system.
tie	Ties a specified **ostream** to this stream.

Operators

operator void*()	Converts a stream to a pointer that can be used only for error checking.
operator !()	Returns a nonzero value if a stream I/O error has occurred.

Protected Members

init	Associates a **streambuf** object with this stream.

Manipulators

ios Manipulators

dec	Causes the interpretation of subsequent fields in decimal format (the default mode).
hex	Causes the interpretation of subsequent fields in hexadecimal format.
oct	Causes the interpretation of subsequent fields in octal format.
binary	Sets the stream's mode to binary (stream must have an associated **filebuf** buffer).
text	Sets the stream's mode to text—the default mode (stream must have an associated **filebuf** buffer).

Parameterized Manipulators

(**#include <iomanip.h>** required)

setiosflags	Sets the stream's format flags.
resetiosflags	Resets the stream's format flags.
setfill	Sets the stream's fill character.
setprecision	Sets the stream's floating-point display precision.
setw	Sets the stream's field width (for the next field only).

Member Functions

ios::bad

Syntax

int bad() const;

Remarks

Returns a nonzero value to indicate a serious I/O error. This condition corresponds to the **badbit** error state being set. Do not continue I/O operations on the stream in this situation.

See Also

ios::good, **ios::fail**, **ios::rdstate**

ios::bitalloc

Syntax

static long bitalloc();

Remarks

The **ios** class currently defines 15 format flag bits accessible through **flags** and other member functions. These bits reside in a 32-bit private **ios** data member and are accessed through enumerators such as **ios::left** and **ios::hex**.

The **bitalloc** member function provides a mask for a previously unused bit in the data member. Once you obtain the mask, you can use it to set or test the corresponding custom flag bit in conjunction with the **ios** member functions and manipulators listed below under "See Also."

See Also

ios::flags, **ios::setf**, **ios::unsetf**, **ios setiosflags** manipulator, **ios resetiosflags** manipulator

ios::clear

Syntax

void clear(int *nState* **= 0);**

Parameters

nState

If 0, all error bits are cleared; otherwise bits are set according to the following masks (**ios** enumerators) that can be combined using the bitwise-OR (|) operator:

Value	Meaning
ios::goodbit	No error condition (no bits set)
ios::eofbit	End of file reached
ios::failbit	A possibly recoverable formatting or conversion error
ios::badbit	A severe I/O error

Remarks

Sets or clears the error-state flags. The **rdstate** function can be used to read the current error state.

See Also

ios::rdstate, **ios::good**, **ios::bad**, **ios::eof**

ios::delbuf

Syntax

void delbuf(int *nDelFlag* **);**

int delbuf() const;

Parameters

nDelFlag

A nonzero value indicates that **~ios** should delete the stream's attached **streambuf** object. A 0 value prevents deletion.

Remarks

The first overloaded **delbuf** function assigns a value to the stream's buffer-deletion flag.

The second function returns the current value of the flag.

This function is public only because it is accessed by the **Iostream_init** class. Treat it as protected.

See Also **ios::rdbuf**, **ios::~ios**

ios::eof

Syntax **int eof() const;**

Remarks Returns a nonzero value if end of file has been reached. This condition corresponds to the **eofbit** error flag being set.

ios::fail

Syntax **int fail() const;**

Remarks Returns a nonzero value if any I/O error (not end of file) has occurred. This condition corresponds to either the **badbit** or **failbit** error flag being set. If a call to **bad** returns 0, you can assume that the error condition is nonfatal and that you can probably continue processing after you clear the flags.

See Also **ios::bad**, **ios::clear**

ios::fill

Syntax **char fill(char** *cFill* **);**

char fill() const;

Parameters *cFill*

The new fill character to be used as padding between fields.

Remarks

The first overloaded function sets the stream's internal fill character variable to *cFill* and returns the previous value. The default fill character is a space.

The second **fill** function returns the stream's fill character.

See Also

ios setfill manipulator

ios::flags

Syntax

long flags(long *lFlags* **);**

long flags() const;

Parameters

lFlags

The new format flag values for the stream. The values are specified by the following bit masks (**ios** enumerators) that can be combined using the bitwise-OR (|) operator:

Value	Meaning
ios::skipws	Skip white space on input.
ios::left	Left-align values; pad on the right with the fill character.
ios::right	Right-align values; pad on the left with the fill character (default alignment).
ios::internal	Add fill characters after any leading sign or base indication, but before the value.
ios::dec	Format numeric values as base 10 (decimal) (default radix).
ios::oct	Format numeric values as base 8 (octal).
ios::hex	Format numeric values as base 16 (hexadecimal).
ios::showbase	Display numeric constants in a format that can be read by the C++ compiler.

Value	Meaning
ios::showpoint	Show decimal point and trailing zeros for floating-point values.
ios::uppercase	Display uppercase A through F for hexadecimal values and E for scientific values.
ios::showpos	Show plus signs (+) for positive values.
ios::scientific	Display floating-point numbers in scientific format.
ios::fixed	Display floating-point numbers in fixed format.
ios::unitbuf	Cause **ostream::osfx** to flush the stream after each insertion. By default, **cerr** is unit buffered.
ios::stdio	Cause **ostream::osfx** to flush **stdout** and **stderr** after each insertion.

Remarks

The first overloaded **flags** function sets the stream's internal flags variable to *lFlags* and returns the previous value.

The second function returns the stream's current flags.

See Also

ios::setf, **ios::unsetf**, **ios setiosflags** manipulator, **ios resetiosflags** manipulator, **ios::adjustfield**, **ios::basefield**, **ios::floatfield**

ios::good

Syntax

int good() const;

Remarks

Returns a nonzero value if all error bits are clear. Note that the **good** member function is not simply the inverse of the **bad** function.

See Also

ios::bad, **ios::fail**, **ios::rdstate**

ios::ios

Syntax

ios(streambuf* *psb* **);**

Parameters

psb
A pointer to an existing **streambuf** object.

Remarks

Constructor for **ios**. You will seldom need to invoke this constructor except in derived classes. Generally, you will be deriving classes not from **ios** but from **istream**, **ostream**, and **iostream**.

ios::~ios

Syntax

virtual ~ios();

Remarks

Virtual destructor for **ios**.

ios::iword

Syntax

long& iword(int *nIndex* **) const;**

Parameters

nIndex
An index into a table of words that are associated with the **ios** object.

Remarks

The **xalloc** member function provides the index into the table of special-purpose words. The **iword** function converts that index to a reference to a 32-bit word.

See Also

ios::xalloc, **ios::pword**

ios::precision

Syntax

int precision(int *np* **);**

int precision() const;

Parameters

np
An integer that indicates the number of significant digits or significant decimal digits to be used for floating-point display.

Remarks

The first overloaded **precision** function sets the stream's internal floating-point precision variable to *np* and returns the previous value. The default precision is six digits. If the display format is scientific or fixed, then the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), then the precision indicates the total number of significant digits.

The second function returns the stream's current precision value.

See Also

ios setprecision manipulator

ios::pword

Syntax

void*& pword(int *nIndex* **) const;**

Parameters

nIndex
An index into a table of words that are associated with the **ios** object.

Remarks

The **xalloc** member function provides the index into the table of special-purpose words. The **pword** function converts that index to a reference to a pointer to a 32-bit word.

See Also

ios::xalloc, ios::iword

ios::rdbuf

Syntax

streambuf* rdbuf() const;

Remarks

Returns a pointer to the **streambuf** object that is associated with this stream. The **rdbuf** function is useful when you need to call **streambuf** member functions.

ios::rdstate

Syntax

int rdstate() const;

Remarks

Returns the current error state as specified by the following masks (**ios** enumerators):

Value	Meaning
ios::goodbit	No error condition
ios::eofbit	End of file reached
ios::failbit	A possibly recoverable formatting or conversion error
ios::badbit	A severe I/O error or unknown state

The returned value can be tested against a mask with the AND (**&**) operator.

See Also

ios::clear

ios::setf

Syntax

long setf(long *lFlags* **);**

long setf(long *lFlags***, long** *lMask* **);**

Parameters

lFlags
Format flag bit values. See the **flags** member function for a list of format flags. These flags can be combined by using the bitwise-OR (|) operator.

lMask
Format flag bit mask.

Remarks

The first overloaded **setf** function turns on only those format bits that are specified by 1s in *lFlags*. It returns a **long** that contains the previous value of all the flags.

The second function alters those format bits specified by 1s in *lMask*. The new values of those format bits are determined by the corresponding bits in *lFlags*. It returns a **long** that contains the previous value of all the flags.

See Also

ios::flags, **ios::unsetf**, **ios setiosflags** manipulator

ios::sync_with_stdio

Syntax

static void sync_with_stdio();

Remarks

Synchronizes the C++ streams with the standard I/O system. The first time this function is called, it resets the predefined streams (**cin**, **cout**, **cerr**, **clog**) to use a **stdiobuf** object rather than a **filebuf** object. After that, I/O using these streams can be freely mixed with I/O using **stdin**, **stdout**, and **stderr**. Some performance decrease will result because there is buffering both in the stream class and in the standard I/O file system.

After the call to **sync_with_stdio**, the **ios::stdio** bit is set for all affected predefined stream objects, and **cout** is set to unit buffered mode.

ios::tie

Syntax

ostream* tie(ostream* *pos* **);**

ostream* tie() const;

Parameters

pos
A pointer to an **ostream** object.

Remarks

The first overloaded **tie** function ties this stream to the specified **ostream** and returns the value of the previous tie pointer (**NULL** if this stream was not previously tied). A stream tie enables automatic flushing of the **ostream** in response to (1) a need for more characters or (2) the presence of characters to be consumed.

By default, **cin** is initially tied to **cout** so that attempts to get more characters from standard input may result in flushing standard output. In addition, **cerr** and **clog** are tied to **cout** by default.

The second function returns the value of the previous tie pointer (**NULL** if this stream was not previously tied).

ios::unsetf

Syntax

long unsetf(long *lFlags* **);**

Parameters

lFlags
Format flag bit values. See the **flags** member function for a list of format flags.

Remarks

Clears the format flags specified by 1s in *lFlags*. It returns a **long** that contains the previous value of all the flags.

See Also

ios::flags, **ios::setf**, **ios resetiosflags** manipulator

ios::width

Syntax

int width(int *nw* **);**

int width() const;

Parameters

nw
The minimum field width in characters.

