

Unit – I

Windows Programming

1. What is Windows Programming?
2. What is an API?
3. Write the ways of implementing API?
4. What are the uses of kernel, user and GDI Libraries?
5. Name any platform independent API?
6. What are the steps required to create a window?
7. How to register a window class?
8. What is a window class?
9. How to create a window?
10. Define Message Loop?
11. What is the use of GetMessage?
12. What is the use of TranslateMessage?
13. What is the use of Dispatch Message?
14. What are the actions of windows procedure?
15. What is the value returned from GetMessage?
16. What is WM_PAINT message?
17. How does a client area become invalid?
18. WM_PAINT message is processed?
19. What is WM_DESTROY message?
20. How the program is terminated?
21. What is GDI?
22. Define Scroll bar and what are the message generated?
23. What is a keyboard accelerator? What are the messages generated?
24. What are the types of messages generated by a mouse?
25. What is a menu? What are its types?

Part B(from charles petzold)

1. Creation of Window.(page 57)
2. Implement Scroll bar control(page 97)
3. explain GDI concepts (page 121)
4. How to draw rectangle and clipping(page 196)
5. Mouse movement programs (page 273)
6. Child window control like list box etc(page 401)

Unit – II and III

1. What is message processing?
2. Give some messages used in windows?
3. What is GDI?
4. What are dynamic link libraries?
5. What is an application framework?
6. Mention some of the program elements?
7. Explain the document-view architecture?
8. What is a view?
9. Difference between Single Document Interface and Multiple Document Interface?
10. Mention some window resources?

11. Explain message map?
12. What is Window's Client area?
13. What is Cscrollview class?
14. Explain about accepting keyboard input?
15. Explain WM_CREATE and WM_CLOSE message?
16. Explain the display context classes CClientDC and CWindowDC?
17. What is the state of the device context?
18. What are GDI objects?
19. Mention some of the GDI derived classes?
20. Difference between modal and modeless dialogs?
21. What is system modal dialog?
22. List some of the COMDLG32 classes?
23. What are dialog controls?
24. Mention some of the windows common control?
25. Explain WM_NOTIFY Message?

Part B(refer David kruglinski)

1. Explain VC++ components.(page6)
 2. Explain MFC (to explain SDI, MDI, and Dialog Based)(page 31)
 3. Write a program to draw a sine wave without using MFC(refer windows programming)
 4. Write a VC++ program to implement property sheets.
 5. Write a VC++ program to add images to the tree list view control.
 6. Menu, Popup menu and keyboard accelerators.
 7. Colors and Fonts.
 8. Explain Toolbar and Status bar
 9. Explain Modal and Modeless dialog.(page 103).
 10. Explain Windows common dialog Control.(page 147)
 11. Write a VC++ program to show the uses of slider control, spin button control and tool tip control.
 12. Write a program to draw a text in a window.
 13. Implement DLL concepts
 14. Explain Class Wizard and application wizard.
 15. Write a VC++ program to create Combo box and push button.
 16. Explain about Serialization (Document View Architecture).

Unit – IV & V

- (1) What is DLL?
- (2) What are the advantages of using DLL?
- (3) What are the disadvantages of using DLL?
- (4) What do you mean by Resource DLL?
- (5) How to create own DLLs?
- (6) What is DLLMain ()?
- (7) Write short notes on the memory issues of DLL.
- (8) What is Load Library () function?
- (9) What is Free Library () function?
- (10) What is GetProcAddress() function?
- (11) Define Object Model.
- (12) What is DCOM?

- (13) What are main features of COM?
- (14) What are the methods of Binary Interface method?
- (15) Write short notes on Threading Model.
- (16) Write short notes on Versioning.
- (17) What is MTS?
- (18) Define OLE.
- (19) What are the applications of OLE?
- (20) What is RFX and where it is used?
- (21) What is the use of CRecordSet :: dynamic?
- (22) What are the steps to create sample database?
- (23) Write the general syntax to create a table. Give one example.
- (24) How to use Single Row Functions?
- (25) What are Data transfer and its two structures?
- (26) What is the use of CFile Dialog Class?
- (27) Which header file contains Ctoolbar class?
- (28) Explain ISAPI server?
- (29)

Part B

- 1. Explain ActiveX control through VC++ program.
- 2. Explain about COM and what way it is different with CORBA.
- 3. Explain OLE control.
- 4. Explain Space ship program.
- 5. Explain ODBC architecture with MFC ODBC classes.
- 6. Explain DAO concepts through VC++ program.
- 7. Employee or Student Database creation.
- 8. Explain how to implement Sound and Video files.

1. Explain about the History of Windows Programming Environment?

A History of Windows

Soon after the introduction of the IBM PC in the fall of 1981, it became evident that the predominant operating system for the PC (and compatibles) would be MS-DOS, which originally stood for Microsoft Disk Operating System. MS-DOS was a minimal operating system. For the user, MS-DOS provided a command-line interface to commands such as DIR and TYPE and loaded application programs into memory for execution. For the application programmer, MS-DOS offered little more than a set of function calls for doing file input/output (I/O). For other tasks—in particular, writing text and sometimes graphics to the video display—applications accessed the hardware of the PC directly.

Due to memory and hardware constraints, sophisticated graphical environments were slow in coming to small computers. Apple Computer offered an alternative to character-mode environments when it released its ill-fated Lisa in January 1983, and then set a standard for graphical environments with the Macintosh in January 1984. Despite the Mac's declining market share, it is still considered the standard against which other graphical environments are measured. All graphical environments, including the Macintosh and Windows, are indebted to the pioneering work done at the Xerox Palo Alto Research Center (PARC) beginning in the mid-1970s.

Windows was announced by Microsoft Corporation in November 1983 (post-Lisa but pre-Macintosh) and was released two years later in November 1985. Over the next two years, Microsoft Windows 1.0 was followed by several updates to support the international market and to provide drivers for additional video displays and printers.

Windows 2.0 was released in November 1987. This version incorporated several changes to the user interface. The most significant of these changes involved the use of overlapping windows rather than the "tiled" windows found in Windows 1.0. Windows 2.0 also included enhancements to the keyboard and mouse interface, particularly for menus and dialog boxes.

Up until this time, Windows required only an Intel 8086 or 8088 microprocessor running in "real mode" to access 1 megabyte (MB) of memory. Windows/386 (released shortly after Windows 2.0) used the "virtual 86" mode of the Intel 386 microprocessor to window and multitask many DOS programs that directly accessed hardware. For symmetry, Windows 2.1 was renamed Windows/286.

Windows 3.0 was introduced on May 22, 1990. The earlier Windows/286 and Windows/386 versions were merged into one product with this release. The big change in Windows 3.0 was the support of the 16-bit protected-mode operation of Intel's 286, 386, and 486 microprocessors. This gave Windows and Windows applications access to up to 16 megabytes of memory. The Windows "shell" programs for running programs and maintaining files were completely revamped. Windows 3.0 was the first version of Windows to gain a foothold in the home and the office.

Any history of Windows must also include a mention of OS/2, an alternative to DOS and Windows that was originally developed by Microsoft in collaboration with IBM. OS/2 1.0 (character-mode only) ran on the Intel 286 (or later) microprocessors and was released in late 1987. The graphical Presentation Manager (PM) came about with OS/2 1.1 in October 1988. PM was originally supposed to be a protected-mode version of Windows, but the graphical API was changed to such a degree that it proved difficult for software manufacturers to support both platforms.

By September 1990, conflicts between IBM and Microsoft reached a peak and required that the two companies go their separate ways. IBM took over OS/2 and Microsoft made it clear that Windows was the center of their

strategy for operating systems. While OS/2 still has some fervent admirers, it has not nearly approached the popularity of Windows.

Microsoft Windows version 3.1 was released in April 1992. Several significant features included the TrueType font technology (which brought scaleable outline fonts to Windows), multimedia (sound and music), Object Linking and Embedding (OLE), and standardized common dialog boxes. Windows 3.1 ran *only* in protected mode and required a 286 or 386 processor with at least 1 MB of memory.

Windows NT, introduced in July 1993, was the first version of Windows to support the 32-bit mode of the Intel 386, 486, and Pentium microprocessors. Programs that run under Windows NT have access to a 32-bit flat address space and use a 32-bit instruction set. (I'll have more to say about address spaces a little later in this chapter.) Windows NT was also designed to be portable to non-Intel processors, and it runs on several RISC-based workstations.

Windows 95 was introduced in August 1995. Like Windows NT, Windows 95 also supported the 32-bit programming mode of the Intel 386 and later microprocessors. Although it lacked some of the features of Windows NT, such as high security and portability to RISC machines, Windows 95 had the advantage of requiring fewer hardware resources.

Windows 98 was released in June 1998 and has a number of enhancements, including performance improvements, better hardware support, and a closer integration with the Internet and the World Wide Web.

Programs running in Windows can share routines that are located in other files called "dynamic-link libraries." Windows includes a mechanism to link the program with the routines in the dynamic-link libraries at run time. Windows itself is basically a set of dynamic-link libraries.

Windows is a graphical interface, and Windows programs can make full use of graphics and formatted text on both the video display and the printer. A graphical interface not only is more attractive in appearance but also can impart a high level of information to the user.

Programs written for Windows do not directly access the hardware of graphics display devices such as the screen and printer. Instead, Windows includes a graphics programming language (called the Graphics Device Interface, or GDI) that allows the easy display of graphics and formatted text. Windows virtualizes display hardware. A program written for Windows will run with any video board or any printer for which a Windows device driver is available. The program does not need to determine what type of device is attached to the system.

Dynamic Linking

Central to the workings of Windows is a concept known as "dynamic linking." Windows provides a wealth of function calls that an application can take advantage of, mostly to implement its user interface and display text and graphics on the video display. These functions are implemented in dynamic-link libraries, or DLLs. These are files with the extension .DLL or sometimes .EXE, and they are mostly located in the \WINDOWS\SYSTEM subdirectory under Windows 98 and the \WINNT\SYSTEM and \WINNT\SYSTEM32 subdirectories under Windows NT.

In the early days, the great bulk of Windows was implemented in just three dynamic-link libraries. These represented the three main subsystems of Windows, which were referred to as Kernel, User, and GDI. While the number of subsystems has proliferated in recent versions of Windows, most function calls that a typical Windows program makes will still fall in one of these three modules. Kernel (which is currently implemented by the 16-bit KRNL386.EXE and the 32-bit KERNEL32.DLL) handles all the stuff that an operating system kernel traditionally handles—memory management, file I/O, and tasking. User (implemented in the 16-bit USER.EXE and the 32-bit USER32.DLL) refers to the user interface, and implements all the windowing logic. GDI

(implemented in the 16-bit GDI.EXE and the 32-bit GDI32.DLL) is the Graphics Device Interface, which allows a program to display text and graphics on the screen and printer.

Windows 98 supports several thousand function calls that applications can use. Each function has a descriptive name, such as *CreateWindow*. This function (as you might guess) creates a window for your program. All the Windows functions that an application may use are declared in header files.

In your Windows program, you use the Windows function calls in generally the same way you use C library functions such as *strlen*. The primary difference is that the machine code for C library functions is linked into your program code, whereas the code for Windows functions is located outside of your program in the DLLs.

When you run a Windows program, it interfaces to Windows through a process called "dynamic linking." A Windows .EXE file contains references to the various dynamic-link libraries it uses and the functions therein. When a Windows program is loaded into memory, the calls in the program are resolved to point to the entries of the DLL functions, which are also loaded into memory if not already there.

When you link a Windows program to produce an executable file, you must link with special "import libraries" provided with your programming environment. These import libraries contain the dynamic-link library names and reference information for all the Windows function calls. The linker uses this information to construct the table in the .EXE file that Windows uses to resolve calls to Windows functions when loading the program.

2. Explain the basics simple windows program and what are the windows header files

The Windows Equivalent

```
/*-----  
HelloMsg.c -- Displays "Hello, Windows 98!" in a message box  
            (c) Charles Petzold, 1998  
-----*/  
  
#include <windows.h>  
  
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                   PSTR szCmdLine, int iCmdShow)  
{  
    MessageBox (NULL, TEXT ("Hello, Windows 98!"), TEXT ("HelloMsg"), 0) ;  
  
    return 0 ;  
}
```

To begin, select New from the File menu. In the New dialog box, pick the Projects tab. Select Win32 Application. In the Location field, select a subdirectory. In the Project Name field, type the name of the project, which in this case is HelloMsg. This will be a subdirectory of the directory indicated in the Location field. The Create New Workspace button should be checked. The Platforms section should indicate Win32. Choose OK.

A dialog box labeled Win32 Application - Step 1 Of 1 will appear. Indicate that you want to create an Empty Project, and press the Finish button.

Select New from the File menu again. In the New dialog box, pick the Files tab. Select C++ Source File. The Add To Project box should be checked, and HelloMsg should be indicated. Type HelloMsg.c in the File Name field. Choose OK.

Now you can type in the HELLOMSG.C file shown above. Or you can select the Insert menu and the File As Text option to copy the contents of HELLOMSG.C from the file on this book's companion CD-ROM.