Remarks

The first overloaded **width** function sets the stream's internal field width variable to *nw*. When the width is 0 (the default), inserters insert only as many characters as necessary to represent the inserted value. When the width is not 0, the inserters pad the field with the stream's fill character, up to *nw*. If the unpadded representation of the field is larger than *nw*, the field is not truncated. Thus *nw* is a minimum field width.

The internal width value is reset to 0 after each insertion or extraction.

The second overloaded **width** function returns the current value of the stream's width variable.

See Also

ios setw manipulator

ios::xalloc

Syntax

static int xalloc();

Remarks

Provides extra **ios** object state variables without the need for class derivation. It does so by returning an index to an unused 32-bit word in an internal array. This index can be subsequently converted into a reference or pointer by using the **iword** or **pword** member functions.

Any call to **xalloc** invalidates values returned by previous calls to **iword** and **pword**.

See Also

ios::iword, ios::pword

Operators

ios::operator void* ()

Syntax

operator void* () **const;**

Remarks

An operator that converts a stream to a pointer that can be compared to 0. The conversion returns 0 if either **failbit** or **badbit** is set in the stream's error state. See **rdstate** for a description of the error state masks. A nonzero pointer is not meant to be dereferenced.

See Also

ios::good, **ios::fail**

ios::operator !()

Syntax

int operator !() **const;**

Remarks

Returns a nonzero value if either **failbit** or **badbit** are set in the stream's error state. See **rdstate** for a description of the error state masks.

See Also

ios::good, **ios::fail**

Data Members

ios::adjustfield

Syntax

static const long adjustfield;

Remarks

A mask that can be used to obtain the padding flag bits (**left, right,** or **internal**).

Example

```
extern ostream os;
if( ( os.flags() & ios::adjustfield ) == ios::left ) .....
```

See Also

ios::flags

---

ios::basefield

Syntax

static const long basefield;

Remarks

A mask that can be used to obtain the current radix flag bits (**dec, oct,** or **hex**).

Example

```
extern ostream os;
if( ( os.flags() & ios::basefield ) == ios::hex ) .....
```

See Also

ios::flags

---

ios::floatfield

Syntax

static const long floatfield;

Remarks

A mask that can be used to obtain floating-point format flag bits (**scientific** or **fixed**).

Example

```
extern ostream os;
if( ( os.flags() & ios::floatfield ) == ios::scientific ) .....
```

See Also

ios::flags

Manipulators

ios& binary

Syntax

binary

#include <fstream.h>

Remarks

Sets the stream's mode to binary. The default mode is text.

The stream must have an associated **filebuf** buffer.

See Also

ios text manipulator, **ofstream::setmode**, **ifstream::setmode**, **filebuf::setmode**

ios& dec

Syntax

dec

Remarks

Sets the format conversion base to 10 (decimal).

See Also

ios hex manipulator, **ios oct** manipulator

ios& hex

Syntax

hex

Remarks

Sets the format conversion base to 16 (hexadecimal).

See Also

ios dec manipulator, **ios oct** manipulator

ios& oct

Syntax

oct

Remarks

Sets the format conversion base to 8 (octal).

See Also

ios dec manipulator, **ios hex** manipulator

resetiosflags

Syntax

SMANIP(long) resetiosflags(long *lFlags* **);**

#include <iomanip.h>

Parameters

lFlags
Format flag bit values. See the **flags** member function for a list of format flags. These flags can be combined by using the OR (|) operator.

Remarks

This parameterized manipulator clears only the specified format flags. This setting remains in effect until the next change.

setfill

Syntax

SMANIP(int) setfill(int *nFill* **);**

#include <iomanip.h>

Parameters

nFill
The new fill character to be used as padding between fields.

Remarks

This parameterized manipulator sets the stream's fill character. The default is a space. This setting remains in effect until the next change.

setiosflags

Syntax

SMANIP(long) setiosflags(long *lFlags* **);**

#include <iomanip.h>

Parameters

lFlags
: Format flag bit values. See the **flags** member function for a list of format flags. These flags can be combined by using the OR (|) operator.

Remarks

This parameterized manipulator sets only the specified format flags. This setting remains in effect until the next change.

setprecision

Syntax

SMANIP(int) setprecision(int *np* **);**

#include <iomanip.h>

Parameters

np
: An integer that indicates the number of significant digits or significant decimal digits to be used for floating-point display.

Remarks

This parameterized manipulator sets the stream's internal floating-point precision variable to *np*. The default precision is six digits. If the display format is scientific or fixed, then the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), then the precision indicates the total number of significant digits.

This setting remains in effect until the next change.

setw

Syntax

SMANIP(int) setw(int *nw* **);**

#include <iomanip.h>

Parameters

nw
The field width in characters.

Remarks

This parameterized manipulator sets the stream's internal field width parameter. See the **width** member function for more information. This setting remains in effect only for the next insertion.

ios& text

Syntax

text

#include <fstream.h>

Remarks

Sets the stream's mode to text (the default mode).

The stream must have an associated **filebuf** buffer.

See Also

ios binary manipulator, **ofstream::setmode**, **ifstream::setmode**, **filebuf::setmode**

class iostream : public istream, public ostream

The **iostream** class provides the basic capability for sequential and random-access I/O. It inherits functionality from both the **istream** and **ostream** classes.

The **iostream** class works in conjunction with classes derived from **streambuf** (for example, **filebuf**). In fact, most of the **iostream** "personality" comes from its attached **streambuf** class. You can use **iostream** objects for sequential disk I/O if you first construct an appropriate **filebuf** object. More often, you will use objects of classes **fstream** and **strstream**.

Derivation For derivation suggestions, see the **istream** and **ostream** classes.

#include <iostream.h>

See Also **istream, ostream, fstream, strstream, stdiostream**

Public Members

iostream	Constructs an **iostream** object that is attached to an existing **streambuf** object.
~iostream	Destroys an **iostream** object.

Protected Members

iostream	Constructs an **iostream** object.

Member Functions

iostream::iostream

Syntax

Public:
iostream(streambuf* *psb* **);**

Protected:
iostream( );

Parameters

psb
A pointer to an existing **streambuf** object (or an object of a derived class).

Remarks

Constructs an object of type **iostream**.

See Also

ios::init

iostream::~iostream

Syntax

virtual ~iostream();

Remarks

Virtual destructor for the **iostream** class.

class Iostream_init

The **Iostream_init** class is a static class that initializes the predefined stream objects **cin**, **cout**, **cerr**, and **clog**. A single object of this class is constructed "invisibly" in response to the reference of any of the predefined objects. The class is documented for completeness only. You will not normally construct objects of this class.

#include <iostream.h>

Public Members

Iostream_init	A constructor that initializes **cin**, **cout**, **cerr**, and **clog**.
~Iostream_init	The destructor for the **Iostream_init** class.

Member Functions

Iostream_init::Iostream_init

Syntax **Iostream_init();**

Remarks **Iostream_init** constructor that initializes **cin**, **cout**, **cerr**, and **clog**. For internal use only.

Iostream_init::~Iostream_init

Syntax **~Iostream_init();**

Remarks **Iostream_init** destructor. For internal use only.

class istream : virtual public ios

The **istream** class provides the basic capability for sequential and random-access input. An **istream** object has a **streambuf**-derived object attached, and the two classes work together; the **istream** class does the formatting, and the **streambuf** class does the low-level buffered input.

You can use **istream** objects for sequential disk input if you first construct an appropriate **filebuf** object. More often, you will use the predefined stream object **cin** (which is actually an object of class **istream_withassign**), or you will use objects of classes **ifstream** (disk file streams) and **istrstream** (string streams).

Derivation

It is not always necessary to derive from **istream** in order to add functionality to a stream; consider deriving from **streambuf** instead, as illustrated in Chapter 19 of the *Class Libraries User's Guide*. The **ifstream** and **istrstream** classes are examples of **istream**-derived classes that construct member objects of predetermined derived **streambuf** classes.

You can add manipulators without deriving a new class.

If you add new extraction operators for a derived **istream** class, then the rules of C++ dictate that you must reimplement all the base class extraction operators. See the "Derivation" section of class **ostream** for an efficient reimplementation technique.

#include <iostream.h>

See Also

streambuf, **ifstream**, **istrstream**, **istream_withassign**

Public Members

Construction/Destruction

istream	Constructs an **istream** object attached to an existing object of a **streambuf**-derived class.
~istream	Destroys an **istream** object.

Prefix/Suffix Functions

ipfx	Check for error conditions prior to extraction operations (input prefix function).
isfx	Called after extraction operations (input suffix function).

Input Functions

get	Extracts characters from the stream up to, but not including, delimiters.
getline	Extracts characters from the stream (extracts and discards delimiters).
read	Extracts data from the stream.
ignore	Extracts and discards characters.
peek	Returns a character without extracting it from the stream.
gcount	Counts the characters extracted in the last unformatted operation.
eatwhite	Extracts leading white space.

Other Functions

putback	Puts characters back to the stream.
sync	Synchronizes the stream buffer with the external source of characters.
seekg	Changes the stream's get pointer.
tellg	Gets the value of the stream's get pointer.

Operators

operator >>	Extraction operator for various types.

Protected Members

istream	Constructs an **istream** object.

Manipulators

ws	Extracts leading white space.

Member Functions

istream::eatwhite

Syntax **void eatwhite();**

Remarks Extracts white space from the stream by advancing the get pointer past spaces and tabs.

See Also **istream ws** manipulator

istream::gcount

Syntax **int gcount() const;**

Remarks Returns the number of characters extracted by the last unformatted input function. Formatted extraction operators may call unformatted input functions and thus reset this number.

See Also **istream::get**, **istream::getline**, **istream::ignore**, **istream::read**

istream::get

Syntax

int get();

istream& get(char* *pch*, **int** *nCount*, **char** *delim* = **'\n'**);

istream& get(unsigned char* *puch*, **int** *nCount*, **char** *delim* = **'\n'**);

istream& get(signed char* *psch*, **int** *nCount*, **char** *delim* = **'\n'**);

istream& get(char& *rch*);

istream& get(unsigned char& *ruch*);

istream& get(signed char& *rsch*);

istream& get(streambuf& *rsb*, **char** *delim* = **'\n'**);

Parameters

pch, *puch*, *psch*
A pointer to a character array.

nCount
The maximum number of characters to store, including the terminating **NULL**.

delim
The delimiter character (defaults to newline).

rch, *ruch*, *rsch*
A reference to a character.

rsb
A reference to an object of a **streambuf**-derived class.

Remarks

These functions extract data from an input stream as follows:

Variation	Description
get();	Extracts a single character from the stream and returns it.
get(char*, int, char);	Extracts characters from the stream until either *delim* is found, the limit *nCount* is reached, or the end of file is reached. The characters are stored in the array followed by a null terminator.

Variation	Description
get(char&);	Extracts a single character from the stream and stores it as specified by the reference argument.
get(streambuf&, char);	Gets characters from the stream and stores them in a **streambuf** object until the delimiter is found or the end of the file is reached. The **ios::failbit** flag is set if the **streambuf** output operation fails.

In all cases, the delimiter is neither extracted from the stream nor returned by the function. The **getline** function, in contrast, extracts the delimiter but does not store it.

See Also

istream::getline, **istream::read**, **istream::ignore**, **istream::gcount**

istream::getline

Syntax

istream& getline(char* *pch*, **int** *nCount*, **char** *delim* = **'\n'**);

istream& getline(unsigned char* *puch*, **int** *nCount*, **char** *delim* = **'\n'**);

istream& getline(signed char* *psch*, **int** *nCount*, **char** *delim* = **'\n'**);

Parameters

pch, puch, psch
A pointer to a character array.

nCount
The maximum number of characters to store, including the terminating **NULL**.

delim
The delimiter character (defaults to newline).

Remarks

Extracts characters from the stream until either the delimiter *delim* is found, the limit *nCount*–1 is reached, or end of file is reached. The characters are stored in the specified array followed by a null terminator. If the delimiter is found, it is extracted but not stored.

The **get** function, in contrast, neither extracts nor stores the delimiter.

See Also

istream::get, **istream::read**

istream::ignore

Syntax

istream& ignore(int *nCount* **= 1, int** *delim* **= EOF);**

Parameters

nCount
The maximum number of characters to extract.

delim
The delimiter character (defaults to **EOF**).

Remarks

Extracts and discards up to *nCount* characters. Extraction stops if the delimiter *delim* is extracted or the end of file is reached. If *delim* = **EOF** (the default), then only the end of file condition causes termination. The delimiter character is extracted.

istream::ipfx

Syntax

int ipfx(int *need* **= 0);**

Parameters

need
Zero if called from formatted input functions; otherwise the minimum number of characters needed.

Remarks

This input prefix function is called by input functions prior to extracting data from the stream. Formatted input functions call **ipfx(0)**, while unformatted input functions usually call **ipfx(1)**.

Any **ios** object tied to this stream is flushed if *need* = 0 or if there are fewer than *need* characters in the input buffer. Also, **ipfx** extracts leading white space if **ios::skipws** is set.

Return Value

A nonzero return value if the operation was successful; 0 if the stream's error state is nonzero, in which case the function does nothing.

See Also

istream::isfx

istream::isfx

Syntax

void isfx();

Remarks

This input suffix function is called at the end of every extraction operation. In the current implementation, it does nothing, but it may be used in future versions of the class library.

istream::istream

Syntax

Public:
  istream(streambuf* *psb* **);**

Protected:
  istream( );

Parameters

psb
  A pointer to an existing object of a **streambuf**-derived class.

Remarks

Constructs an object of type **istream**.

See Also

ios::init

istream::~istream

Syntax **virtual ~istream();**

Remarks Virtual destructor for the **istream** class.

istream::peek

Syntax **int peek() const;**

Remarks Returns the next character without extracting it from the stream. Returns **EOF** if the stream is at end of file or if the **ipfx** function indicates an error.

istream::putback

Syntax **istream& putback(char** *ch* **);**

Parameters *ch*

The character to put back; must be the character previously extracted.

Remarks Puts a character back into the input stream. The **putback** function may fail and set the error state. If *ch* does not match the character that was previously extracted, then the result is undefined.

istream::read

Syntax **istream& read(char*** *pch*, **int** *nCount* **);**

istream& read(unsigned char* *puch*, **int** *nCount* **);**

istream& read(signed char* *psch*, **int** *nCount* **);**

Parameters

pch, puch, psch
A pointer to a character array.

nCount
The maximum number of characters to read.

Remarks

Extracts bytes from the stream until the limit *nCount* is reached or until the end of file is reached. The **read** function is useful for binary stream input.

See Also

istream::get, **istream::getline**, **istream::gcount**, **istream::ignore**

istream::seekg

Syntax

istream& seekg(streampos *pos* **);**

istream& seekg(streamoff *off*, **ios::seek_dir** *dir* **);**

Parameters

pos
The new position value; **streampos** is a **typedef** equivalent to **long**.

off
The new offset value; **streamoff** is a **typedef** equivalent to **long**.

dir
The seek direction. Must be one of the following enumerators:

Value	Meaning
ios::beg	Seek from the beginning of the stream.
ios::cur	Seek from the current position in the stream.
ios::end	Seek from the end of the stream.

Remarks

Changes the get pointer for the stream. Not all derived classes of **istream** need support positioning; it is most often used with file-based streams.

See Also

istream::tellg, **ostream::seekp**, **ostream::tellp**

istream::sync

Syntax **int sync();**

Remarks Synchronizes the stream's internal buffer with the external source of characters. This function calls the virtual **streambuf::sync** function so you can customize its implementation by deriving a new class from **streambuf**.

Return Value **EOF** to indicate errors.

See Also **streambuf::sync**

istream::tellg

Syntax **streampos tellg();**

Remarks Gets the value for the stream's get pointer.

Return Value A **streampos** type, corresponding to a **long**.

See Also **istream::seekg**, **ostream::tellp**, **ostream::seekp**

Operators

istream::operator >>

Syntax

istream& operator >>(char* *psz* **);**

istream& operator >>(unsigned char* *pusz* **);**

istream& operator >>(signed char* *pssz* **);**

istream& operator >>(char& *rch* **);**

istream& operator >>(unsigned char& *ruch* **);**

istream& operator >>(signed char& *rsch* **);**

istream& operator >>(short& *s* **);**

istream& operator >>(unsigned short& *us* **);**

istream& operator >>(int& *n* **);**

istream& operator >>(unsigned int& *un* **);**

istream& operator >>(long& *l* **);**

istream& operator >>(unsigned long& *ul* **);**

istream& operator >>(float& *f* **);**

istream& operator >>(double& *d* **);**

istream& operator >>(long double& *ld* **);**

istream& operator >>(streambuf* *psb* **);**

istream& operator >>(istream& (**fcn***)(istream&));**

istream& operator >>(ios& (**fcn***)(ios&));**

Remarks

These overloaded operators extract their argument from the stream. The last two variations allow the use of manipulators that are defined for both **istream** and **ios**.

Manipulators

istream& ws

Syntax ws

Remarks Extracts leading white space from the stream by calling the **eatwhite** function.

See Also **istream::eatwhite**

class istream_withassign : public istream

The **istream_withassign** class is a variant of **istream** that allows object assignment. The predefined object **cin** is an object of this class and thus may be reassigned at run time to a different **istream** object.

A program that normally expects input from **stdin**, for example, could be temporarily directed to accept its input from a disk file.

Predefined Objects The **cin** object is a predefined object of class **ostream_withassign**. It is connected to **stdin** (standard input, file descriptor 0).

The objects **cin**, **cerr**, and **clog** are tied to **cout** so that use of any of these may cause **cout** to be flushed.

#include <iostream.h>

See Also **ostream_withassign**

Public Members

Construction/Destruction

istream_withassign	Constructs an **istream_withassign** object.
~istream_withassign	Destroys an **istream_withassign** object.

Operators

operator =	Indicates an assignment operator.

Member Functions

istream_withassign::istream_withassign

Syntax

istream_withassign(streambuf* *psb* **);**

istream_withassign();

Parameters

psb
A pointer to an existing object of a **streambuf**-derived class.

Remarks

The first constructor creates a ready-to-use object of type **istream_withassign**, complete with attached **streambuf** object.

The second constructor creates an object but does not initialize it. You must subsequently use the second variation of the **istream_withassign** assignment operator to attach the **streambuf** object, or you must use the first variation to initialize this object to match the specified **istream** object.

See Also

istream_withassign::operator =

istream_withassign::~istream_withassign

Syntax

~istream_withassign();

Remarks

Destructor for the **istream_withassign** class.

Operators

istream_withassign::operator =

Syntax

istream& operator =(const istream& *ris* **);**

istream& operator =(streambuf* *psb* **);**

Remarks

The first overloaded assignment operator assigns the specified **istream** object to this **istream_withassign** object.

The second operator attaches a **streambuf** object to an existing **istream_withassign** object, and it initializes the state of the **istream_withassign** object. This operator is often used in conjunction with the **void**-argument constructor.

Example 1

```
char buffer[100];
class xistream; // A special-purpose class derived from istream
extern xistream xin; // An xistream object constructed elsewhere

cin = xin; // cin is reassigned to xin
cin >> buffer; // xin used instead of cin
```

Example 2

```
char buffer[100];
extern filedesc fd; // A file descriptor for an open file
filebuf fb( fd ); // Construct a filebuf attached to fd

cin = &fb;     // fb associated with cin
cin >> buffer; // cin now gets its intput from the fb file
```

See Also

istream_withassign::istream_withassign, cin

class istrstream : public istream

The **istrstream** class supports input streams that have character arrays as a source. You must allocate a character array prior to the construction of an **istrstream** object. All the **istream** operators and functions (including seeking) can then be used on this character data.

You must be aware that there is a get pointer working behind the scenes in the attached **strstreambuf** class. This pointer advances as you extract fields from the stream's array. The only way you can make it go backwards is to use the **istream::seekg** function. If the get pointer reaches the end of the string (and sets the **ios::eof** flag), then you must call **clear** before **seekg**.

#include <strstrea.h>

See Also **strstreambuf**, **streambuf**, **strstream**, **ostrstream**

Public Members

Construction/Destruction

istrstream	Constructs an **istrstream** object.
~istrstream	Destroys an **istrstream** object.

Other Functions

rdbuf	Returns a pointer to the stream's associated **strstreambuf** object.
str	Returns a character array pointer to the string stream's contents.

Member Functions

istrstream::istrstream

Syntax

istrstream(char* *psz* **);**

istrstream(char* *pch***, int** *nLength* **);**

Parameters

psz
A null-terminated character array (string).

pch
A character array that is not necessarily null terminated.

nLength
The size (in characters) of *pch*. If 0, then *pch* is assumed to point to a null-terminated array; if less than 0, then the array is assumed to have unlimited length.

Remarks

The first constructor uses the specified *psz* buffer to make an **istrstream** object with length corresponding to the string length.