Structurally, HELLOMSG.C is identical to the K&R "hello, world" program. The header file STDIO.H has been replaced with WINDOWS.H, the entry point *main* has been replaced with *WinMain*, and the C run-time library function *printf* has been replaced with the Windows API function *MessageBox*. However, there is much in the program that is new, including several strange-looking uppercase identifiers.

Let's start at the top.

The Header Files

HELLOMSG.C begins with a preprocessor directive that you'll find at the top of virtually every Windows program written in C:

```
#include <windows.h>
```

WINDOWS.H is a master include file that includes other Windows header files, some of which also include other header files. The most important and most basic of these header files are:

- *WINDEF.H* Basic type definitions.
- *WINNT.H* Type definitions for Unicode support.
- *WINBASE.H* Kernel functions.
- *WINUSER.H* User interface functions.
- *WINGDI.H* Graphics device interface functions.

These header files define all the Windows data types, function calls, data structures, and constant identifiers.

Program Entry Point

Just as the entry point to a C program is the function *main*, the entry point to a Windows program is *WinMain*, which always appears like this:

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                    PSTR szCmdLine, int iCmdShow)
```

This entry point is documented in */Platform SDK/User Interface Services/Windowing/Windows/Window Reference/Window Functions*. It is declared in WINBASE.H like so (line breaks and all):

```
int  
WINAPI  
WinMain(  
    HINSTANCE hInstance,  
    HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine,  
    int nShowCmd  
);
```

You'll notice I've made a couple of minor changes in HELLOMSG.C. The third parameter is defined as an LPSTR in WINBASE.H, and I've made it a PSTR. These two data types are both defined in WINNT.H as pointers to character strings. The LP prefix stands for "long pointer" and is an artifact of 16-bit Windows.

I've also changed two of the parameter names from the *WinMain* declaration; many Windows programs use a system called "**Hungarian notation**" for naming variables. This system involves prefacing the variable name with a short prefix that indicates the variable's data type. just keep in mind that the prefix *i* stands for *int* and *sz* stands for "string terminated with a zero."

The *WinMain* function is declared as returning an *int*. The WINAPI identifier is defined in WINDEF.H with the statement:

```
#define WINAPI __stdcall
```

This statement specifies a calling convention that involves how machine code is generated to place function call arguments on the stack. Most Windows function calls are declared as WINAPI.

The first parameter to *WinMain* is something called an "instance handle." In Windows programming, a handle is simply a number that an application uses to identify something. In this case, the handle uniquely identifies the program. It is required as an argument to some other Windows function calls. In early versions of Windows, when you ran the same program concurrently more than once, you created *multiple instances* of that program. All instances of the same application shared code and read-only memory (usually resources such as menu and dialog box templates). A program could determine if other instances of itself were running by checking the *hPrevInstance* parameter. It could then skip certain chores and move some data from the previous instance into its own data area.

In the 32-bit versions of Windows, this concept has been abandoned. The second parameter to *WinMain* is always NULL (defined as 0).

The third parameter to *WinMain* is the command line used to run the program. Some Windows applications use this to load a file into memory when the program is started. The fourth parameter to *WinMain* indicates how the program should be initially displayed—either normally or maximized to fill the window, or minimized to be displayed in the task list bar.

3. Write a windows program to create a window.

HELLOWIN.C -- Displays "Hello, Windows 98!" in client area
(c) Charles Petzold, 1998

```
-----*/  
  
#include <windows.h>  
  
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;  
  
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                    PSTR szCmdLine, int iCmdShow)  
{  
    static TCHAR szAppName[] = TEXT ("HelloWin") ;  
    HWND      hwnd ;  
    MSG       msg ;  
    WNDCLASS  wndclass ;  
  
    wndclass.style      = CS_HREDRAW | CS_VREDRAW ;  
    wndclass.lpfnWndProc = WndProc ;
```



```
wndclass.cbClsExtra = 0 ;
wndclass.cbWndExtra = 0 ;
wndclass.hInstance = hInstance ;
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
wndclass.lpszMenuName = NULL ;
wndclass.lpszClassName = szAppName ;

if (!RegisterClass (&wndclass))
{
    MessageBox (NULL, TEXT ("This program requires Windows NT!"),
        szAppName, MB_ICONERROR) ;
    return 0 ;
}
hwnd = CreateWindow (szAppName,           // window class name
    TEXT ("The Hello Program"), // window caption
    WS_OVERLAPPEDWINDOW,           // window style
    CW_USEDEFAULT,                 // initial x position
    CW_USEDEFAULT,                 // initial y position
    CW_USEDEFAULT,                 // initial x size
    CW_USEDEFAULT,                 // initial y size
    NULL,                          // parent window handle
    NULL,                          // window menu handle
    hInstance,                     // program instance handle
    NULL) ;                        // creation parameters

ShowWindow (hwnd, iCmdShow) ;
UpdateWindow (hwnd) ;

while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
}
return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC      hdc ;
    PAINTSTRUCT ps ;
    RECT      rect ;

    switch (message)
    {
    case WM_CREATE:
        PlaySound (TEXT ("hellowin.wav"), NULL, SND_FILENAME | SND_ASYNC) ;
        return 0 ;

    case WM_PAINT:
        hdc = BeginPaint (hwnd, &ps) ;
```

```
GetClientRect (hwnd, &rect) ;

DrawText (hdc, TEXT ("Hello, Windows 98!"), -1, &rect,
          DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
EndPaint (hwnd, &ps) ;
return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

This program creates a normal application window, as shown in Figure 3-2, and displays, "Hello, Windows 98!" in the center of that window. If you have a sound board installed, you will also hear me saying the same thing.

3. List out the various Windows Function Call.

The Windows Function Calls

HELLOWIN makes calls to no fewer than 18 Windows functions. In the order they occur, these functions (with a brief description) are:

- *LoadIcon* Loads an icon for use by a program.
- *LoadCursor* Loads a mouse cursor for use by a program.
- *GetStockObject* Obtains a graphic object, in this case a brush used for painting the window's background.
- *RegisterClass* Registers a window class for the program's window.
- *MessageBox* Displays a message box.
- *CreateWindow* Creates a window based on a window class.
- *ShowWindow* Shows the window on the screen.
- *UpdateWindow* Directs the window to paint itself.
- *GetMessage* Obtains a message from the message queue.
- *TranslateMessage* Translates some keyboard messages.
- *DispatchMessage* Sends a message to a window procedure.
- *PlaySound* Plays a sound file.
- *BeginPaint* Initiates the beginning of window painting.
- *GetClientRect* Obtains the dimensions of the window's client area.
- *DrawText* Displays a text string.
- *EndPaint* Ends window painting.
- *PostQuitMessage* Inserts a "quit" message into the message queue.
- *DefWindowProc* Performs default processing of messages.

These functions are described in the Platform SDK documentation, and they are declared in various header files, mostly in WINUSER.H.

Prefix

Constant

CS

Class style option

CW	Create window option
DT	Draw text option
IDI	ID number for an icon
IDC	ID number for a cursor
MB	Message box options
SND	Sound option
WM	Window message
WS	Window style

You almost never need to remember numeric constants when programming for Windows. Virtually every numeric constant has an identifier defined in the header files.

HELLOWIN also uses four data structures (which I'll discuss later in this chapter) defined in the Windows header files. These data structures are shown in the table below.

<i>Structure</i>	<i>Meaning</i>
MSG	Message structure
WNDCLASS	Window class structure
PAINTSTRUCT	Paint structure
RECT	Rectangle structure

The first two data structures are used in *WinMain* to define two structures named *msg* and *wndclass*. The second two are used in *WndProc* to define two structures named *ps* and *rect*.

Getting a Handle on Handles

Finally, there are three uppercase identifiers for various types of "handles":

<i>Identifier</i>	<i>Meaning</i>
HINSTANCE	Handle to an "instance"—the program itself
HWND	Handle to a window
HDC	Handle to a device context

Handles are used quite frequently in Windows. Before the chapter is over, you will also encounter HICON (a handle to an icon), HCURSOR (a handle to a mouse cursor), and HBRUSH (a handle to a graphics brush).

A handle is simply a number (usually 32 bits in size) that refers to an object. The handles in Windows are similar to file handles used in conventional C or MS-DOS programming. A program almost always obtains a handle by calling a Windows function. The program uses the handle in other Windows functions to refer to the object. The

actual value of the handle is unimportant to your program, but the Windows module that gives your program the handle knows how to use it to reference the object.

The variable name prefixes I'll generally be using in this book are shown in the following table.

<i>Prefix</i>	<i>Data Type</i>
<i>c</i>	char or WCHAR or TCHAR
<i>by</i>	BYTE (unsigned char)
<i>n</i>	Short
<i>i</i>	Int
<i>x, y</i>	int used as x-coordinate or y-coordinate
<i>cx, cy</i>	int used as x or y length; c stands for "count"
<i>b</i> or <i>f</i>	BOOL (int); f stands for "flag"
<i>w</i>	WORD (unsigned short)
<i>l</i>	LONG (long)
<i>dw</i>	DWORD (unsigned long)
<i>fn</i>	Function
<i>s</i>	String
<i>sz</i>	string terminated by 0 character
<i>h</i>	Handle
<i>p</i>	Pointer

Registering the Window Class

A window is always created based on a window class. The window class identifies the window procedure that processes messages to the window.

Before you create an application window, you must register a window class by calling *RegisterClass*. This function requires a single parameter, which is a pointer to a structure of type WNDCLASS. This structure includes two fields that are pointers to character strings, so the structure is defined two different ways in the WINUSER.H header file. First, there's the ASCII version, WNDCLASSA:

```
typedef struct tagWNDCLASSA
{
    UINT    style ;
    WNDPROC lpfnWndProc ;
    int     cbClsExtra ;
    int     cbWndExtra ;
    HINSTANCE hInstance ;
    HICON    hIcon ;
    HCURSOR  hCursor ;
```

```
HBRUSH    hbrBackground ;  
LPCSTR    lpzMenuName ;  
LPCSTR    lpzClassName ;  
}  
WNDCLASSA, * PWNDCLASSA, NEAR * NPWNDCLASSA, FAR * LPWNDCLASSA ;
```

Notice some uses of Hungarian notation here: The *lpfn* prefix means "long pointer to a function." (Recall that in the Win32 API there is no distinction between long pointers and near pointers. This is a remnant of 16-bit Windows.) The *cb* prefix stands for "count of bytes" and is often used for a variable that denotes a byte size. The *h* prefix is a handle, and the *hbr* prefix means "handle to a brush." The *lpz* prefix is a "long pointer to a string terminated with a zero."

4. Explain about Painting and Repainting

Windows is a message-driven system. Windows informs applications of various events by posting messages in the application's message queue or sending messages to the appropriate window procedure. Windows informs a window procedure that part of the window's client area needs painting by posting a WM_PAINT message.

The WM_PAINT Message

Most Windows programs call the function *UpdateWindow* during initialization in *WinMain* shortly before entering the message loop. Windows takes this opportunity to send the window procedure its first WM_PAINT message. This message informs the window procedure that the client area must be painted. Thereafter, that window procedure should be ready at almost any time to process additional WM_PAINT messages and even to repaint the entire client area of the window if necessary. A window procedure receives a WM_PAINT message whenever one of the following events occurs:

- A previously hidden area of the window is brought into view when a user moves a window or uncovers a window.
- The user resizes the window (if the window class style has the CS_HREDRAW and CW_VREDRAW bits set).
- The program uses the *ScrollWindow* or *ScrollDC* function to scroll part of its client area.
- The program uses the *InvalidateRect* or *InvalidateRgn* function to explicitly generate a WM_PAINT message.

In some cases when part of the client area is temporarily written over, Windows attempts to save an area of the display and restore it later. This is not always successful. Windows may sometimes post a WM_PAINT message when:

- Windows removes a dialog box or message box that was overlaying part of the window.
- A menu is pulled down and then released.
- A tool tip is displayed.

In a few cases, Windows always saves the area of the display it overwrites and then restores it. This is the case whenever:

- The mouse cursor is moved across the client area.
- An icon is dragged across the client area.

Dealing with WM_PAINT message requires that you alter the way you think about how you write to the video display. Your program should be structured so that it accumulates all the information necessary to paint the client area but paints only "on demand"—when Windows sends the window procedure a WM_PAINT message. If your

program needs to update its client area at some other time, it can force Windows to generate this WM_PAINT message. This may seem a roundabout method of displaying something on the screen, but the structure of your program will benefit from it.

Valid and Invalid Rectangles

Although a window procedure should be prepared to update the entire client area whenever it receives a WM_PAINT message, it often needs to update only a smaller area, most often a rectangular area within the client area. This is most obvious when a dialog box overlies part of the client area. Repainting is required only for the rectangular area uncovered when the dialog box is removed.

That area is known as an "invalid region" or "update region." The presence of an invalid region in a client area is what prompts Windows to place a WM_PAINT message in the application's message queue. Your window procedure receives a WM_PAINT message only if part of your client area is invalid.

Windows internally maintains a "paint information structure" for each window. This structure contains, among other information, the coordinates of the smallest rectangle that encompasses the invalid region. This is known as the "invalid rectangle." If another region of the client area becomes invalid before the window procedure processes a pending WM_PAINT message, Windows calculates a new invalid region (and a new invalid rectangle) that encompasses both areas and stores this updated information in the paint information structure. Windows does not place multiple WM_PAINT messages in the message queue.