The second constructor makes an **istrstream** object out of the first *nLength* characters of the *pch* buffer.

Both constructors automatically construct a **strstreambuf** object that manages the specified character buffer.

istrstream::~istrstream

Syntax

~istrstream();

Remarks

Destroys an **istrstream** object and its associated **strstreambuf** object. The character buffer is not released because it was allocated by the user prior to **istrstream** construction.

istrstream::rdbuf

Syntax

strstreambuf* rdbuf() const;

Remarks

Returns a pointer to the **strstreambuf** buffer object that is associated with this stream. Note that this is not the character buffer itself; the **strstreambuf** object contains a pointer to the character area.

See Also

istrstream::str

istrstream::str

Syntax

char* str();

Remarks

Returns a pointer to the string stream's character array. This pointer corresponds to the array used to construct the **istrstream** object.

See Also

istrstream::istrstream

class ofstream : public ostream

The **ofstream** class is an **ostream** derivative specialized for disk file output. All of its constructors automatically create and associate a **filebuf** buffer object.

The **filebuf** class documentation describes the get and put areas and their associated pointers. Only the put area and the put pointer are active for the **ofstream** class.

#include <fstream.h>

See Also **filebuf**, **streambuf**, **ifstream**, **fstream**

Public Members

Construction/Destruction

ofstream	Constructs an **ofstream** object.
~ofstream	Destroys an **ofstream** object.

Operations

open	Opens a file and attaches it to the **filebuf** object and thus to the stream.
close	Flushes any waiting output and closes the stream's file.
setbuf	Associates the specified reserve area to the stream's **filebuf** object.
setmode	Sets the stream's mode to binary or text.
attach	Attaches the stream (through the **filebuf** object) to an open file.

Status/Information

rdbuf	Gets the stream's **filebuf** object.
fd	Returns the file descriptor associated with the stream.
is_open	Tests whether the stream's file is open.

Member Functions

ofstream::attach

Syntax

void attach(filedesc *fd* **);**

Parameters

fd

A file descriptor as returned by a call to the run-time function **_open** or **_sopen**; **filedesc** is a **typedef** equivalent to **int**.

Remarks

Attaches this stream to the open file specified by *fd*. The function fails when the stream is already attached to a file. In that case, the function sets **ios::failbit** in the stream's error state.

See Also

filebuf::attach, **ofstream::fd**

ofstream::close

Syntax

void close();

Remarks

Calls the **close** member function for the associated **filebuf** object. This function, in turn, flushes any waiting output, closes the file, and disconnects the file from the **filebuf** object. The **filebuf** object is not destroyed.

The stream's error state is cleared unless the call to **filebuf::close** fails.

See Also

filebuf::close, **ofstream::open**, **ofstream::is_open**

ofstream::fd

Syntax

filedesc fd() const;

Remarks

Returns the file descriptor associated with the stream. **filedesc** is a **typedef** equivalent to **int**. Its value is supplied by the underlying file system.

See Also

filebuf::fd, **ofstream::attach**

ofstream::is_open

Syntax

int is_open() const;

Remarks

Returns a nonzero value if this stream is attached to an open disk file identified by a file descriptor; otherwise 0.

See Also

filebuf::is_open, **ofstream::open**, **ofstream::close**

ofstream::ofstream

Syntax

ofstream();

ofstream(const char* *szName*, **int** *nMode* **= ios::out,**
int *nProt* **= filebuf::openprot);**

ofstream(filedesc *fd* **);**

ofstream(filedesc *fd*, **char*** *pch*, **int** *nLength* **);**

Parameters

szName
The name of the file to be opened during construction.

nMode
An integer that contains mode bits defined as **ios** enumerators that can be combined with the bitwise-OR (|) operator:

Value	Meaning
ios::app	The function performs a seek to the end of file. When new bytes are written to the file, they are always appended to the end, even if the position is moved with the **ostream::seekp** function.
ios::ate	The function performs a seek to the end of file. When the first new byte is written to the file, it is appended to the end, but when subsequent bytes are written, they are written to the current position.
ios::in	If this mode is specified, then the original file (if it exists), will not be truncated.
ios::out	The file is opened for output (implied for all **ofstream** objects).
ios::trunc	If the file already exists, its contents are discarded. This mode is implied if **ios::out** is specified and **ios::ate**, **ios::app**, and **ios:in** are not specified.
ios::nocreate	If the file does not already exist, the function fails.
ios::noreplace	If the file already exists, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

nProt
The file protection specification; defaults to the static integer **filebuf::openprot** that is equivalent to **filebuf::sh_compat**. The possible *nProt* values are:

Value	Meaning
filebuf::sh_compat	Compatibility share mode.
filebuf::sh_none	Exclusive mode; no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The **filebuf::sh_read** and **filebuf::sh_write** modes can be combined with the logical OR (|) operator.

fd
A file descriptor as returned by a call to the run-time function **_open** or **_sopen**. **filedesc** is a **typedef** equivalent to **int**.

pch
Pointer to a previously allocated reserve area of length *nLength*. A **NULL** value (or *nLength* = 0) indicates that the stream will be unbuffered.

nLength
The length (in bytes) of the reserve area (0 = unbuffered).

Remarks

The four **ofstream** constructors are described as follows:

Constructor	Description
ofstream()	Constructs an **ofstream** object without opening a file.
ofstream(const char*, int, int)	Contructs an **ofstream** object, opening the specified file.
ofstream(filedesc)	Constructs an **ofstream** object that is attached to an open file.
ofstream(filedesc, char*, int)	Constructs an **ofstream** object that is associated with a **filebuf** object. The **filebuf** object is attached to an open file and to a specified reserve area.

All **ofstream** constructors construct a **filebuf** object. The first three use an internally allocated reserve area, but the fourth uses a user-allocated area. The user-allocated area is not automatically released during destruction.

ofstream::~ofstream

Syntax

~ofstream();

Remarks

Flushes the buffer, then destroys an **ofstream** object along with its associated **filebuf** object. The file is closed only if was opened by the constructor or by the **open** member function.

The **filebuf** destructor releases the reserve buffer only if it was internally allocated.

ofstream::open

Syntax

void open(const char* *szName*, **int** *nMode* **= ios::out,**
int *nProt* **= filebuf::openprot);**

Parameters

szName
The name of the file to be opened during construction.

nMode
An integer containing mode bits defined as **ios** enumerators that can be combined with the OR (|) operator. See the **ofstream** constructor for a list of the enumerators. The **ios::out** mode is implied.

nProt
The file protection specification; defaults to the static integer **filebuf::openprot**. See the **ofstream** constructor for a list of the other allowed values.

Remarks

Opens a disk file and attaches it to the stream's **filebuf** object. If the **filebuf** object is already attached to an open file, or if a **filebuf** call fails, the **ios::failbit** is set. If the file is not found, then the **ios::failbit** is set only if the **ios::nocreate** mode was used.

See Also

filebuf::open, **ofstream::ofstream**, **ofstream::close**, **ofstream::is_open**

ofstream::rdbuf

Syntax

filebuf* rdbuf() const;

Remarks

Returns a pointer to the **filebuf** buffer object that is associated with this stream. (Note that this is not the character buffer; the **filebuf** object contains a pointer to the character area.)

Example

```
extern ofstream ofs;
int fd = ofs.rdbuf()->fd(); // Get the file descriptor for ofs
```

ofstream::setbuf

Syntax

streambuf* setbuf(char* *pch*, **int** *nLength* **);**

Parameters

pch
A pointer to a previously allocated reserve area of length *nLength.* A **NULL** value indicates an unbuffered stream.

nLength
The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Remarks

Attaches the specified reserve area to the stream's **filebuf** object. If the file is open and a buffer has already been allocated, the function returns **NULL**; otherwise it returns a pointer to the **filebuf** cast as a **streambuf**. The reserve area will not be released by the destructor.

ofstream::setmode

Syntax

int setmode(int *nMode* = **filebuf::text);**

Parameters

nMode
An integer that must be one of the static **filebuf** constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return–linefeed pairs).
filebuf::binary	Binary mode (no translation).

Remarks

This function sets the binary/text mode of the stream's **filebuf** object. It may be called only after the file is opened.

Return Value

The previous mode; –1 if the parameter is invalid, the file is not open, or the mode cannnot be changed.

See Also

ios binary manipulator, **ios text** manipulator

class ostream : virtual public ios

The **ostream** class provides the basic capability for sequential and random-access output. An **ostream** object has a **streambuf**-derived object attached, and the two classes work together; the **ostream** class does the formatting, and the **streambuf** class does the low-level buffered output.

You can use **ostream** objects for sequential disk output if you first construct an appropriate **filebuf** object. (The **filebuf** class is derived from **streambuf**.) More often, you will use the predefined stream objects **cout**, **cerr**, and **clog** (actually objects of class **ostream_withassign**), or you will use objects of classes **ofstream** (disk file streams) and **ostrstream** (string streams).

All of the **ostream** member functions write unformatted data; formatted output is handled by the insertion operators.

Derivation

It is not always necessary to derive from **ostream** to add functionality to a stream; consider deriving from **streambuf** instead, as illustrated in Chapter 19 of the *Class Libraries User's Guide*. The **ofstream** and **ostrstream** classes are examples of **ostream**-derived classes that construct member objects of predetermined derived **streambuf** classes.

You can add manipulators without deriving a new class.

If you add new insertion operators for a derived **ostream** class, then the rules of C++ dictate that you must reimplement all the base class insertion operators. If, however, you reimplement the operators through inline equivalence, no extra code will be generated. For example,

```
class xstream : public ostream
{
public:
    // Constructors, etc.
    // ........
    inline xstream& operator << ( char ch ) // insertion for char
    {
          return (xstream&)ostream::operator << ( ch );
    }
    // ........
    // Insertions for other types
};
```

#include <iostream.h>

See Also

streambuf, **ofstream**, **ostrstream**, **cout**, **cerr**, **clog**

Public Members

Construction/Destruction

ostream	Constructs an **ostream** object that is attached to an existing **streambuf** object.
~ostream	Destroys an **ostream** object.

Prefix/Suffix Functions

opfx	Output prefix function, called prior to insertion operations to check for error conditions, and so forth.
osfx	Output suffix function, called after insertion operations; flushes the stream's buffer if it is unit buffered.

Unformatted Output

put	Inserts a single byte into the stream.
write	Inserts a series of bytes into the stream.

Other Functions

flush	Flushes the buffer associated with this stream.
seekp	Changes the stream's put pointer.
tellp	Gets the value of the stream's put pointer.

Operators

operator <<	An insertion operator for various types.

Manipulators

endl	Inserts a newline sequence and flushes the buffer.
ends	Inserts a null character to terminate a string.
flush	Flushes the stream's buffer.

Member Functions

ostream::flush

Syntax **ostream& flush();**

Remarks Flushes the buffer associated with this stream. The **flush** function calls the **sync** function of the associated **streambuf**.

See Also **ostream flush** manipulator, **streambuf::sync**

---

ostream::opfx

Syntax **int opfx();**

Remarks This output prefix function is called before every insertion operation. If another **ostream** object is tied to this stream then the **opfx** function flushes that stream.

Return Value If the **ostream** object's error state is not 0, **opfx** returns 0 immediately; otherwise it returns a nonzero value.

---

ostream::osfx

Syntax **void osfx();**

Remarks This output suffix function is called after every insertion operation. It flushes the **ostream** object if **ios::unitbuf** is set. It flushes **stdout** and **stderr** if **ios::stdio** is set.

ostream::ostream

Syntax

Public:
  ostream(streambuf* *psb* **);**

Protected:
  ostream( );

Parameters

psb
  A pointer to an existing object of a **streambuf**-derived class.

Remarks

Constructs an object of type **ostream**.

See Also

ios::init

ostream::~ostream

Syntax

virtual ~ostream();

Remarks

Destroys an **ostream** object. The output buffer is flushed as appropriate. The attached **streambuf** object is destroyed only if it was allocated internally within the **ostream** constructor.

ostream::put

Syntax

ostream& put(char *ch* **);**

Parameters

ch
  The character to insert.

Remarks

This function inserts a single character into the output stream.

ostream::seekp

Syntax

ostream& seekp(streampos *pos* **);**

ostream& seekp(streamoff *off*, **ios::seek_dir** *dir* **);**

Parameters

pos
The new position value; **streampos** is a **typedef** equivalent to **long**.

off
The new offset value; **streamoff** is a **typedef** equivalent to **long**.

dir
The seek direction specified by the enumerated type **ios::seek_dir**, as follows:

Value	Meaning
ios::beg	Seek from the beginning of the stream.
ios::cur	Seek from the current position in the stream.
ios::end	Seek from the end of the stream.

Remarks

Changes the position value for the stream. Not all derived classes of **ostream** need support positioning. For file streams, the position is the byte offset from the beginning of the file; for string streams, it is the byte offset from the beginning of the string.

See Also

ostream::tellp, **istream::seekg**, **istream::tellg**

ostream::tellp

Syntax

streampos tellp();

Remarks

Gets the position value for the stream. Not all derived classes of **ostream** need support positioning. For file streams, the position is the byte offset from the beginning of the file; for string streams, it is the byte offset from the beginning of the string. Gets the value for the stream's put pointer.

Return Value

A **streampos** type that corresponds to a **long**.

See Also

ostream::seekp, **istream::tellg**, **istream::seekg**

ostream::write

Syntax

ostream& write(const char* *pch*, **int** *nCount* **);**

ostream& write(const unsigned char* *puch*, **int** *nCount* **);**

ostream& write(const signed char* *psch*, **int** *nCount* **);**

Parameters

pch, puch, psch
A pointer to a character array.

nCount
The number of characters to be written.

Remarks

Inserts a specified number of bytes from a buffer into the stream. If the underlying file was opened in text mode, additional carriage return characters may be inserted. The **write** function is useful for binary stream output.

Operators

ostream::operator <<

Syntax

ostream& operator <<(char *ch* **);**

ostream& operator <<(unsigned char *uch* **);**

ostream& operator <<(signed char *sch* **);**

ostream& operator <<(const char* *psz* **);**

ostream& operator <<(const unsigned char **pusz* **);**

ostream& operator <<(const signed char **pssz* **);**

ostream& operator <<(short *s* **);**

ostream& operator <<(unsigned short *us* **);**

ostream& operator <<(int *n* **);**

ostream& operator <<(unsigned int *un* **);**

ostream& operator <<(long *l* **);**

ostream& operator <<(unsigned long *ul* **);**

ostream& operator <<(float *f* **);**

ostream& operator <<(double *d* **);**

ostream& operator <<(long double *ld* **);**

ostream& operator <<(void* *pv* **);**

ostream& operator <<(streambuf* *psb* **);**

ostream& operator <<(ostream& (*fcn***)(ostream&));**

ostream& operator <<(ios& (*fcn***)(ios&));**

Remarks

These overloaded operators insert their argument into the stream. The last two variations allow the use of manipulators that are defined for both **ostream** and **ios**.

Manipulators

ostream& endl

Syntax **endl**

Remarks This manipulator, when inserted into an output stream, inserts a newline character and then flushes the buffer.

ostream& ends

Syntax **ends**

Remarks This manipulator, when inserted into an output stream, inserts a null-terminator character. It is particularly useful for **ostrstream** objects.

ostream& flush

Syntax **flush**

Remarks This manipulator, when inserted into an output stream, flushes the output buffer by calling the **streambuf::sync** member function.

See Also **ostream::flush**, **streambuf::sync**

class ostream_withassign : public ostream

The **ostream_withassign** class is a variant of **ostream** that allows object assignment. The predefined objects **cout**, **cerr**, and **clog** are objects of this class and thus may be reassigned at run time to a different **ostream** object.

A program that normally sends output to **stdout**, for example, could be temporarily directed to send its output to a disk file.

Predefined Objects

There are three predefined objects of class **ostream_withassign**. They are connected as follows:

cout
Standard output (file descriptor 1).

cerr
Unit buffered standard error (file descriptor 2).

clog
Fully buffered standard error (file descriptor 2).

Unit buffering, as used by **cerr**, means that characters are flushed after each insertion operation. The objects **cin**, **cerr**, and **clog** are tied to **cout** so that use of any of these will cause **cout** to be flushed.

#include <iostream.h>

See Also

istream_withassign

Public Members

Construction/Destruction

ostream_withassign	Constructs an **ostream_withassign** object.
~ostream_withassign	Destroys an **ostream_withassign** object.

Operators

operator =	Assignment operator.

Member Functions

ostream_withassign::ostream_withassign

Syntax

ostream_withassign(streambuf* *psb* **);**

ostream_withassign();

Parameters

psb
A pointer to an existing object of a **streambuf**-derived class.

Remarks

The first constructor makes a ready-to-use object of type **ostream_withassign**, complete with an attached **streambuf** object.

The second constructor makes an object but does not initialize it. You must subsequently use the **streambuf** assignment operator to attach the **streambuf** object, or you must use the **ostream** assignment operator to initialize this object to match the specified object.

See Also

ostream_withassign::operator =

ostream_withassign::~ostream_withassign

Syntax

~ostream_withassign();

Remarks

Destructor for the **ostream_withassign** class.

Operators

ostream_withassign::operator =

Syntax

ostream& operator =(ostream& *ros* **);**

ostream& operator =(streambuf* *sbp* **);**

Remarks

The first overloaded assignment operator assigns the specified **ostream** object to this **ostream_withassign** object.

The second operator attaches a **streambuf** object to an existing **ostream_withassign** object, and it initializes the state of the **ostream_withassign** object. This operator is often used in conjunction with the **void**-argument constructor.

Example

```
filebuf fb( "test.dat" ); // Filebuf object attached to "test.dat"
cout = &fb;         // fb associated with cout
cout << "testing"; // Message goes to "test.dat" instead of stdout
```

See Also

ostream_withassign::ostream_withassign, cout

class ostrstream : public ostream

The **ostrstream** class supports output streams that have character arrays as a destination. You can allocate a character array prior to construction, or the constructor can internally allocate an expandable array. All the **ostream** operators and functions can then be used to fill the array.

You must be aware that there is a put pointer working behind the scenes in the attached **strstreambuf** class. This pointer advances as you insert fields into the stream's array. The only way you can make it go backwards is to use the **ostream::seekp** function. If the put pointer reaches the end of user-allocated memory (and sets the **ios::eof** flag), then you must call **clear** before **seekp**.

#include <strstrea.h>

See Also **strstreambuf**, **streambuf**, **strstream**, **istrstream**

Public Members

Construction/Destruction

ostrstream	Constructs an **ostrstream** object.
~ostrstream	Destroys an **ostrstream** object.

Other Functions

pcount	Returns the number of bytes that have been stored in the stream's buffer.
rdbuf	Returns a pointer to the stream's associated **strstreambuf** object.
str	Returns a character array pointer to the string stream's contents and freezes the array.

Member Functions

ostrstream::ostrstream

Syntax

ostrstream();

ostrstream(char* *pch*, **int** *nLength*, **int** *nMode* = **ios::out);**

Parameters

pch

A character array that is large enough to accommodate future output stream activity.

nLength

The size (in characters) of *pch*. If 0, then *pch* is assumed to point to a null-terminated array and **strlen(** *pch* **)** is used as the length; if less than 0, then the array is assumed to have infinite length.

nMode

The stream-creation mode. Must be one of the following enumerators as defined in class **ios**:

Value	Meaning
ios::out	Default; storing begins at *pch*
ios::ate	The *pch* parameter is assumed to be a null-terminated array; storing begins at the **NULL** character
ios::app	Same as **ios::ate**

Remarks

The first constructor makes an **ostrstream** object that uses an internal, dynamic buffer.

The second constructor makes an **ostrstream** object out of the first *nLength* characters of the *pch* buffer. The stream will not accept characters once the length reaches *nLength*.

ostrstream::~ostrstream

Syntax

~ostrstream();

Remarks

Destroys an **ostrstream** object and its associated **strstreambuf** object, thus releasing all internally allocated memory. If you used the **void**-argument constructor, then the internally allocated character buffer is released; otherwise, you must release it yourself.

An internally allocated character buffer will not be released if it was previously frozen by an **str** or **strstreambuf::freeze** function call.

See Also

ostrstream::str, **strstreambuf::freeze**

ostrstream::pcount

Syntax

int pcount() const;

Remarks

Returns the number of bytes that have been stored in the buffer. This information is especially useful when you have stored binary data in the object.

ostrstream::rdbuf

Syntax

strstreambuf* rdbuf() const;

Remarks

Returns a pointer to the **strstreambuf** buffer object that is associated with this stream. Note that this is not the character buffer; the **strstreambuf** object contains a pointer to the character area.