A window procedure can invalidate a rectangle in its own client area by calling *InvalidateRect*. If the message queue already contains a WM_PAINT message, Windows calculates a new invalid rectangle. Otherwise, it places a WM_PAINT message in the message queue. A window procedure can obtain the coordinates of the invalid rectangle when it receives a WM_PAINT message (as we'll see later in this chapter). It can also obtain these coordinates at any other time by calling *GetUpdateRect*.

After the window procedure calls *BeginPaint* during the WM_PAINT message, the entire client area is validated. A program can also validate any rectangular area within the client area by calling the *ValidateRect* function. If this call has the effect of validating the entire invalid area, then any WM_PAINT message currently in the queue is removed.

5. Explain about paint information structure

The Paint Information Structure

Earlier I mentioned a "paint information structure" that Windows maintains for each window. That's what PAINTSTRUCT is. The structure is defined as follows:

```
typedef struct tagPAINTSTRUCT
{
    HDC      hdc ;
    BOOL     fErase ;
    RECT     rcPaint ;
    BOOL     fRestore ;
    BOOL     fIncUpdate ;
    BYTE     rgbReserved[32] ;
} PAINTSTRUCT ;
```

Windows fills in the fields of this structure when your program calls *BeginPaint*. Your program can use only the first three fields. The others are used internally by Windows. The *hdc* field is the handle to the device context. In a

redundancy typical of Windows, the value returned from *BeginPaint* is also this device context handle. In most cases, *fErase* will be flagged FALSE (0), meaning that Windows has already erased the background of the invalid rectangle. This happens earlier in the *BeginPaint* function. (If you want to do some customized background erasing in your window procedure, you can process the WM_ERASEBKGND message.) Windows erases the background using the brush specified in the *hbrBackground* field of the WNDCLASS structure that you use when registering the window class during *WinMain* initialization. Many Windows programs specify a white brush for the window background. This is indicated when the program sets up the fields of the window class structure with a statement like this:

```
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

However, if your program invalidates a rectangle of the client area by calling *InvalidateRect*, the last argument of the function specifies whether you want the background erased. If this argument is FALSE (that is, 0), Windows will not erase the background and the *fErase* field of the PAINTSTRUCT structure will be TRUE (nonzero) after you call *BeginPaint*.

The *rcPaint* field of the PAINTSTRUCT structure is a structure of type RECT., the RECT structure defines a rectangle with four fields named *left*, *top*, *right*, and *bottom*. The *rcPaint* field in the PAINTSTRUCT structure defines the boundaries of the invalid rectangle, as shown in Figure 4-1. The values are in units of pixels relative to the upper left corner of the client area. The invalid rectangle is the area that you should repaint.

The *rcPaint* rectangle in PAINTSTRUCT is not only the invalid rectangle; it is also a "clipping" rectangle. This means that Windows restricts painting to within the clipping rectangle. More precisely, if the invalid region is not rectangular, Windows restricts painting to within that region.

To paint outside the update rectangle while processing WM_PAINT messages, you can make this call:

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

before calling *BeginPaint*. This invalidates the entire client area and causes *BeginPaint* to erase the background. A FALSE value in the last argument will not erase the background. Whatever was there will stay.

It is usually most convenient for a Windows program to simply repaint the entire client area whenever it receives a WM_PAINT message, regardless of the *rcPaint* structure. For example, if part of the display output in the client area includes a circle but only part of the circle falls within the invalid rectangle, it makes little sense to draw only the invalid part of the circle. Draw the whole circle. When you use the device context handle returned from *BeginPaint*, Windows will not paint outside the *rcPaint* rectangle anyway.

6.Explain about Text out function and system metric.

TextOut: The Details

TextOut is the most common GDI function for displaying text. Its syntax is

```
TextOut (hdc, x, y, psText, iLength) ;
```

Let's examine this function in more detail.

The first argument is the handle to the device context—either the *hdc* value returned from *GetDC* or the *hdc* value returned from *BeginPaint* during processing of a WM_PAINT message.

The attributes of the device context control the characteristics of this displayed text. For instance, one attribute of the device context specifies the text color. The default color (we discover with some degree of comfort) is black. The default device context also defines a text background color, and this is white. When a program writes text to the display, Windows uses this background color to fill in the rectangular space surrounding each character, called the "character box."

The text background color is not the same background you set when defining the window class. The background in the window class is a brush—which is a pattern that may or may not be a pure color—that Windows uses to erase the client area. It is not part of the device context structure. When defining the window class structure, most Windows applications use `WHITE_BRUSH` so that the default text background color in the default device context is the same color as the brush Windows uses to erase the background of the client area.

The *psText* argument is a pointer to a character string, and *iLength* is the number of characters in the string. If *psText* points to a Unicode character string, then the number of bytes in the string is double the *iLength* value. The string should not contain any ASCII control characters such as carriage returns, linefeeds, tabs, or backspaces. Windows displays these control characters as boxes or solid blocks. *TextOut* does not recognize a zero byte (or for Unicode, a zero short integer) as denoting the end of a string. The function uses the *iLength* argument to determine the string's length.

The *x* and *y* arguments to *TextOut* define the starting point of the character string within the client area. The *x* value is the horizontal position; the *y* value is the vertical position. The upper left corner of the first character is positioned at the coordinate point (*x*, *y*). In the default device context, the origin (that is, the point where *x* and *y* both equal 0) is the upper left corner of the client area. If you use zero values for *x* and *y* in *TextOut*, the character string starts flush against the upper left corner of the client area.

Just as a program can determine information about the sizes (or "metrics") of user interface items by calling the *GetSystemMetrics* function, a program can determine font sizes by calling *GetTextMetrics*. *GetTextMetrics* requires a handle to a device context because it returns information about the font currently selected in the device context. Windows copies the various values of text metrics into a structure of type `TEXTMETRIC` defined in `WINGDI.H`. The `TEXTMETRIC` structure has 20 fields, but we're interested in only the first seven:

```
typedef struct tagTEXTMETRIC
{
    LONG tmHeight ;
    LONG tmAscent ;
    LONG tmDescent ;
    LONG tmInternalLeading ;
    LONG tmExternalLeading ;
    LONG tmAveCharWidth ;
    LONG tmMaxCharWidth ;
    [other structure fields]
}
TEXTMETRIC, * PTEXTMETRIC ;
```

The values of these fields are in units that depend on the mapping mode currently selected for the device context. In the default device context, this mapping mode is `MM_TEXT`, so the dimensions are in units of pixels.

To use the *GetTextMetrics* function, you first need to define a structure variable, commonly called *tm*:

```
TEXTMETRIC tm ;
```

When you need to determine the text metrics, you get a handle to a device context and call *GetTextMetrics*:


```
hdc = GetDC (hwnd) ;
GetTextMetrics (hdc, &tm) ;
ReleaseDC (hwnd, hdc) ;
```

You can then examine the values in the text metric structure and probably save a few of them for future use.

Text Metrics: The Details

The TEXTMETRIC structure provides various types of information about the font currently selected in the device context. However, the vertical size of a font is defined by only five fields of the structure, four of which are shown in Figure 4-3.

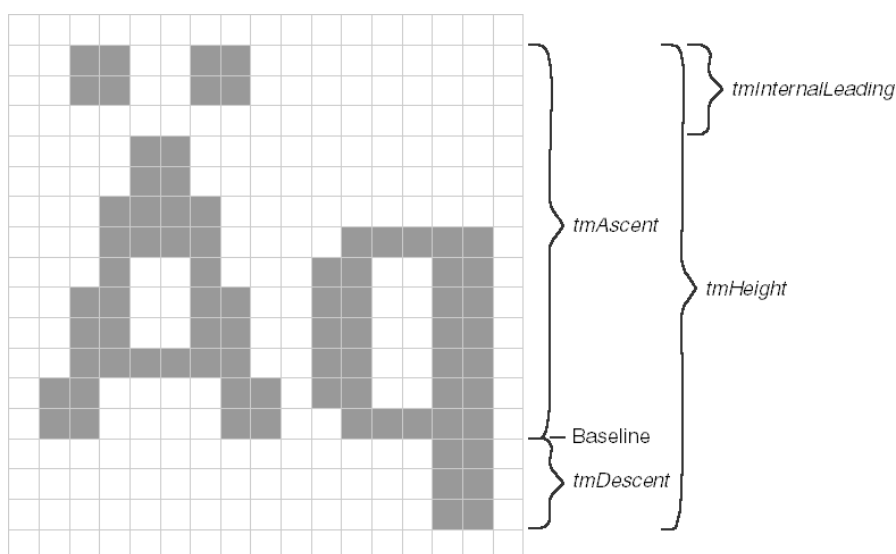


Figure 4-3. Four values defining vertical character sizes in a font.

The most important value is *tmHeight*, which is the sum of *tmAscent* and *tmDescent*. These two values represent the maximum vertical extents of characters in the font above and below the baseline. The term "leading" refers to space that a printer inserts between lines of text. In the TEXTMETRIC structure, internal leading is included in *tmAscent* (and thus in *tmHeight*) and is often the space in which accent marks appear. The *tmInternalLeading* field could be set to 0, in which case accented letters are made a little shorter so that the accent marks fit within the ascent of the character.

The TEXTMETRIC structure also includes a field named *tmExternalLeading*, which is not included in the *tmHeight* value. This is an amount of space that the designer of the font suggests be added between successive rows of displayed text. You can accept or reject the font designer's suggestion for including external leading when spacing lines of text. In the system fonts that I've encountered recently, *tmExternalLeading* has been zero, which is why I didn't include it in Figure 4-3. (Despite my vow not to tell you the dimensions of a system font, Figure 4-3 is accurate for the system font that Windows uses by default for a 640 by 480 display.)

The TEXTMETRIC structure contains two fields that describe character widths: the *tmAveCharWidth* field is a weighted average of lowercase characters, and *tmMaxCharWidth* is the width of the widest character in the font. For a fixed-pitch font, these values are the same. (For the font illustrated in Figure 4-3, these values are 7 and 14, respectively.)

The sample programs in this chapter will require another character width—the average width of uppercase letters. You can calculate this fairly accurately as 150% of *tmAveCharWidth*.

It's important to realize that the dimensions of a system font are dependent on the pixel size of the video display on which Windows runs and, in some cases, on the system font size the user has selected. Windows provides a device-independent graphics interface, but you have to help. Don't write your Windows programs so that they guess at character dimensions. Don't hard-code any values. Use the *GetTextMetrics* function to obtain this information.

7. Write a windows program to draw a scroll bar.

```
#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("SysMets2") ;
    HWND        hwnd ;
    MSG         msg ;
    WNDCLASS    wndclass ;

    wndclass.style      = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra  = 0 ;
    wndclass.cbWndExtra  = 0 ;
    wndclass.hInstance   = hInstance ;
    wndclass.hIcon       = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor     = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 2"),
                        WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
```

```
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}
```

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

```
{
    static int  cxChar, cxCaps, cyChar, cyClient, iVscrollPos ;
    HDC        hdc ;
    int        i, y ;
    PAINTSTRUCT ps ;
    TCHAR      szBuffer[10] ;
    TEXTMETRIC tm ;

    switch (message)
    {
    case WM_CREATE:
        hdc = GetDC (hwnd) ;

        GetTextMetrics (hdc, &tm) ;
        cxChar = tm.tmAveCharWidth ;
        cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
        cyChar = tm.tmHeight + tm.tmExternalLeading ;

        ReleaseDC (hwnd, hdc) ;

        SetScrollRange (hwnd, SB_VERT, 0, NUMLINES - 1, FALSE) ;
        SetScrollPos  (hwnd, SB_VERT, iVscrollPos, TRUE) ;
        return 0 ;

    case WM_SIZE:
        cyClient = HIWORD (lParam) ;
        return 0 ;

    case WM_VSCROLL:
        switch (LOWORD (wParam))
        {
        case SB_LINEUP:
            iVscrollPos -= 1 ;
            break ;

        case SB_LINEDOWN:
            iVscrollPos += 1 ;
            break ;

        case SB_PAGEUP:
            iVscrollPos -= cyClient / cyChar ;
            break ;

        case SB_PAGEDOWN:
            iVscrollPos += cyClient / cyChar ;
            break ;
        }
    }
}
```

```
case SB_THUMBPOSITION:
    iVscrollPos = HIWORD (wParam) ;
    break ;

default :
    break ;
}

iVscrollPos = max (0, min (iVscrollPos, NUMLINES - 1)) ;

if (iVscrollPos != GetScrollPos (hwnd, SB_VERT))
{
    SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
    InvalidateRect (hwnd, NULL, TRUE) ;
}
return 0 ;

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    for (i = 0 ; i < NUMLINES ; i++)
    {
        y = cyChar * (i - iVscrollPos) ;

        TextOut (hdc, 0, y,
            sysmetrics[i].szLabel,
            strlen (sysmetrics[i].szLabel)) ;

        TextOut (hdc, 22 * cxCaps, y,
            sysmetrics[i].szDesc,
            strlen (sysmetrics[i].szDesc)) ;

        SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
        TextOut (hdc, 22 * cxCaps + 40 * cxChar, y, szBuffer,
            sprintf (szBuffer, TEXT ("%5d"),
                GetSystemMetrics (sysmetrics[i].iIndex))) ;

        SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
    }
    EndPaint (hwnd, &ps) ;
    return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

8. Write a windows program to draw a sine wave

SINEWAVE.C

```
/*-----
```

```
SINEWAVE.C -- Sine Wave Using Polyline
```

```
(c) Charles Petzold, 1998
```

```
-----*/
```

```
#include <windows.h>
```

```
#include <math.h>
```

```
#define NUM 1000
```

```
#define TWOPI (2 * 3.14159)
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    PSTR szCmdLine, int iCmdShow)
```

```
{
```

```
    static TCHAR szAppName[] = TEXT ("SineWave") ;
```

```
    HWND      hwnd ;
```

```
    MSG       msg ;
```

```
    WNDCLASS  wndclass ;
```

```
    wndclass.style      = CS_HREDRAW | CS_VREDRAW ;
```

```
    wndclass.lpfnWndProc = WndProc ;
```

```
    wndclass.cbClsExtra  = 0 ;
```

```
    wndclass.cbWndExtra  = 0 ;
```

```
    wndclass.hInstance  = hInstance ;
```

```
    wndclass.hIcon       = LoadIcon (NULL, IDI_APPLICATION) ;
```

```
    wndclass.hCursor     = LoadCursor (NULL, IDC_ARROW) ;
```

```
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

```
    wndclass.lpszMenuName = NULL ;
```

```
    wndclass.lpszClassName = szAppName ;
```

```
    if (!RegisterClass (&wndclass))
```

```
    {
```

```
        MessageBox (NULL, TEXT ("Program requires Windows NT!"),  
            szAppName, MB_ICONERROR) ;
```

```
        return 0 ;
```

```
    }
```

```
    hwnd = CreateWindow (szAppName, TEXT ("Sine Wave Using Polyline"),  
        WS_OVERLAPPEDWINDOW,  
        CW_USEDEFAULT, CW_USEDEFAULT,  
        CW_USEDEFAULT, CW_USEDEFAULT,  
        NULL, NULL, hInstance, NULL) ;
```

```
    ShowWindow (hwnd, iCmdShow) ;
```

```
    UpdateWindow (hwnd) ;
```

```
    while (GetMessage (&msg, NULL, 0, 0))
```

```
    {
```

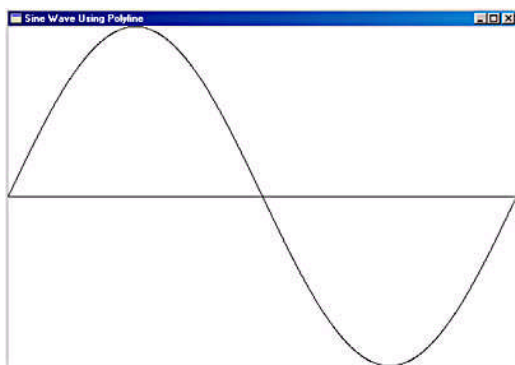
```
        TranslateMessage (&msg) ;
```

```
        DispatchMessage (&msg) ;
```

```
    }  
    return msg.wParam ;  
}
```

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

```
{  
    static int  cxClient, cyClient ;  
    HDC        hdc ;  
    int        i ;  
    PAINTSTRUCT ps ;  
    POINT      apt [NUM] ;  
  
    switch (message)  
    {  
    case WM_SIZE:  
        cxClient = LOWORD (lParam) ;  
        cyClient = HIWORD (lParam) ;  
        return 0 ;  
  
    case WM_PAINT:  
        hdc = BeginPaint (hwnd, &ps) ;  
  
        MoveToEx (hdc, 0,      cyClient / 2, NULL) ;  
        LineTo   (hdc, cxClient, cyClient / 2) ;  
  
        for (i = 0 ; i < NUM ; i++)  
        {  
            apt[i].x = i * cxClient / NUM ;  
            apt[i].y = (int) (cyClient / 2 * (1 - sin (TWOPI * i / NUM))) ;  
        }  
  
        Polyline (hdc, apt, NUM) ;  
        return 0 ;  
  
    case WM_DESTROY:  
        PostQuitMessage (0) ;  
        return 0 ;  
    }  
    return DefWindowProc (hwnd, message, wParam, lParam) ;  
}
```



9. Explain about Child window controls.

These programs display a grid of rectangles. When you click the mouse in a rectangle, the program draws an X. When you click again, the X disappears. Although the CHECKER1 and CHECKER2 versions of this program use only one main window, the CHECKER3 version uses a child window for each rectangle. The rectangles are maintained by a separate window procedure named *ChildProc*.

If we wanted to, we could add a facility to *ChildProc* to send a message to its parent window procedure (*WndProc*) whenever a rectangle is checked or unchecked. Here's how: The child window procedure can determine the window handle of its parent by calling *GetParent*,

```
hwndParent = GetParent (hwnd) ;
```

where *hwnd* is the window handle of the child window. It can then send a message to the parent window procedure:

```
SendMessage (hwndParent, message, wParam, lParam) ;
```

What would *message* be set to? Well, anything you want, really, as long as the numeric value is set to *WM_USER* or above. These numbers represent a range of messages that do not conflict with the predefined *WM_* messages. Perhaps for this message the child window could set *wParam* to its child window ID. The *lParam* could then be set to a 1 if the child window were being checked and a 0 if it were being unchecked.

This in effect creates a "child window control." The child window processes mouse and keyboard messages and notifies the parent window when the child window's state has changed. In this way, the child window becomes a high-level input device for the parent window. It encapsulates a specific functionality with regard to its graphical appearance on the screen, its response to user input, and its method of notifying another window when an important input event has occurred.

Child window controls are used most often in dialog boxes. As you'll see in [Chapter 11](#), the position and size of the child window controls are defined in a dialog box template contained in the program's resource script. However, you can also use predefined child window controls on the surface of a normal window's client area. You create each child window with a *CreateWindow* call and adjust the position and size of the child windows with calls to *MoveWindow*. The parent window procedure sends messages to the child window controls, and the child window controls send messages back to the parent window procedure.

The Windows programming documentation discusses child window controls in two places: First, the simple standard controls that you've seen in countless dialog boxes are described in */Platform SDK/User Interface Services/Controls*. These are buttons (including check boxes and radio buttons), static controls (such as text labels), edit boxes (which let you enter and edit lines or multiple lines of text), scroll bars, list boxes, and combo boxes. With the exception of the combo box, these controls have been around since Windows 1.0. This section of the Windows documentation also includes the rich edit control, which is similar to the edit box but allows editing formatted text with different fonts and such, and application desktop toolbars.

The Button Class

We'll begin our exploration of the button window class with a program named *BTNLOOK* ("button look"), which is shown in Figure 9-1. *BTNLOOK* creates 10 child window button controls, one for each of the 10 standard styles of buttons.

Creating the Child Windows

BTNLOOK defines a structure called *button* that contains button window styles and descriptive text strings for each of the 10 types of buttons. The button window styles all begin with the letters BS, which stand for "button style." The 10 button child windows are created in a *for* loop during WM_CREATE message processing in *WndProc*. The *CreateWindow* call uses the following parameters:

Class name	TEXT ("button")
Window text	button[i].szText
Window style	WS_CHILD WS_VISIBLE button[i].iStyle
x position	cxChar
y position	cyChar * (1 + 2 * i)
Width	20 * xChar
Height	7 * yChar / 4
Parent window	hwnd
Child window ID	(HMENU) i
Instance handle	((LPCREATESTRUCT) lParam) -> hInstance
Extra parameters	NULL

The class name parameter is the predefined name. The window style uses WS_CHILD, WS_VISIBLE, and one of the 10 button styles (BS_PUSHBUTTON, BS_DEFPUSHBUTTON, and so forth) that are defined in the button structure. The window text parameter (which for a normal window is the text that appears in the caption bar) is text that will be displayed with each button. I've simply used text that identifies the button style.

List Box Styles

The Listbox Class

The final predefined child window control I'll discuss in this chapter is the list box. A list box is a collection of text strings displayed as a scrollable columnar list within a rectangle. A program can add or remove strings in the list by sending messages to the list box window procedure. The list box control sends WM_COMMAND messages to its parent window when an item in the list is selected. The parent window can then determine which item has been selected.

A list box can be either single selection or multiple selection. The latter allows the user to select more than one item from the list box. When a list box has the input focus, it displays a dashed line surrounding an item in the list box. This cursor does not indicate the selected item in the list box. The selected item is indicated by highlighting, which displays the item in reverse video.

You create a list box child window control with *CreateWindow* using "listbox" as the window class and WS_CHILD as the window style. However, this default list box style does not send WM_COMMAND messages to its parent, meaning that a program would have to interrogate the list box (via messages to the list box controls) regarding the selection of items within the list box. Therefore, list box controls almost always include the list box style identifier LBS_NOTIFY, which allows the parent window to receive WM_COMMAND messages from the

list box. If you want the list box control to sort the items in the list box, you can also use LBS_SORT, another common style.

10. Explain about rectangle and clipping.

Rectangles, Regions, and Clipping

Windows includes several additional drawing functions that work with RECT (rectangle) structures and regions. A region is an area of the screen that is a combination of rectangles, polygons, and ellipses.

Working with Rectangles

These three drawing functions require a pointer to a rectangle structure:

```
FillRect (hdc, &rect, hBrush) ;  
FrameRect (hdc, &rect, hBrush) ;  
InvertRect (hdc, &rect) ;
```

In these functions, the *rect* parameter is a structure of type RECT with four fields: *left*, *top*, *right*, and *bottom*. The coordinates in this structure are treated as logical coordinates.

FillRect fills the rectangle (up to but not including the right and bottom coordinates) with the specified brush. This function doesn't require that you first select the brush into the device context.

FrameRect uses the brush to draw a rectangular frame, but it does not fill in the rectangle. Using a brush to draw a frame may seem a little strange, because with the functions that you've seen so far (such as *Rectangle*) the border is drawn with the current pen. *FrameRect* allows you to draw a rectangular frame that isn't necessarily a pure color. This frame is one logical unit wide. If logical units are larger than device units, the frame will be 2 or more pixels wide.

InvertRect inverts all the pixels in the rectangle, turning ones to zeros and zeros to ones. This function turns a white area to black, a black area to white, and a green area to magenta.

Windows also includes nine functions that allow you to manipulate RECT structures easily and cleanly. For instance, to set the four fields of a RECT structure to particular values, you would conventionally use code that looks like this:

```
rect.left    = xLeft ;  
rect.top     = xTop  ;  
rect.right   = xRight ;  
rect.bottom  = xBottom ;
```

By calling the *SetRect* function, however, you can achieve the same result with a single line:

```
SetRect (&rect, xLeft, yTop, xRight, yBottom) ;
```

Clipping with Rectangles and Regions

Regions can also play a role in clipping. The *InvalidateRect* function invalidates a rectangular area of the display and generates a WM_PAINT message. For example, you can use the *InvalidateRect* function to erase the client area and generate a WM_PAINT message:

`InvalidateRect (hwnd, NULL, TRUE) ;`

You can obtain the coordinates of the invalid rectangle by calling *GetUpdateRect*, and you can validate a rectangle of the client area using the *ValidateRect* function. When you receive a `WM_PAINT` message, the coordinates of the invalid rectangle are available from the `PAINTSTRUCT` structure that is filled in by the *BeginPaint* function. This invalid rectangle also defines a "clipping region." You cannot paint outside the clipping region.

Windows has two functions similar to *InvalidateRect* and *ValidateRect* that work with regions rather than rectangles:

`InvalidateRgn (hwnd, hRgn, bErase) ;`

and

`ValidateRgn (hwnd, hRgn) ;`

When you receive a `WM_PAINT` message as a result of an invalid region, the clipping region will not necessarily be rectangular in shape.

You can create a clipping region of your own by selecting a region into the device context using either

`SelectObject (hdc, hRgn) ;`

or

`SelectClipRgn (hdc, hRgn) ;`

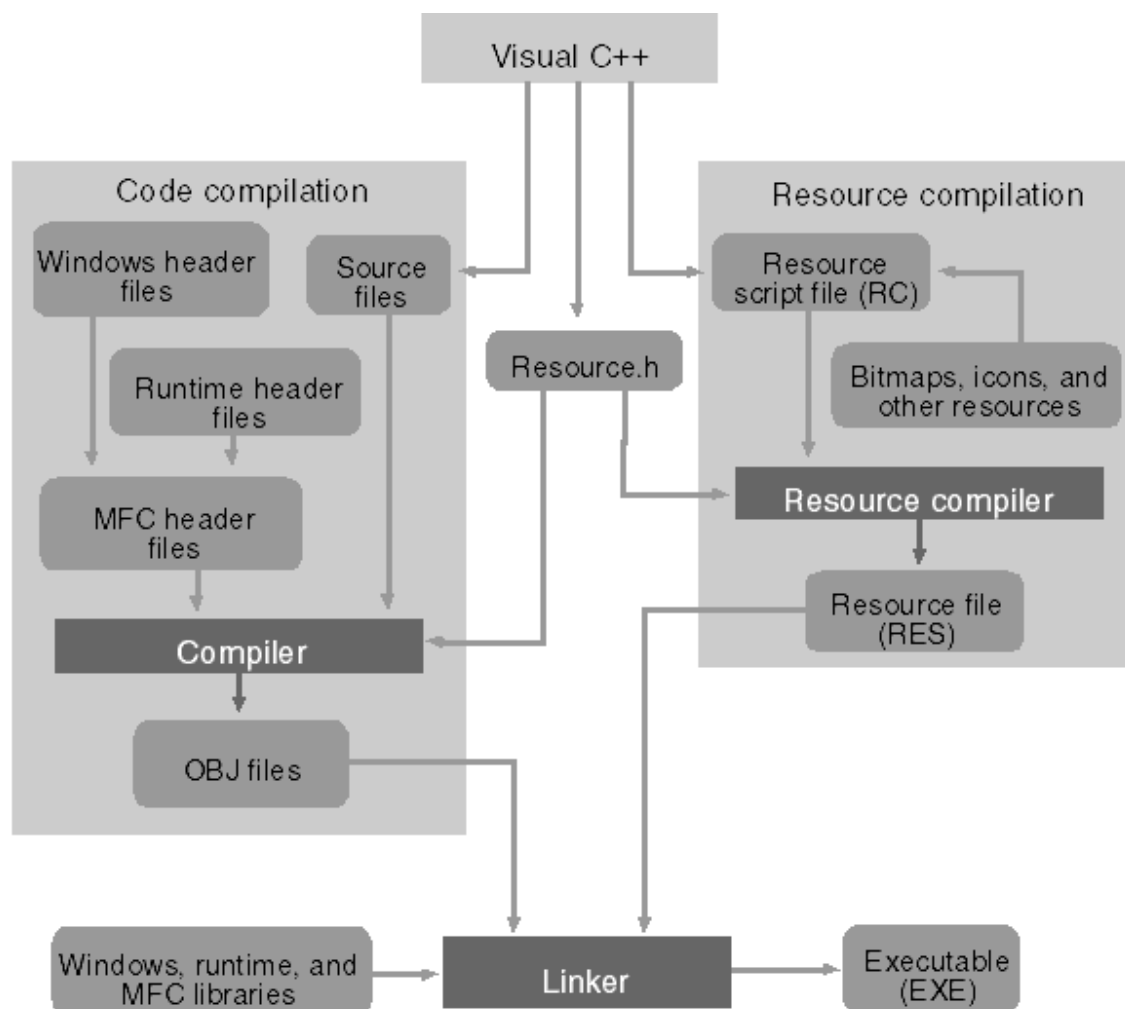
A clipping region is assumed to be measured in device coordinates.

GDI makes a copy of the clipping region, so you can delete the region object after you select it in the device context. Windows also includes several functions to manipulate this clipping region, such as *ExcludeClipRect* to exclude a rectangle from the clipping region, *IntersectClipRect* to create a new clipping region that is the intersection of the previous clipping region and a rectangle, and *OffsetClipRgn* to move a clipping region to another part of the client area.

VISUAL PROGRAMMING

Unit – II

1. Explain about VC++ components



The Visual C++ Components

Microsoft Visual C++ is two complete Windows application development systems in one product. If you so choose, you can develop C-language Windows programs using only the Win32 API. C-language Win32 programming is described in Charles Petzold's book *Programming Windows 95* (Microsoft Press, 1996). You can use many Visual C++ tools, including the resource editors, to make low-level Win32 programming easier.

Visual C++ also includes the ActiveX Template Library (ATL), which you can use to develop ActiveX controls for the Internet. ATL programming is neither Win32 C-language programming nor MFC programming, and it's complex enough to deserve its own book.

Use of the MFC library programming interface doesn't cut you off from the Win32 functions. In fact, you'll almost always need some direct Win32 calls in your MFC library programs.

Microsoft Visual C++ 6.0 and the Build Process

Visual Studio 6.0 is a suite of developer tools that includes Visual C++ 6.0. The Visual C++ IDE is shared by several tools including Microsoft Visual J++. The IDE has come a long way from the original Visual Workbench, which was based on QuickC for Windows. Docking windows, configurable toolbars, plus a customizable editor that runs macros, are now part of Visual Studio. The online help system (now integrated with the MSDN Library viewer) works like a Web browser. Figure 1-2 shows Visual C++ 6.0 in action.

If you've used earlier versions of Visual C++ or another vendor's IDE, you already understand how Visual C++ 6.0 operates. But if you're new to IDEs, you'll need to know what a project is. A project is a collection of interrelated source files that are compiled and linked to make up an executable Windows-based program or a DLL. Source files for each project are generally stored in a separate subdirectory. A project depends on many files outside the project subdirectory too, such as include files and library files.

Visual C++ creates some intermediate files too. The following table lists the files that Visual C++ generates in the workspace.

File Extension	Description
APS	Supports ResourceView
BSC	Browser information file
CLW	Supports ClassWizard
DEP	Dependency file
DSP	Project file*
DSW	Workspace file*
MAK	External makefile
NCB	Supports ClassView
OPT	Holds workspace configuration
PLG	Builds log file

* Do not delete or edit in a text editor.

The Resource Editors—Workspace ResourceView

When you click on the ResourceView tab in the Visual C++ Workspace window, you can select a resource for editing. The main window hosts a resource editor appropriate for the resource type. The window can also host a wysiwyg editor for menus and a powerful graphical editor for dialog boxes, and it includes tools for editing icons, bitmaps, and strings. The dialog editor allows you to insert ActiveX controls in addition to standard Windows controls and the new Windows common controls (which have been further extended in Visual C++ 6.0). Each project usually has one text-format resource script (RC) file that describes the project's menu, dialog, string, and accelerator resources. The RC file also has *#include* statements to bring in resources from other subdirectories. These resources include project-specific items, such as bitmap (BMP) and icon (ICO) files, and resources common to all Visual C++ programs, such as error message strings. Editing the RC file outside the resource editors is not recommended. The resource editors can also process

EXE and DLL files, so you can use the clipboard to "steal" resources, such as bitmaps and icons, from other Windows applications.

The C/C++ Compiler

The Visual C++ compiler can process both C source code and C++ source code. It determines the language by looking at the source code's filename extension. A C extension indicates C source code, and CPP or CXX indicates C++ source code. The compiler is compliant with all ANSI standards, including the latest recommendations of a working group on C++ libraries, and has additional Microsoft extensions. Templates, exceptions, and runtime type identification (RTTI) are fully supported in Visual C++ version 6.0. The C++ Standard Template Library (STL) is also included, although it is not integrated into the MFC library.

The Source Code Editor

Visual C++ 6.0 includes a sophisticated source code editor that supports many features such as dynamic syntax coloring, auto-tabbing, keyboard bindings for a variety of popular editors (such as VI and EMACS), and pretty printing. In Visual C++ 6.0, an exciting new feature named AutoComplete has been added. If you have used any of the Microsoft Office products or Microsoft Visual Basic, you might already be familiar with this technology. Using the Visual C++ 6.0 AutoComplete feature, all you have to do is type the beginning of a programming statement and the editor will provide you with a list of possible completions to choose from. This feature is extremely handy when you are working with C++ objects and have forgotten an exact member function or data member name—they are all there in the list for you. You no longer have to memorize thousands of Win32 APIs or rely heavily on the online help system, thanks to this new feature.

The Resource Compiler

The Visual C++ resource compiler reads an ASCII resource script (RC) file from the resource editors and writes a binary RES file for the linker.

The Linker

The linker reads the OBJ and RES files produced by the C/C++ compiler and the resource compiler, and it accesses LIB files for MFC code, runtime library code, and Windows code. It then writes the project's EXE file. An incremental link option minimizes the execution time when only minor changes have been made to the source files. The MFC header files contain *#pragma* statements (special compiler directives) that specify the required library files, so you don't have to tell the linker explicitly which libraries to read.

The Debugger

If your program works the first time, you don't need the debugger. The rest of us might need one from time to time. The Visual C++ debugger has been steadily improving, but it doesn't actually fix the bugs yet. The debugger works closely with Visual C++ to ensure that breakpoints are saved on disk. Toolbar buttons insert and remove breakpoints and control single-step execution. Figure 1-3 illustrates the Visual C++ debugger in action. Note that the Variables and Watch windows can expand an object pointer to show all data members of the derived class and base classes. If you position the cursor on a simple variable, the debugger shows you its value in a little window. To debug a program, you must build the program with the compiler and linker options set to generate debugging information.

Figure 1-3. *The Visual C++ debugger window.*

Visual C++ 6.0 adds a new twist to debugging with the Edit And Continue feature. Edit And Continue lets you debug an application, change the application, and then continue debugging with the new code. This feature dramatically reduces the amount of time you spend debugging because you no longer have to manually leave the debugger, recompile, and then debug again. To use this feature, simply edit any code while in the debugger and then hit the continue button. Visual C++ 6.0 automatically compiles the changes and restarts the debugger for you.

AppWizard

AppWizard is a code generator that creates a working skeleton of a Windows application with features, class names, and source code filenames that you specify through dialog boxes. You'll use AppWizard extensively as you work through the examples in this book. Don't confuse AppWizard with older code generators that generate all the code for an application. AppWizard code is minimalist code; the functionality is inside the application framework base classes. AppWizard gets you started quickly with a new application.

Advanced developers can build custom AppWizards. Microsoft Corporation has exposed its macro-based system for generating projects. If you discover that your team needs to develop multiple projects with a telecommunications interface, you can build a special wizard that automates the process.

ClassWizard

ClassWizard is a program (implemented as a DLL) that's accessible from Visual C++'s View menu. ClassWizard takes the drudgery out of maintaining Visual C++ class code. Need a new class, a new virtual function, or a new message-handler function? ClassWizard writes the prototypes, the function bodies, and (if necessary) the code to link the Windows message to the function. ClassWizard can update class code that you write, so you avoid the maintenance problems common to ordinary code generators. Some ClassWizard features are available from Visual C++'s WizardBar toolbar, shown in Figure 1-2.

The Source Browser

If you write an application from scratch, you probably have a good mental picture of your source code files, classes, and member functions. If you take over someone else's application, you'll need some assistance. The Visual C++ Source Browser (the browser, for short) lets you examine (and edit) an application from the class or function viewpoint instead of from the file viewpoint. It's a little like the "inspector" tools available with object-oriented libraries such as Smalltalk. The browser has the following viewing modes:

- **Definitions and References**—You select any function, variable, type, macro, or class and then see where it's defined and used in your project.
- **Call Graph/Callers Graph**—For a selected function, you'll see a graphical representation of the functions it calls or the functions that call it.
- **Derived Classes and Members/Base Classes and Members**—These are graphical class hierarchy diagrams. For a selected class, you see the derived classes or the base classes plus members. You can control the hierarchy expansion with the mouse.
- **File Outline**—For a selected file, the classes, functions, and data members appear together with the places in which they're defined and used in your project.

If you rearrange the lines in any source code file, Visual C++ regenerates the browser database when you rebuild the project. This increases the build time.

2. List out the feature of application framework

One definition of application framework is "an integrated collection of object-oriented software components that offers all that's needed for a generic application." That isn't a very useful definition, is it? If you really want to know what an application framework is, you'll have to read the rest of this book. The application framework example that you'll familiarize yourself with later in this chapter is a good starting point.

Why Use the Application Framework?

If you're going to develop applications for Windows, you've got to choose a development environment. Assuming that you've already rejected non-C options such as Microsoft Visual Basic and Borland Delphi, here are some of your remaining options:

- Program in C with the Win32 API
- Write your own C++ Windows class library that uses Win32
- Use the MFC application framework
- Use another Windows-based application framework such as Borland's Object Windows Library (OWL)

If you're starting from scratch, any option involves a big learning curve. If you're already a Win16 or Win32 programmer, you'll still have a learning curve with the MFC library. Since its release, MFC has become the dominant Windows class library. But even if you're familiar with it, it's still a good idea to step through the features of this programming choice.

The MFC library is the C++ Microsoft Windows API. If you accept the premise that the C++ language is now the standard for serious application development, you'd have to say that it's natural for Windows to have a C++ programming interface. What better interface is there than the one produced by Microsoft, creator of Windows? That interface is the MFC library.

Application framework applications use a standard structure. Any programmer starting on a large project develops some kind of structure for the code. The problem is that each programmer's structure is different, and it's difficult for a new team member to learn the structure and conform to it. The MFC library application framework includes its own application structure—one that's been proven in many software environments and in many projects. If you write a program for Windows that uses the MFC library, you can safely retire to a Caribbean island, knowing that your minions can easily maintain and enhance your code back home.

Don't think that the MFC library's structure makes your programs inflexible. With the MFC library, your program can call Win32 functions at any time, so you can take maximum advantage of Windows.

Application framework applications are small and fast. Back in the 16-bit days, you could build a self-contained Windows EXE file that was less than 20 kilobytes (KB) in size. Today, Windows programs are larger. One reason is that 32-bit code is fatter. Even with the large memory model, a Win16 program used 16-bit addresses for stack variables and many globals. Win32 programs use 32-bit addresses for everything and often use 32-bit integers because they're more efficient than 16-bit integers. In addition, the new C++ exception-handling code consumes a lot of memory.

That old 20-KB program didn't have a docking toolbar, splitter windows, print preview capabilities, or control container support—features that users expect in modern programs. MFC programs are bigger because they do more and look better. Fortunately, it's now easy to build applications that dynamically link to the MFC code (and to C runtime code), so the size goes back down again—from 192 KB to about 20 KB! Of course, you'll need some big support DLLs in the background, but those are a fact of life these days.

As far as speed is concerned, you're working with machine code produced by an optimizing compiler. Execution is fast, but you might notice a startup delay while the support DLLs are loaded.

The Visual C++ tools reduce coding drudgery. The Visual C++ resource editors, AppWizard, and ClassWizard significantly reduce the time needed to write code that is specific to your application. For example, the resource editor creates a header file that contains assigned values for *#define* constants. AppWizard generates skeleton code for your entire application, and ClassWizard generates prototypes and function bodies for message handlers.

." This document-view architecture is the core of the application framework and is loosely based on the Model/View/Controller classes from the Smalltalk world.

In simple terms, the document-view architecture separates data from the user's view of the data. One obvious benefit is multiple views of the same data. Consider a document that consists of a month's worth of stock quotes stored on disk. Suppose a table view and a chart view of the data are both available. The user updates values through the table view window, and the chart view window changes because both windows display the same information (but in different views).

In an MFC library application, documents and views are represented by instances of C++ classes

3. What are the various types of mapping modes available .

Mapping Modes

Up to now, your drawing units have been display pixels, also known as device coordinates. The EX04A drawing units are pixels because the device context has the default mapping mode, *MM_TEXT*, assigned to it. The statement