See Also

ostrstream::str

ostrstream::str

Syntax

char* str();

Remarks

Returns a pointer to the internal character array. If the stream was built with the **void**-argument constructor, then **str** freezes the array. You must not send characters to a frozen stream, and you are responsible for deleting the array. You can, however, subsequently unfreeze the array by calling **rdbuf->freeze(0**).

If the stream was built with the constructor that specified the buffer, then the pointer contains the same address as the array used to construct the **ostrstream** object.

See Also

ostrstream::ostrstream, **ostrstream::rdbuf**, **strstreambuf::freeze**

class stdiobuf : public streambuf

The run-time library supports three conceptual sets of I/O functions: iostreams (C++ only), standard I/O (the functions declared in STDIO.H), and low-level I/O (the functions declared in IO.H). The **stdiobuf** class is a derived class of **streambuf** that is specialized for buffering to and from the standard I/O system.

Because the standard I/O system does its own internal buffering, the extra buffering level provided by **stdiobuf** may reduce overall input/output efficiency. The **stdiobuf** class is useful when you need to mix iostream I/O with standard I/O (**printf** and so forth).

You can avoid use of the **stdiobuf** class if you use the **filebuf** class. You must also use the stream class's **ios::flags** member function to set the **ios::stdio** format flag value.

#include <stdiostr.h>

See Also **stdiostream**, **filebuf**, **strstreambuf**, **ios::flags**

Public Members

Construction/Destruction

stdiobuf	Constructs a **stdiobuf** object from a **FILE** pointer.
~stdiobuf	Destroys a **stdiobuf** object.

Other Functions

stdiofile	Gets the file that is attached to the **stdiofile** object.

Member Functions

stdiobuf::stdiobuf

Syntax

stdiobuf(FILE* *fp* **);**

Parameters

fp
A standard I/O file pointer (can be obtained through an **fopen** or **_fsopen** call).

Remarks

Objects of class **stdiobuf** are constructed from open standard I/O files, including **stdin**, **stdout**, and **stderr**. The object is unbuffered by default.

stdiobuf::~stdiobuf

Syntax

~stdiobuf();

Remarks

Destroys a **stdiobuf** object and, in the process, flushes the put area. The destructor does not close the attached file.

stdiobuf::stdiofile

Syntax

FILE* stdiofile();

Remarks

Returns the standard I/O file pointer associated with a **stdiobuf** object.

class stdiostream : public iostream

The **stdiostream** class makes I/O calls (through the **stdiobuf** class) to the standard I/O system, which does its own internal buffering. Calls to the functions declared in STDIO.H, such as **printf**, can be mixed with **stdiostream** I/O calls.

This class is included for compatibility with earlier stream libraries. You can avoid use of the **stdiostream** class if you use the **ostream** or **istream** class with an associated **filebuf** class. You must also use the stream class's **ios::flags** member function to set the **ios::stdio** format flag value.

The use of the **stdiobuf** class may reduce efficiency because it imposes an extra level of buffering. Do not use this feature unless you need to mix iostream library calls with standard I/O calls for the same file.

#include <stdiostr.h>

See Also **stdiobuf**, **ios::flags**

Public Members

Construction/Destruction

stdiostream	Constructs a **stdiostream** object that is associated with a standard I/O **FILE** pointer.
~stdiostream	Destroys a **stdiostream** object (virtual).

Other Functions

rdbuf	Gets the stream's **stdiobuf** object.

Member Functions

stdiostream::rdbuf

Syntax

stdiobuf* rdbuf() const;

Remarks

Returns a pointer to the **stdiobuf** buffer object that is associated with this stream. The **rdbuf** function is useful when you need to call **stdiobuf** member functions.

stdiostream::stdiostream

Syntax

stdiostream(FILE* *fp* **);**

Parameters

fp
A standard I/O file pointer (can be obtained through an **fopen** or **_fsopen** call). Could be **stdin**, **stdout**, or **stderr**.

Remarks

Objects of class **stdiostream** are constructed from open standard I/O files. An unbuffered **stdiobuf** object is automatically associated, but the standard I/O system provides its own buffering.

Example

```
stdiostream myStream( stdout );
```

stdiostream::~stdiostream

Syntax

virtual ~stdiostream();

Remarks

This destructor destroys the **stdiobuf** object associated with this stream; however, the attached file is not closed.

class streambuf

All the iostream classes in the **ios** hierarchy depend on an attached **streambuf** class for the actual I/O processing. This class is an abstract class, but the iostream class library contains the following derived buffer classes for use with streams:

filebuf
Buffered disk file I/O.

strstreambuf
Stream data held entirely within an in-memory byte array.

stdiobuf
Disk I/O with buffering done by the underlying standard I/O system.

All **streambuf** objects, when configured for buffered processing, maintain a fixed memory buffer, called a reserve area, that can be dynamically partitioned into a get area for input and a put area for output. These areas may or may not overlap. Protected member functions allow access and manipulation of a get pointer for character retrieval and a put pointer for character storage. The exact behavior of the buffers and pointers depends on the implementation of the derived class.

The capabilities of the iostream classes can be extended significantly through the derivation of new **streambuf** classes. The **ios** class tree supplies the programming interface and all formatting features, but the **streambuf** class does the real work. The **ios** classes call the **streambuf** public members, including a set of virtual functions.

The **streambuf** class provides a default implementation of certain virtual member functions. The "Default Implementation" section for each such function suggests function behavior for the derived class.

#include <iostream.h>

Public Members

Character Input Functions

in_avail	Returns the number of characters in the get area.
sgetc	Returns the character at the get pointer, but does not move the pointer.
snextc	Advances the get pointer, and then returns the next character.
sbumpc	Returns the current character, and then advances the get pointer.

stossc	Moves the get pointer forward one position, but does not return a character.
sputbackc	Attempts to move the get pointer back one position.
sgetn	Gets a sequence of characters from the **streambuf** object's buffer.

Character Output Functions

out_waiting	Returns the number of characters in the put area.
sputc	Stores a character in the put area and advances the put pointer.
sputn	Stores a sequence of characters in the **streambuf** object's buffer and advances the put pointer.

Diagnostic Functions

dbp	Prints buffer statistics and pointer values.

Virtual Functions

sync	Empties the get area and the put area.
setbuf	Attempts to attach a reserve area to the **streambuf** object.
seekoff	Seeks to a specified offset.
seekpos	Seeks to a specified position.
overflow	Empties the put area.
underflow	Fills the get area if necessary.
pbackfail	Augments the **sputbackc** function.

Protected Members

Construction/Destruction

streambuf	Constructors for use in derived classes.
~streambuf	Virtual destructor.

Other Protected Member Functions

base	Returns a pointer to the start of the reserve area.
ebuf	Returns a pointer to the end of the reserve area.
blen	Returns the size of the reserve area.
pbase	Returns a pointer to the start of the put area.
pptr	Returns the put pointer.
epptr	Returns a pointer to the end of the put area.
eback	Returns the lower bound of the get area.
gptr	Returns the get pointer.
egptr	Returns a pointer to the end of the get area.
setp	Sets all the put area pointers.
setg	Sets all the get area pointers.
pbump	Increments the put pointer.
gbump	Increments the get pointer.
setb	Sets up the reserve area.
unbuffered	Tests or sets the **streambuf** buffer state variable.
allocate	Allocates a buffer, if needed, by calling **doalloc**.
doallocate	Allocates a reserve area (virtual function).

Member Functions

streambuf::allocate

Syntax

Protected:
  int allocate();

Remarks

Calls the virtual function **doallocate** to set up a reserve area. If a reserve area already exists or if the **streambuf** object is unbuffered, **allocate** returns 0. If the space allocation fails, **allocate** returns **EOF**.

See Also

streambuf::doallocate, **streambuf::unbuffered**

streambuf::base

Syntax

Protected:
  char* base() const;

Remarks

Returns a pointer to the first byte of the reserve area. The reserve area consists of space between the pointers returned by **base** and **ebuf**.

See Also

streambuf::ebuf, **streambuf::setb**, **streambuf::blen**

streambuf::blen

Syntax

Protected:
  int blen() const;

Remarks

Returns the size, in bytes, of the reserve area.

See Also

streambuf::base, **streambuf::ebuf**, **streambuf::setb**

streambuf::dbp

Syntax

void dbp();

Remarks

Writes ASCII debugging information directly on **stdout**. Treat this function as part of the protected interface.

Some sample output follows:

```
STREAMBUF DEBUG INFO: this = 00E7:09DC, _fAlloc=1
 base()=00E7:0A0C, ebuf()=00E7:0C0C,  blen()=512
pbase()=00E7:0A0C, pptr()=00E7:0A22, epptr()=00E7:0C0C
eback()=0000:0000, gptr()=0000:0000, egptr()=0000:0000
```

streambuf::doallocate

Syntax

Protected:
  virtual int doallocate();

Remarks

Called by **allocate** when space is needed. The **doallocate** function must allocate a reserve area, then call **setb** to attach the reserve area to the **streambuf** object. If the reserve area allocation fails, **doallocate** returns **EOF**.

Default Implementation

Attempts to allocate a reserve area using operator **new**.

See Also

streambuf::allocate, **streambuf::setb**

streambuf::eback

Syntax

Protected:
  char* eback() const;

Remarks

Returns the lower bound of the get area. Space between the **eback** and **gptr** pointers is available for putting a character back to the stream.

See Also

streambuf::sputbackc, streambuf::gptr

streambuf::ebuf

Syntax

Protected:
  char* ebuf() const;

Remarks

Returns a pointer to the byte after the last byte of the reserve area. The reserve area consists of space between the pointers returned by **base** and **ebuf**.

See Also

streambuf::base, streambuf::setb, streambuf::blen

streambuf::egptr

Syntax

Protected:
  char* egptr() const;

Remarks

Returns a pointer to the byte after the last byte of the get area.

See Also

streambuf::setg, streambuf::eback, streambuf::gptr

streambuf::epptr

Syntax

Protected:
char* epptr() const;

Remarks

Returns a pointer to the byte after the last byte of the put area.

See Also

streambuf::setp, **streambuf::pbase**, **streambuf::pptr**

streambuf::gbump

Syntax

Protected:
void gbump(int *nCount* **);**

Parameters

nCount
The number of bytes to increment the get pointer. May be positive or negative.

Remarks

Increments the get pointer. No bounds checks are made on the result.

See Also

streambuf::pbump

streambuf::gptr

Syntax

Protected:
char* gptr() const;

Remarks

Returns a pointer to the next character to be fetched from the **streambuf** buffer. This pointer is known as the get pointer.

See Also

streambuf::setg, **streambuf::eback**, **streambuf::egptr**

streambuf::in_avail

Syntax

int in_avail() const;

Remarks

Returns the number of characters in the get area that are available for fetching. These characters are between the **gptr** and **egptr** pointers and may be fetched with a guarantee of no errors.

streambuf::out_waiting

Syntax

int out_waiting() const;

Remarks

Returns the number of characters in the put area that have not been sent to the final output destination. These characters are between the **pbase** and **pptr** pointers.

streambuf::overflow

Syntax

virtual int overflow(int *nCh* = EOF) = 0;

Parameters

nCh
EOF or the character to output.

Remarks

The virtual **overflow** function, together with the **sync** and **underflow** functions, defines the characteristics of the **streambuf**-derived class. Each derived class might implement **overflow** differently, but the interface with the calling stream class is the same.

The **overflow** function is most frequently called by public **streambuf** functions like **sputc** and **sputn** when they sense that the put area is full, but other classes, including the stream classes, can call **overflow** anytime.

The function "consumes" the characters in the put area between the **pbase** and **pptr** pointers and then reinitializes the put area. The **overflow** function must also consume *nCh* (if *nCh* is not **EOF**), or it might choose to put that character in the new put area so that it will be consumed on the next call.

The definition of "consume" varies among derived classes. The **filebuf** class, for example, writes its characters to a file. The **strsteambuf** class, on the other hand, keeps them in its buffer and (if the buffer is designated as dynamic) expands the buffer in response to a call to **overflow**. This expansion is achieved by freeing the old buffer and replacing it with a new, larger one. The pointers are adjusted as necessary.

Default Implementation No default implementation. Derived classes must define this function.

Return Value **EOF** to indicate an error.

See Also **streambuf::pbase**, **streambuf::pptr**, **streambuf::setp**, **streambuf::sync**, **streambuf::underflow**

streambuf::pbackfail

Syntax **virtual int pbackfail(int** *nCh* **);**

Parameters

nCh
The character used in a previous **sputbackc** call.

Remarks This function is called by **sputbackc** if it fails, usually because the **eback** pointer equals the **gptr** pointer. The **pbackfail** function should deal with the situation, if possible, by such means as repositioning the external file pointer.

Default Implementation Returns **EOF**.

Return Value The *nCh* parameter if successful; otherwise **EOF**.

See Also **streambuf::sputbackc**

streambuf::pbase

Syntax

Protected:
char* pbase() const;

Remarks

Returns a pointer to the start of the put area. Characters between the **pbase** pointer and the **pptr** pointer have been stored in the buffer but not flushed to the final output destination.

See Also

streambuf::pptr, **streambuf::setp**, **streambuf::out_waiting**

streambuf::pbump

Syntax

Protected:
void pbump(int *nCount* **);**

Parameters

nCount
The number of bytes to increment the put pointer. May be positive or negative.

Remarks

Increments the put pointer. No bounds checks are made on the result.

See Also

streambuf::gbump, **streambuf::setp**

streambuf::pptr

Syntax

Protected:
char* pptr() const;

Remarks

Returns a pointer to the first byte of the put area. This pointer is known as the put pointer and is the destination for the next character(s) sent to the **streambuf** object.

See Also

streambuf::epptr, **streambuf::pbase**, **streambuf::setp**

streambuf::sbumpc

Syntax

int sbumpc();

Remarks

Returns the current character, and then advances the get pointer. Returns **EOF** if the get pointer is currently at the end of the sequence (equal to the **egptr** pointer).

See Also

streambuf::epptr, **streambuf::gbump**

streambuf::seekoff

Syntax

virtual streampos seekoff(streamoff *off*, **ios::seek_dir** *dir*,
  int *nMode* **= ios::in | ios::out);**

Parameters

off
  The new offset value; **streamoff** is a **typedef** equivalent to **long**.

dir
  The seek direction specified by the enumerated type **seek_dir**, as follows:

Value	Meaning
ios::beg	Seek from the beginning of the stream.
ios::cur	Seek from the current position in the stream.
ios::end	Seek from the end of the stream.

nMode
  An integer that contains a bitwise-OR (|) combination of the enumerators **ios::in** and **ios::out**.

Remarks

Changes the position for the **streambuf** object. Not all derived classes of **streambuf** need to support positioning; however, the **filebuf**, **strstreambuf**, and **stdiobuf** classes do support positioning.

Classes derived from **streambuf** often support independent input and output position values. The *nMode* parameter determines which value(s) is set.

Default Implementation

Returns **EOF**.

Return Value

The new position value. This is the byte offset from the start of the file (or string). If both **ios::in** and **ios::out** are specified, then the function returns the output position. If the derived class does not support positioning, then the function returns **EOF**.

See Also

streambuf::seekpos

streambuf::seekpos

Syntax

virtual streampos seekpos(streampos *pos*, **int** *nMode* **= ios::in | ios::out);**

Parameters

pos

The new position value; **streampos** is a **typedef** equivalent to **long**.

nMode

An integer that contains mode bits defined as **ios** enumerators that can be combined with the OR (|) operator. See **ofstream::ofstream** for a listing of the enumerators.

Remarks

Changes the position, relative to the beginning of the stream, for the **streambuf** object. Not all derived classes of **streambuf** need to support positioning; however, the **filebuf**, **strstreambuf**, and **stdiobuf** classes do support positioning.

Classes derived from **streambuf** often support independent input and output position values. The *nMode* parameter determines which value(s) is set.

Default Implementation

Calls **seekoff((streamoff)** *pos*, **ios::beg,** *nMode*). Thus, to define seeking in a derived class, it is usually necessary to redefine only **seekoff**.

Return Value

The new position value. If both **ios::in** and **ios::out** are specified, then the function returns the output position. If the derived class does not support positioning, then the function returns **EOF**.

See Also

streambuf::seekoff

streambuf::setb

Syntax

Protected:
void setb(char* *pb*, **char*** *peb*, **int** *nDelete* = **0);**

Parameters

pb
The new value for the **base** pointer.

peb
The new value for the **ebuf** pointer.

nDelete
Flag that controls automatic deletion. If *nDelete* is not 0, then the reserve area will be deleted (1) when the **base** pointer is changed by another **setb** call or (2) when the **streambuf** destructor is called.

Remarks

Sets the values of the reserve area pointers. If both *pb* and *peb* are **NULL**, then there is no reserve area. If *pb* is not **NULL** and *peb* is **NULL**, then the reserve area has a length of 0.

See Also

streambuf::base, **streambuf::ebuf**

streambuf::setbuf

Syntax

virtual streambuf* setbuf(char* *pr*, **int** *nLength* **);**

Parameters

pr
A pointer to a previously allocated reserve area of length *nLength.* A **NULL** value indicates an unbuffered stream.

nLength
The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Remarks

Attaches the specified reserve area to the **streambuf** object. Derived classes may or may not use this area.

Default Implementation Accepts the request if there is not a reserved area already.

Return Value A **streambuf** pointer if the buffer is accepted; otherwise **NULL**.

streambuf::setg

Syntax

Protected:
void setg(char* *peb*, **char*** *pg*, **char*** *peg* **);**

Parameters

peb
The new value for the **eback** pointer.

pg
The new value for the **gptr** pointer.

peg
The new value for the **egptr** pointer.

Remarks Sets the values for the get area pointers.

See Also **streambuf::eback**, **streambuf::gptr**, **streambuf::egptr**

streambuf::setp

Syntax

Protected:
void setp(char* *pp*, **char*** *pep* **);**

Parameters

pp
The new value for the **pbase** and **pptr** pointers.

pep
The new value for the **epptr** pointer.

Remarks Sets the values for the put area pointers.

See Also **streambuf::pptr**, **streambuf::pbase**, **streambuf::epptr**

streambuf::sgetc

Syntax **int sgetc();**

Remarks Returns the character at the get pointer. The **sgetc** function does not move the get pointer. Returns **EOF** if there is no character available.

See Also **streambuf::sbumpc**, **streambuf::sgetn**, **streambuf::snextc**, **streambuf::stossc**

streambuf::sgetn

Syntax **int sgetn(char*** *pch*, **int** *nCount* **);**

Parameters

pch
A pointer to a buffer that will receive characters from the **streambuf** object.

nCount
The number of characters to get.

Remarks Gets the *nCount* characters that follow the get pointer and stores them in the area starting at *pch*. When fewer than *nCount* characters remain in the **streambuf** object, **sgetn** fetches whatever characters remain. The function repositions the get pointer to follow the fetched characters.

Return Value The number of characters fetched.

See Also **streambuf::sbumpc**, **streambuf::sgetc**, **streambuf::snextc**, **streambuf::stossc**

streambuf::snextc

Syntax

int snextc();

Remarks

First tests the get pointer, then returns **EOF** if it is already at the end of the get area. Otherwise, it moves the get pointer forward one character and returns the character that follows the new position. It returns **EOF** if the pointer has been moved to the end of the get area.

See Also

streambuf::sbumpc, **streambuf::sgetc**, **streambuf::sgetn**, **streambuf::stossc**

streambuf::sputbackc

Syntax

int sputbackc(char *ch* **);**

Parameters

ch
The character to be put back to the **streambuf** object.

Remarks

Moves the get pointer back one position. The *ch* character must match the character just before the get pointer.

Return Value

EOF on failure.

See Also

streambuf::sbumpc, **streambuf::pbackfail**

streambuf::sputc

Syntax

int sputc(int *nCh* **);**

Parameters

nCh
The character to store in the **streambuf** object.

Remarks

Stores a character in the put area and advances the put pointer.

This public function is available to code outside the class, including the classes derived from **ios**. A derived **streambuf** class can gain access to its buffer directly by using protected member functions.

Return Value The number of characters successfully stored; **EOF** on error.

See Also **streambuf::sputn**

streambuf::sputn

Syntax **int sputn(const char*** *pch*, **int** *nCount* **);**

Parameters

pch
A pointer to a buffer that contains data to be copied to the **streambuf** object.

nCount
The number of characters in the buffer.

Remarks Copies *nCount* characters from *pch* to the **streambuf** buffer following the put pointer. The function repositions the put pointer to follow the stored characters.

Return Value The number of characters stored. This number is usually *nCount* but could be less if an error occurs.

See Also **streambuf::sputc**

streambuf::stossc

Syntax **void stossc();**

Remarks Moves the get pointer forward one character. If the pointer is at the end of the get area already, the function has no effect.

See Also **streambuf::sbumpc**, **streambuf::sgetn**, **streambuf::snextc**, **streambuf::sgetc**

streambuf::streambuf

Syntax

Protected:
streambuf();

Protected:
streambuf(char* *pr*, **int** *nLength* **);**

Parameters

pr
A pointer to a previously allocated reserve area of length *nLength.* A **NULL** value indicates an unbuffered stream.

nLength
The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Remarks

The first constructor makes an uninitialized **streambuf** object. This object is not suitable for use until a **setbuf** call is made. A derived class constructor usually calls **setbuf** or uses the second constructor.

The second constructor initializes the **streambuf** object with the specified reserve area or marks it as unbuffered.

See Also

streambuf::setbuf

streambuf::~streambuf

Syntax

Protected:
virtual ~streambuf();

Remarks

The **streambuf** destructor flushes the buffer if the stream is being used for output.

streambuf::sync

Syntax

virtual int sync();

Remarks

The virtual **sync** function, together with the **overflow** and **underflow** functions, defines the characteristics of the **streambuf**-derived class. Each derived class might implement **sync** differently, but the interface with the calling stream class is the same.

The **sync** function flushes the put area. It also empites the get area and, in the process, sends any unprocessed characters back to the source, if necessary.

Default Implementation

Returns 0 if the get area is empty and there are no more characters to output; otherwise, it returns **EOF**.

Return Value

EOF if an error occurs.

See Also

streambuf::overflow

streambuf::unbuffered

Syntax

Protected:
void unbuffered(int *nState* **);**

Protected:
int unbuffered() const;

Parameters

nState
The value of the buffering state variable; 0 = buffered, nonzero = unbuffered.

Remarks

The first overloaded **unbuffered** function sets the value of the **streambuf** object's buffering state. This variable's primary purpose is to control whether the **allocate** function automatically allocates a reserve area.

The second function returns the current buffering state variable.

See Also

streambuf::allocate, streambuf::doallocate

streambuf::underflow

Syntax

virtual int underflow() = 0;

Remarks

The virtual **underflow** function, together with the **sync** and **overflow** functions, defines the characteristics of the **streambuf**-derived class. Each derived class might implement **underflow** differently, but the interface with the calling stream class is the same.

The **underflow** function is most frequently called by public **streambuf** functions like **sgetc** and **sgetn** when they sense that the get area is empty, but other classes, including the stream classes, can call **underflow** anytime.

The **underflow** function supplies the get area with characters from the input source. If the get area contains characters, then **underflow** returns the first character. If the get area is empty, then it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then **underflow** returns **EOF** and leaves the get area empty.

In the **strstreambuf** class, **underflow** adjusts the **egptr** pointer to access storage that was dynamically allocated by a call to **overflow**.

Default Implementation

No default implementation. Derived classes must define this function.

class strstream : public iostream

The **strstream** class supports I/O streams that have character arrays as a source and destination. You can allocate a character array prior to construction, or the constructor can internally allocate a dynamic array. All the input and output stream operators and functions can then be used to fill the array.

You must be aware that there is a put pointer and a get pointer working independently behind the scenes in the attached **strstreambuf** class. The put pointer advances as you insert fields into the stream's array, and the get pointer advances as you extract fields. The **ostream::seekp** function moves the put pointer, and the **istream::seekg** function moves the get pointer. If either pointer reaches the end of the string (and sets the **ios::eof** flag), then you must call **clear** before seeking.

#include <strstrea.h>

See Also **strstreambuf**, **streambuf**, **istrstream**, **ostrstream**

Public Members

Construction/Destruction

strstream	Constructs a **strstream** object.
~strstream	Destroys a **strstream** object.

Other Functions

pcount	Returns the number of bytes that have been stored in the stream's buffer.
rdbuf	Returns a pointer to the stream's associated **strstreambuf** object.
str	Returns a pointer to the string stream's character buffer and freezes it.

Member Functions

strstream::pcount

Syntax

int pcount() const;

Remarks

Returns the number of bytes that have been stored in the buffer. This information is especially useful when you have stored binary data in the object.

strstream::rdbuf

Syntax

strstreambuf* rdbuf() const;

Remarks

Returns a pointer to the **strstreambuf** buffer object that is associated with this stream. Note that this is not the character buffer; the **strstreambuf** object contains a pointer to the character area.

See Also

strstream::str

strstream::str

Syntax

char* str();

Remarks

Returns a pointer to the internal character array. If the stream was built with the **void**-argument constructor, then **str** freezes the array. You must not send characters to a frozen stream, and you are responsible for deleting the array. You can unfreeze the the stream by calling **rdbuf->freeze(0)**.

If the stream was built with the constructor that specified the buffer, then the pointer contains the same address as the array used to construct the **ostrstream** object.

See Also

strstreambuf::freeze, strstream::rdbuf

strstream::strstream

Syntax

strstream();

strstream(char* *pch*, **int** *nLength*, **int** *nMode* **);**

Parameters

pch
A character array that is large enough to accommodate future output stream activity.

nLength
The size (in characters) of *pch*. If 0, then *pch* is assumed to point to a null-terminated array; if less than 0, then the array is assumed to have infinite length.

nMode
The stream creation mode. Must be one of the following enumerators as defined in class **ios**:

Value	Meaning
ios::in	Retrieval begins at the beginning of the array.
ios::out	By default, storing begins at *pch*.
ios::ate	The *pch* parameter is assumed to be a null-terminated array; storing begins at the **NULL** character.
ios::app	Same as **ios::ate**.

The use of the **ios::in** and **ios::out** flags is optional for this class; both input and output are implied.

Remarks

The first constructor makes an **strstream** object that uses an internal, dynamic buffer that is initially empty.

The second constructor makes an **strstream** object out of the first *nLength* characters of the *psc* buffer. The stream will not accept characters once the length reaches *nLength.*

strstream::~strstream

Syntax **~strstream();**

Remarks Destroys a **strstream** object and its associated **strstreambuf** object, thus releasing all internally allocated memory. If you used the **void**-argument constructor, then the internally allocated character buffer is released; otherwise, you must release it yourself.

An internally allocated character buffer will not be released if it was previously frozen by calling **rdbuf->freeze(0)**.

See Also **strstream::rdbuf**

class strstreambuf : public streambuf

The **strstreambuf** class is a derived class of **streambuf** that manages an in-memory character array.

The file stream classes, **ostrstream**, **istrstream**, and **strstream**, use **strstreambuf** member functions to fetch and store characters. Some of these member functions are virtual functions defined for the **streambuf** class.

The reserve area, put area, and get area were introduced in the **streambuf** class description. For **strsteambuf** objects, the put area is the same as the get area, but the get pointer and the put pointer move independently.

#include <strstrea.h>

See Also **istrstream**, **ostrstream**, **filebuf**, **stdiobuf**

Public Members

Construction/Destruction

strstreambuf	Constructs a **strstreambuf** object.
~strstreambuf	Destroys a **strstreambuf** object.

Other Functions

freeze	Freezes a stream.
str	Returns a pointer to the string.

Member Functions

strstreambuf::freeze

Syntax

void freeze(int *n* **= 1);**

Parameters

n

A 0 value permits automatic deletion of the current array and also its automatic growth (if it is dynamic); a nonzero value prevents deletion.

Remarks

If a **strstreambuf** object has a dynamic array, then memory is usually deleted on destruction and size adjustment. The **freeze** function provides a means of preventing that automatic deletion. Once an array is frozen, no further input or output is permitted. The results of such operations are undefined.

The **freeze** function can also unfreeze a frozen buffer.

See Also

strstreambuf::str

strstreambuf::str

Syntax

char* str();

Remarks

Returns a pointer to the object's internal character array. If the **strstreambuf** object was constructed with a user-supplied buffer, then that buffer address is returned. If the object has a dynamic array, then **str** freezes the array. You must not send characters to a frozen **strstreambuf** object, and you are responsible for deleting the array. If a dynamic array is empty, then **str** returns **NULL**.

You can use the **freeze** function with a 0 parameter to unfreeze a frozen **strstreambuf** object.

See Also

strstreambuf::freeze

strstreambuf::strstreambuf

Syntax

strstreambuf();

strstreambuf(int *nBytes* **);**

strstreambuf(char* *pch*, **int** *n*, **char*** *pstart* = **0);**

strstreambuf(unsigned char* *puch*, **int** *n*, **unsigned char*** *pustart* = **0);**

strstreambuf(signed char* *psch*, **int** *n*, **signed char*** *psstart* = **0);**

strstreambuf(void* (**falloc***)(long), void (****ffree***)(void*));**

Parameters

nBytes
The initial length of a dynamic stream buffer.

pch, puch, psch
A pointer to a character buffer that will be attached to the object. The get pointer is initialized to this value.

n
An integer parameter with the following meanings:

Value	Meaning
positive	*n* bytes, starting at *pch*, is used as a fixed-length stream buffer.
0	The *pch* parameter points to the start of a null-terminated string that constitutes the stream buffer (terminator excluded).
negative	The *pch* parameter points to a stream buffer that continues indefinitely.

pstart, pustart, psstart
The initial value of the put pointer.

falloc
A memory-allocation function with the prototype **void* falloc(long)**. The default is **new**.

ffree
A function that frees allocated memory with the prototype **void ffree(void*)**. The default is **delete**.

Remarks

The four **streambuf** constructors are described as follows:

Constructor	Description
strstreambuf()	Constructs an empty **strstreambuf** object with dynamic buffering. The buffer is allocated internally by the class and grows as needed, unless it is frozen.
strstreambuf(int)	Constructs an empty **strstreambuf** object with a dynamic buffer *n* bytes long to start with. The buffer is allocated internally by the class and grows as needed, unless it is frozen.
strstreambuf(char*, int, char*)	Constructs a **strstreambuf** object from already-allocated memory as specified by the arguments. There are constructor variations for both unsigned and signed character arrays.
strstreambuf(void*(*), void(*))	Constructs an empty **strstreambuf** object with dynamic buffering. The *falloc* function is called for allocation. The **long** parameter specifies the buffer length and the function returns the buffer address. If the *falloc* pointer is **NULL**, then operator **new** is used. The *ffree* function frees memory allocated by *falloc*. If the *ffree* pointer is **NULL**, the operator **delete** is used.

strstreambuf::~strstreambuf

Syntax

~strstreambuf();

Remarks

Destroys a **strstreambuf** object and, in the process, releases all internally allocated dynamic memory unless the object is frozen. The destructor does not release user-allocated memory.

Index

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®

1191 Part No. 24776