```
pDC->Rectangle(CRect(0, 0, 200, 200));
```

draws a square of 200-by-200 pixels, with its top-left corner at the top left of the window's client area. (Positive y values increase as you move down the window.) This square would look smaller on a high-resolution display of 1024-by-768 pixels than it would look on a standard VGA display that is 640-by-480 pixels, and it would look tiny if printed on a laser printer with 600-dpi resolution. (Try EX04A's Print Preview feature to see for yourself.)

What if you want the square to be 4-by-4 centimeters (cm), regardless of the display device? Windows provides a number of other mapping modes, or coordinate systems, that can be associated with the device context. Coordinates in the current mapping mode are called logical coordinates. If you assign the *MM_HIMETRIC* mapping mode, for example, a logical unit is $\frac{1}{100}$ millimeter (mm) instead of 1 pixel. In the *MM_HIMETRIC* mapping mode, the y axis runs in the opposite direction to that in the *MM_TEXT* mode: y values decrease as you move down. Thus, a 4-by-4-cm square is drawn in logical coordinates this way:

```
pDC->Rectangle(CRect(0, 0, 4000, -4000));
```


Looks easy, doesn't it? Well, it isn't, because you can't work only in logical coordinates. Your program is always switching between device coordinates and logical coordinates, and you need to know when to convert between them. This section gives you a few rules that could make your programming life easier. First you need to know what mapping modes Windows gives you.

The *MM_TEXT* Mapping Mode

At first glance, *MM_TEXT* appears to be no mapping mode at all, but rather another name for device coordinates. Almost. In *MM_TEXT*, coordinates map to pixels, values of x increase as you move right, and values of y increase as you move down, but you're allowed to change the origin through calls to the *CDC* functions *SetViewportOrg* and *SetWindowOrg*. Here's some code that sets the window origin to (100, 100) in logical coordinate space and then draws a 200-by-200-pixel square offset by (100, 100). (An illustration of the output is shown in Figure 4-2.) The logical point (100, 100) maps to the device point (0, 0). A scrolling window uses this kind of transformation.

```
void CMyView::OnDraw(CDC* pDC)
{
    pDC->SetMapMode(MM_TEXT);
    pDC->SetWindowOrg(CPoint(100, 100));
    pDC->Rectangle(CRect(100, 100, 300, 300));
}
```

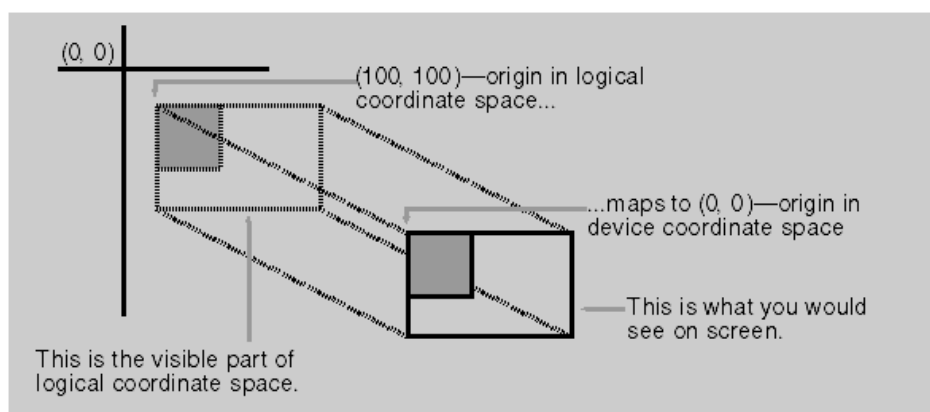


Figure 4-2. A square drawn after the origin has been moved to (100, 100).

The Fixed-Scale Mapping Modes

One important group of Windows mapping modes provides fixed scaling. You have already seen that, in the *MM_HIMETRIC* mapping mode, x values increase as you move right and y values decrease as you move down. All fixed mapping modes follow this convention, and you can't change it. The only difference among the fixed mapping modes is the actual scale factor, listed in the table shown here.

Mapping Mode	Logical Unit
MM_LOENGLISH	0.01 inch
MM_HIENGLISH	0.001 inch
MM_LOMETRIC	0.1 mm

MM_HIMETRIC

0.01 mm

MM_TWIPS

$\frac{1}{1440}$ inch

The last mapping mode, *MM_TWIPS*, is most often used with printers. One *twip* unit is $\frac{1}{20}$ point. (A *point* is a type measurement unit. In Windows it equals exactly $\frac{1}{72}$ inch.) If the mapping mode is *MM_TWIPS* and you want, for example, 12-point type, set the character height to 12×20 , or 240, twips.

The Variable-Scale Mapping Modes

Windows provides two mapping modes, *MM_ISOTROPIC* and *MM_ANISOTROPIC*, that allow you to change the scale factor as well as the origin. With these mapping modes, your drawing can change size as the user changes the size of the window. Also, if you invert the scale of one axis, you can "flip" an image about the other axis and you can define your own arbitrary fixed-scale factors.

With the *MM_ISOTROPIC* mode, a 1:1 aspect ratio is always preserved. In other words, a circle is always a circle as the scale factor changes. With the *MM_ANISOTROPIC* mode, the *x* and *y* scale factors can change independently. Circles can be squished into ellipses.

Here's an *OnDraw* function that draws an ellipse that fits exactly in its window:

```
void CMyView::OnDraw(CDC* pDC)
{
    CRect rectClient;

    GetClientRect(rectClient);
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1000, 1000);
    pDC->SetViewportExt(rectClient.right, -rectClient.bottom);
    pDC->SetViewportOrg(rectClient.right / 2, rectClient.bottom / 2);

    pDC->Ellipse(CRect(-500, -500, 500, 500));
}
```

What's going on here? The functions *SetWindowExt* and *SetViewportExt* work together to set the scale, based on the window's current client rectangle returned by the *GetClientRect* function. The resulting window size is exactly 1000-by-1000 logical units. The *SetViewportOrg* function sets the origin to the center of the window. Thus, a centered ellipse with a radius of 500 logical units fills the window exactly, as illustrated in Figure 4-3.

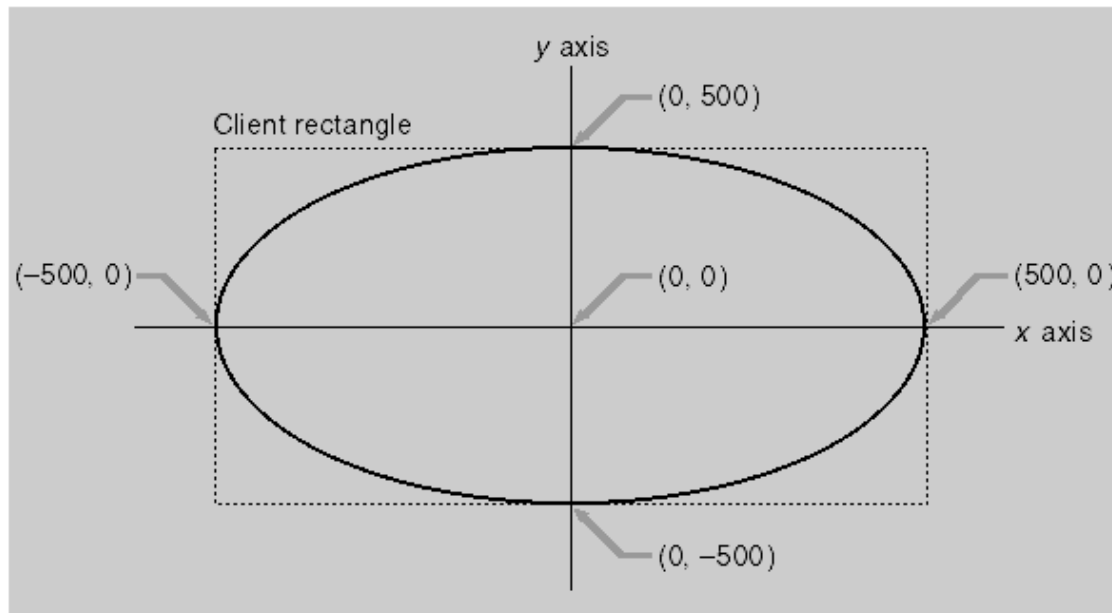


Figure 4-3. A centered ellipse drawn in the `MM_ANISOTROPIC` mapping mode.

Here are the formulas for converting logical units to device units:

$x \text{ scale factor} = x \text{ viewport extent} / x \text{ window extent}$
 $y \text{ scale factor} = y \text{ viewport extent} / y \text{ window extent}$
 $\text{device } x = \text{logical } x \times x \text{ scale factor} + x \text{ origin offset}$
 $\text{device } y = \text{logical } y \times y \text{ scale factor} + y \text{ origin offset}$

Suppose the window is 448 pixels wide (`rectClient.right`). The right edge of the ellipse's client rectangle is 500 logical units from the origin. The x scale factor is $448/1000$, and the x origin offset is $448/2$ device units. If you use the formulas shown on the previous page, the right edge of the ellipse's client rectangle comes out to 448 device units, the right edge of the window. The x scale factor is expressed as a ratio (viewport extent/window extent) because Windows device coordinates are integers, not floating-point values. The extent values are meaningless by themselves.

If you substitute `MM_ISOTROPIC` for `MM_ANISOTROPIC` in the preceding example, the "ellipse" is always a circle, as shown in Figure 4-4. It expands to fit the smallest dimension of the window rectangle.

4.Explain MFC (to explain SDI, MDI, and Dialog Based)

5.Explain in detail about GDI aobjects

GDI Objects

A Windows GDI object type is represented by an MFC library class. `CGdiObject` is the abstract base class for the GDI object classes. A Windows GDI object is represented by a C++ object of a class derived from `CGdiObject`. Here's a list of the GDI derived classes:

- **CBitmap**—A bitmap is an array of bits in which one or more bits correspond to each display pixel. You can use bitmaps to represent images, and you can use them to create brushes.
- **CBrush**—A brush defines a bitmapped pattern of pixels that is used to fill areas with color.
- **CFont**—A font is a complete collection of characters of a particular typeface and a particular size. Fonts are generally stored on disk as resources, and some are device-specific.
- **CPalette**—A palette is a color mapping interface that allows an application to take full advantage of the color capability of an output device without interfering with other applications.
- **CPen**—A pen is a tool for drawing lines and shape borders. You can specify a pen's color and thickness and whether it draws solid, dotted, or dashed lines.
- **CRgn**—A region is an area whose shape is a polygon, an ellipse, or a combination of polygons and ellipses. You can use regions for filling, clipping, and mouse hit-testing.

Constructing and Destroying GDI Objects

You never construct an object of class *CGdiObject*; instead, you construct objects of the derived classes. Constructors for some GDI derived classes, such as *CPen* and *CBrush*, allow you to specify enough information to create the object in one step. Others, such as *CFont* and *CRgn*, require a second creation step. For these classes, you construct the C++ object with the default constructor and then you call a create function such as the *CreateFont* or *CreatePolygonRgn* function.

The *CGdiObject* class has a virtual destructor. The derived class destructors delete the Windows GDI objects that are attached to the C++ objects. If you construct an object of a class derived from *CGdiObject*, you must delete it prior to exiting the program. To delete a GDI object, you must first separate it from the device context. You'll see an example of this in the next section.

Failure to delete a GDI object was a serious offense with Win16. GDI memory was not released until the user restarted Windows. With Win32, however, the GDI memory is owned by the process and is released when your program terminates. Still, an unreleased GDI bitmap object can waste a significant amount of memory.

Tracking GDI Objects

OK, so you know that you have to delete your GDI objects and that they must first be disconnected from their device contexts. How do you disconnect them? A member of the *CDC::SelectObject* family of functions does the work of selecting a GDI object into the device context, and in the process it returns a pointer to the previously selected object (which gets deselected in the process). Trouble is, you can't deselect the old object without selecting a new object. One easy way to track the objects is to "save" the original GDI object when you select your own GDI object and "restore" the original object when you're finished. Then you'll be ready to delete your own GDI object. Here's an example:

```
void CMyView::OnDraw(CDC* pDC)
{
    CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0); // black pen,
                                                // 2 pixels wide
    CPen* pOldPen = pDC->SelectObject(&newPen);

    pDC->MoveTo(10, 10);
    pDC->LineTo(110, 10);
    pDC->SelectObject(pOldPen);                // newPen is deselected
```

```
} // newPen automatically destroyed on exit
```

When a device context object is destroyed, all its GDI objects are deselected. Thus, if you know that a device context will be destroyed before its selected GDI objects are destroyed, you don't have to deselect the objects. If, for example, you declare a pen as a view class data member (and you initialize it when you initialize the view), you don't have to deselect the pen inside *OnDraw* because the device context, controlled by the view base class's *OnPaint* handler, will be destroyed first.

Stock GDI Objects

Windows contains a number of stock GDI objects that you can use. Because these objects are part of Windows, you don't have to worry about deleting them. (Windows ignores requests to delete stock objects.) The MFC library function *CDC::SelectStockObject* selects a stock object into the device context and returns a pointer to the previously selected object, which it deselects. Stock objects are handy when you want to deselect your own nonstock GDI object prior to its destruction. You can use a stock object as an alternative to the "old" object you used in the previous example, as shown here:

```
void CMyView::OnDraw(CDC* pDC)
{
    CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0); // black pen,
                                                // 2 pixels wide

    pDC->SelectObject(&newPen);
    pDC->MoveTo(10, 10);
    pDC->LineTo(110, 10);
    pDC->SelectStockObject(BLACK_PEN);          // newPen is deselected
} // newPen destroyed on exit
```

The Microsoft Foundation Class Reference lists, under *CDC::SelectStockObject*, the stock objects available for pens, brushes, fonts, and palettes.

6. Explain about Windows color mapping

Windows Color Mapping

The Windows GDI provides a hardware-independent color interface. Your program supplies an "absolute" color code, and the GDI maps that code to a suitable color or color combination on your computer's video display. Most programmers of applications for Windows try to optimize their applications' color display for a few common video card categories.

Standard Video Graphics Array Video Cards

A standard Video Graphics Array (VGA) video card uses 18-bit color registers and thus has a palette of 262,144 colors. Because of video memory constraints, however, the standard VGA board accommodates 4-bit color codes, which means it can display only 16 colors at a time. Because Windows needs fixed colors for captions, borders, scroll bars, and so forth, your programs can use only 16 "standard" pure colors. You cannot conveniently access the other colors that the board can display.

Each Windows color is represented by a combination of 8-bit "red," "green," and "blue" values. The 16 standard VGA "pure" (nondithered) colors are shown in the table below.

Color-oriented GDI functions accept 32-bit *COLORREF* parameters that contain 8-bit color codes each for red, green, and blue. The Windows RGB macro converts 8-bit red, green, and blue values to a *COLORREF* parameter. The following statement, when executed on a system with a standard VGA board, constructs a brush with a dithered color (one that consists of a pattern of pure-color pixels):

```
CBrush brush(RGB(128, 128, 192));
```

Red	Green	Blue	Color
0	0	0	Black
0	0	255	Blue
0	255	0	Green
0	255	255	Cyan
255	0	0	Red
255	0	255	Magenta
255	255	0	Yellow
255	255	255	White
0	0	128	Dark blue
0	128	0	Dark green
0	128	128	Dark cyan
128	0	0	Dark red
128	0	128	Dark magenta
128	128	0	Dark yellow
128	128	128	Dark gray
192	192	192	Light gray

The following statement (in your view's *OnDraw* function) sets the text background to red:

```
pDC->SetBkColor(RGB(255, 0, 0));
```

The *CDC* functions *SetBkColor* and *SetTextColor* don't display dithered colors as the brush-oriented drawing functions do. If the dithered color pattern is too complex, the closest matching pure color is displayed.

256-Color Video Cards

Most video cards can accommodate 8-bit color codes at all resolutions, which means they can display 256 colors simultaneously. This 256-color mode is now considered to be the "lowest common denominator" for color programming.

If Windows is configured for a 256-color display card, your programs are limited to 20 standard pure colors unless you activate the Windows color palette system as supported by the MFC library *CPalette* class and the

Windows API, in which case you can choose your 256 colors from a total of more than 16.7 million.. With an SVGA 256-color display driver installed, you get the 16 VGA colors listed in the previous table plus 4 more, for a total of 20. The following table lists the 4 additional colors.

Red	Green	Blue	Color
192	220	192	Money green
166	202	240	Sky blue
255	251	240	Cream
160	160	164	Medium gray

The RGB macro works much the same as it does with the standard VGA. If you specify one of the 20 standard colors for a brush, you get a pure color; otherwise, you get a dithered color. If you use the *PALETTE*RGB macro instead, you don't get dithered colors; you get the closest matching standard pure color as defined by the current palette.

16-Bit-Color Video Cards

Most modern video cards support a resolution of 1024-by-768 pixels, and 1 MB of video memory can support 8-bit color at this resolution. If a video card has 2 MB of memory, it can support 16-bit color, with 5 bits each for red, green, and blue. This means that it can display 32,768 colors simultaneously. That sounds like a lot, but there are only 32 shades each of pure red, green, and blue. Often, a picture will look better in 8-bit-color mode with an appropriate palette selected. A forest scene, for example, can use up to 236 shades of green. Palettes are not supported in 16-bit-color mode.

24-Bit-Color Video Cards

High-end cards (which are becoming more widely used) support 24-bit color. This 24-bit capability enables the display of more than 16.7 million pure colors. If you're using a 24-bit card, you have direct access to all the colors. The RGB macro allows you to specify the exact colors you want. You'll need 2.5 MB of video memory, though, if you want 24-bit color at 1024-by-768-pixel resolution.

7.What are modal and modeless windows.

Modal vs. Modeless Dialogs

The *CDialog* base class supports both modal and modeless dialogs. With a modal dialog, such as the Open File dialog, the user cannot work elsewhere in the same application (more correctly, in the same user interface thread) until the dialog is closed. With a modeless dialog, the user can work in another window in the application while the dialog remains on the screen. Microsoft Word's Find and Replace dialog is a good example of a modeless dialog; you can edit your document while the dialog is open.

Your choice of a modal or a modeless dialog depends on the application. Modal dialogs are much easier to program, which might influence your decision.

Programming a Modal Dialog

Modal dialogs are the most frequently used dialogs. A user action (a menu choice, for example) brings up a dialog on the screen, the user enters data in the dialog, and then the user closes the dialog. Here's a summary of the steps to add a modal dialog to an existing project:

1. Use the dialog editor to create a dialog resource that contains various controls. The dialog editor updates the project's resource script (RC) file to include your new dialog resource, and it updates the project's resource.h file with corresponding *#define* constants.
2. Use ClassWizard to create a dialog class that is derived from *CDialog* and attached to the resource created in step 1. ClassWizard adds the associated code and header file to the Microsoft Visual C++ project.
3. Use ClassWizard to add data members, exchange functions, and validation functions to the dialog class.
4. Use ClassWizard to add message handlers for the dialog's buttons and other event-generating controls.
5. Write the code for special control initialization (in *OnInitDialog*) and for the message handlers. Be sure the *CDialog* virtual member function *OnOK* is called when the user closes the dialog (unless the user cancels the dialog). (Note: *OnOK* is called by default.)
6. Write the code in your view class to activate the dialog. This code consists of a call to your dialog class's constructor followed by a call to the *DoModal* dialog class member function. *DoModal* returns only when the user exits the dialog window.

Now we'll proceed with a real example, one step at a time.

Modeless Dialogs

In the Microsoft Foundation Class (MFC) Library version 6.0, modal and modeless dialogs share the same base class, *CDialog*, and they both use a dialog resource that you can build with the dialog editor. If you're using a modeless dialog with a view, you'll need to know some specialized programming techniques.

Creating Modeless Dialogs

For modal dialogs, you've already learned that you construct a dialog object using a *CDialog* constructor that takes a resource template ID as a parameter, and then you display the modal dialog window by calling the *DoModal* member function. The window ceases to exist as soon as *DoModal* returns. Thus, you can construct a modal dialog object on the stack, knowing that the dialog window has been destroyed by the time the C++ dialog object goes out of scope.

Modeless dialogs are more complicated. You start by invoking the *CDialog* default constructor to construct the dialog object, but then to create the dialog window you need to call the *CDialog::Create* member function instead of *DoModal*. *Create* takes the resource ID as a parameter and returns immediately with the dialog window still on the screen. You must worry about exactly when to construct the dialog object, when to create the dialog window, when to destroy the dialog, and when to process user-entered data.

Here's a summary of the differences between creating a modal dialog and a modeless dialog.

	Modal Dialog	Modeless Dialog
Constructor used	Constructor with resource ID param	Default constructor (no params)
Function used to create window	<i>DoModal</i>	<i>Create</i> with resource ID param

User-Defined Messages

Suppose you want the modeless dialog window to be destroyed when the user clicks the dialog's OK button. This presents a problem. How does the view know that the user has clicked the OK button? The dialog could call a view class member function directly, but that would "marry" the dialog to a particular view class. A better solution is for the dialog to send the view a user-defined message as the result of a call to the OK button message-handling function. When the view gets the message, it can destroy the dialog window (but not the object). This sets the stage for the creation of a new dialog.

You have two options for sending Windows messages: the *CWnd::SendMessage* function or the *PostMessage* function. The former causes an immediate call to the message-handling function, and the latter posts a message in the Windows message queue. Because there's a slight delay with the *PostMessage* option, it's reasonable to expect that the handler function has returned by the time the view gets the message.

8.What are the various windows common control under modal window.

Windows Common Controls

The controls you used in EX06A are great learning controls because they're easy to program. Now you're ready for some more "interesting" controls. We'll take a look at some important new Windows controls, introduced for Microsoft Windows 95 and available in Microsoft Windows NT. These include the progress indicator, trackbar, spin button control, list control, and tree control.

The code for these controls is in the Windows COMCTL32.DLL file. This code includes the window procedure for each control, together with code that registers a window class for each control. The registration code is called when the DLL is loaded. When your program initializes a dialog, it uses the symbolic class name in the dialog resource to connect to the window procedure in the DLL. Thus your program owns the control's window, but the code is in the DLL. Except for ActiveX controls, most controls work this way.

Example EX06B uses the aforementioned controls. Figure 6-2 shows the dialog from that example. Refer to it when you read the control descriptions that follow.

Be aware that ClassWizard offers no member variable support for the common controls. You'll have to add code to your *OnInitDialog* and *OnOK* functions to initialize and read control data. ClassWizard will, however, allow you to map notification messages from common controls.

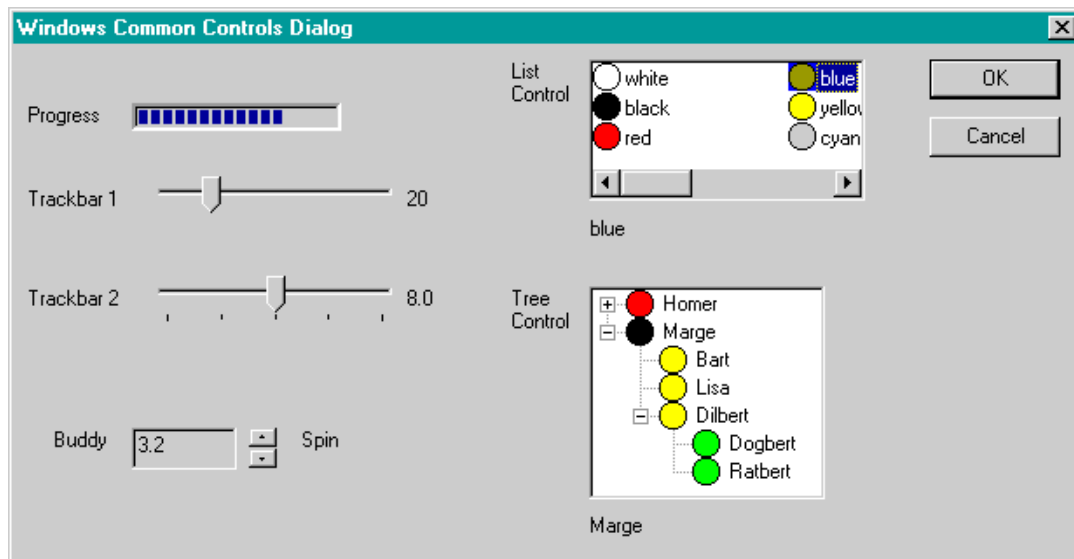


Figure 6-2. *The Windows Common Controls Dialog example.*

The Progress Indicator Control

The progress indicator is the easiest common control to program and is represented by the MFC *CProgressCtrl* class. It is generally used only for output. This control, together with the trackbar, can effectively replace the scroll bar controls you saw in the previous example. To initialize the progress indicator, call the *SetRange* and *SetPos* member functions in your *OnInitDialog* function, and then call *SetPos* anytime in your message handlers. The progress indicator shown in Figure 6-2 has a range of 0 to 100, which is the default range.

The Trackbar Control

The trackbar control (class *CSliderCtrl*), sometimes called a slider, allows the user to set an "analog" value. (Trackbars would have been more effective than sliders for Loyalty and Reliability in the EX06A example.) If you specify a large range for this control—0 to 100 or more, for example—the trackbar's motion appears continuous. If you specify a small range, such as 0 to 5, the tracker moves in discrete increments. You can program tick marks to match the increments. In this discrete mode, you can use a trackbar to set such items as the display screen resolution, lens f-stop values, and so forth. The trackbar does not have a default range.

The trackbar is easier to program than the scroll bar because you don't have to map the *WM_HSCROLL* or *WM_VSCROLL* messages in the dialog class. As long as you set the range, the tracker moves when the user slides it or clicks in the body of the trackbar. You might choose to map the scroll messages anyway if you want to show the position value in another control. The *GetPos* member function returns the current position value. The top trackbar in Figure 6-2 operates continuously in the range 0 to 100. The bottom trackbar has a range of 0 to 4, and those indexes are mapped to a series of double-precision values (4.0, 5.6, 8.0, 11.0, and 16.0).

The Spin Button Control

The spin button control (class *CSpinButtonCtrl*) is an itty-bitsy scroll bar that's most often used in conjunction with an edit control. The edit control, located just ahead of the spin control in the dialog's tabbing order, is known as the spin control's buddy. The idea is that the user holds down the left mouse

button on the spin control to raise or lower the value in the edit control. The spin speed accelerates as the user continues to hold down the mouse button.

If your program uses an integer in the buddy, you can avoid C++ programming almost entirely. Just use ClassWizard to attach an integer data member to the edit control, and set the spin control's range in the *OnInitDialog* function. (You probably won't want the spin control's default range, which runs backward from a minimum of 100 to a maximum of 0.) Don't forget to select Auto Buddy and Set Buddy Integer in the spin control's Styles property page. You can call the *SetRange* and *SetAccel* member functions in your *OnInitDialog* function to change the range and the acceleration profile.

If you want your edit control to display a noninteger, such as a time or a floating-point number, you must map the spin control's WM_VSCROLL (or WM_HSCROLL) messages and write handler code to convert the spin control's integer to the buddy's value.

The List Control

Use the list control (class *CListCtrl*) if you want a list that contains images as well as text. Figure 6-2 shows a list control with a "list" view style and small icons. The elements are arranged in a grid, and the control includes horizontal scrolling. When the user selects an item, the control sends a notification message, which you map in your dialog class. That message handler can determine which item the user selected. Items are identified by a zero-based integer index.

Both the list control and the tree control get their graphic images from a common control element called an image list (class *CImageList*). Your program must assemble the image list from icons or bitmaps and then pass an image list pointer to the list control. Your *OnInitDialog* function is a good place to create and attach the image list and to assign text strings. The *InsertItem* member function serves this purpose.

List control programming is straightforward if you stick with strings and icons. If you implement drag and drop or if you need custom owner-drawn graphics, you've got more work to do.

The Tree Control

You're already familiar with tree controls if you've used Microsoft Windows Explorer or Visual C++'s Workspace view. The MFC *CTreeCtrl* class makes it easy to add this same functionality to your own programs. Figure 6-2 illustrates a tree control that shows a modern American combined family. The user can expand and collapse elements by clicking the + and - buttons or by double-clicking the elements. The icon next to each item is programmed to change when the user selects the item with a single click.

The list control and the tree control have some things in common: they can both use the same image list, and they share some of the same notification messages. Their methods of identifying items are different, however. The tree control uses an *HTREEITEM* handle instead of an integer index. To insert an item, you call the *InsertItem* member function, but first you must build up a *TV_INSERTSTRUCT* structure that identifies (among other things) the string, the image list index, and the handle of the parent item (which is null for top-level items).

As with list controls, infinite customization possibilities are available for the tree control. For example, you can allow the user to edit items and to insert and delete items.

The WM_NOTIFY Message

The original Windows controls sent their notifications in WM_COMMAND messages. The standard 32-bit *wParam* and *lParam* message parameters are not sufficient, however, for the information that a common control needs to send to its parent. Microsoft solved this "bandwidth" problem by defining a new message, WM_NOTIFY. With the WM_NOTIFY message, *wParam* is the control ID and *lParam* is a pointer to an *NMHDR* structure, which is managed by the control. This C structure is defined by the following code:

```
typedef struct tagNMHDR {
    HWND hwndFrom; // handle to control sending the message
    UINT idFrom;   // ID of control sending the message
    UINT code;     // control-specific notification code
} NMHDR;
```

Many controls, however, send WM_NOTIFY messages with pointers to structures larger than *NMHDR*. Those structures contain the three members above plus appended control-specific members. Many tree control notifications, for example, pass a pointer to an *NM_TREEVIEW* structure that contains *TV_ITEM* structures, a drag point, and so forth. When ClassWizard maps a WM_NOTIFY message, it generates a pointer to the appropriate structure.

9. Explain about ActiveX controls?

ActiveX Controls vs. Ordinary Windows Controls

An ActiveX control is a software module that plugs into your C++ program the same way a Windows control does. At least that's the way it seems at first. It's worthwhile here to analyze the similarities and differences between ActiveX controls and the controls you already know.

Ordinary Controls—A Frame of Reference

In [Chapter 6](#), you used ordinary Windows controls such as the edit control and the list box, and you saw the Windows common controls that work in much the same way. These controls are all child windows that you use most often in dialogs, and they are represented by MFC classes such as *CEdit* and *CTreeCtrl*. The client program is always responsible for the creation of the control's child window.

Ordinary controls send notification command messages (standard Windows messages), such as BN_CLICKED, to the dialog. If you want to perform an action on the control, you call a C++ control class member function, which sends a Windows message to the control. The controls are all windows in their own right. All the MFC control classes are derived from *CWnd*, so if you want to get the text from an edit control, you call *CWnd::GetWindowText*. But even that function works by sending a message to the control.

Windows controls are an integral part of Windows, even though the Windows common controls are in a separate DLL. Another species of ordinary control, the so-called custom control, is a programmer-created control that acts as an ordinary control in that it sends WM_COMMAND notifications to its parent window and receives user-defined messages. You'll see one of these in [Chapter 22](#).

How ActiveX Controls Are Similar to Ordinary Controls

You can consider an ActiveX control to be a child window, just as an ordinary control is. If you want to include an ActiveX control in a dialog, you use the dialog editor to place it there, and the identifier for the control turns up in the resource template. If you're creating an ActiveX control on the fly, you call a *Create* member function for a class that represents the control, usually in the WM_CREATE handler for the parent

window. When you want to manipulate an ActiveX control, you call a C++ member function, just as you do for a Windows control. The window that contains a control is called a container.

How ActiveX Controls Are Different from Ordinary Controls—Properties and Methods

The most prominent ActiveX Controls features are properties and methods. Those C++ member functions that you call to manipulate a control instance all revolve around properties and methods. Properties have symbolic names that are matched to integer indexes. For each property, the control designer assigns a property name, such as `BackColor` or `GridCellEffect`, and a property type, such as string, integer, or double. There's even a picture type for bitmaps and icons. The client program can set an individual ActiveX control property by specifying the property's integer index and its value. The client can get a property by specifying the index and accepting the appropriate return value. In certain cases, ClassWizard lets you define data members in your client window class that are associated with the properties of the controls the client class contains. The generated Dialog Data Exchange (DDX) code exchanges data between the control properties and the client class data members.

ActiveX Controls methods are like functions. A method has a symbolic name, a set of parameters, and a return value. You call a method by calling a C++ member function of the class that represents the control. A control designer can define any needed methods, such as *PreviousYear*, *LowerControlRods*, and so forth.

An ActiveX control doesn't send `WM_` notification messages to its container the way ordinary controls do; instead, it "fires events." An event has a symbolic name and can have an arbitrary sequence of parameters—it's really a container function that the control calls. Like ordinary control notification messages, events don't return a value to the ActiveX control. Examples of events are `Click`, `KeyDown`, and `NewMonth`. Events are mapped in your client class just as control notification messages are.

In the MFC world, ActiveX controls act just like child windows, but there's a significant layer of code between the container window and the control window. In fact, the control might not even have a window. When you call *Create*, the control's window isn't created directly; instead, the control code is loaded and given the command for "in-place activation." The ActiveX control then creates its own window, which MFC lets you access through a `CWnd` pointer. It's not a good idea for the client to use the control's `hWnd` directly, however.

A DLL is used to store one or more ActiveX controls, but the DLL often has an OCX filename extension instead of a DLL extension. Your container program loads the DLLs when it needs them, using sophisticated COM techniques that rely on the Windows Registry. For the time being, simply accept the fact that once you specify an ActiveX control at design time, it will be loaded for you at runtime. Obviously, when you ship a program that requires special ActiveX controls, you'll have to include the OCX files and an appropriate setup program.

Installing ActiveX Controls

Let's assume you've found a nifty ActiveX control that you want to use in your project. Your first step is to copy the control's DLL to your hard disk. You could put it anywhere, but it's easier to track your ActiveX controls if you put them in one place, such as in the system directory (typically `\Windows\System` for Microsoft Windows 95 or `\Winnt\System32` for Microsoft Windows NT). Copy associated files such as help (HLP) or license (LIC) files to the same directory.

Your next step is to register the control in the Windows Registry.

After you register your ActiveX control, you must install it in each project that uses it. That doesn't mean that the OCX file gets copied. It means that ClassWizard generates a copy of a C++ class that's specific to the control, and it means that the control shows up in the dialog editor control palette for that project.

To install an ActiveX control in a project, choose Add To Project from the Project menu and then choose Components And Controls. Select Registered ActiveX Controls, as shown in the following illustration.

The Calendar Control

The MSCal.ocx control is a popular Microsoft ActiveX Calendar control that's probably already installed and registered on your computer.

Creating ActiveX Controls at Runtime

You've seen how to use the dialog editor to insert ActiveX controls at design time. If you need to create an ActiveX control at runtime without a resource template entry, here are the programming steps:

1. Insert the component into your project. ClassWizard will create the files for a wrapper class.
2. Add an embedded ActiveX control wrapper class data member to your dialog class or other C++ window class. An embedded C++ object is then constructed and destroyed along with the window object.
3. Choose Resource Symbols from Visual C++'s View menu. Add an ID constant for the new control.
4. If the parent window is a dialog, use ClassWizard to map the dialog's WM_INITDIALOG message, thus overriding *CDialog::OnInitDialog*. For other windows, use ClassWizard to map the WM_CREATE message. The new function should call the embedded control class's *Create* member function. This call indirectly displays the new control in the dialog. The control will be properly destroyed when the parent window is destroyed.
5. In the parent window class, manually add the necessary event message handlers and prototypes for your new control. Don't forget to add the event sink map macros.

10.Explain about Menus and keyboard accelerator.

Windows Menus

A Microsoft Windows menu is a familiar application element that consists of a top-level horizontal list of items with associated pop-up menus that appear when the user selects a top-level item. Most of the time, you define for a frame window a default menu resource that loads when the window is created. You can also define a menu resource independent of a frame window. In that case, your program must call the functions necessary to load and activate the menu.

A menu resource completely defines the initial appearance of a menu. Menu items can be grayed or have check marks, and bars can separate groups of menu items. Multiple levels of pop-up menus are possible. If a first-level menu item is associated with a subsidiary pop-up menu, the menu item carries a right-pointing arrow symbol, as shown next to the Start Debug menu item in Figure 13-2.

Visual C++ includes an easy-to-use menu-resource editing tool. This tool lets you edit menus in a wysiwyg environment. Each menu item has a properties dialog that defines all the characteristics of that item. The resulting resource definition is stored in the application's resource script (RC) file. Each menu item is associated with an ID, such as *ID_FILE_OPEN*, that is defined in the resource.h file.

The keyboard accelerator resource consists of a table of key combinations with associated command IDs. The Edit Copy menu item (with command ID `ID_EDIT_COPY`), for example, might be linked to the Ctrl-C key combination through a keyboard accelerator entry. A keyboard accelerator entry does not have to be associated with a menu item. If no Edit Copy menu item were present, the Ctrl-C key combination would nevertheless activate the `ID_EDIT_COPY` command.

11. Explain about Tool bar and status bar.

Control Bars and the Application Framework

The toolbar is an object of class `CToolBar`, and the status bar is an object of class `CStatusBar`. Both these classes are derived from class `CControlBar`, which is itself derived from `CWnd`. The `CControlBar` class supports control bar windows that are positioned inside frame windows. These control bar windows resize and reposition themselves as the parent frame moves and changes size. The application framework takes care of the construction, window creation, and destruction of the control bar objects. AppWizard generates control bar code for its derived frame class located in the files `MainFrm.cpp` and `MainFrm.h`.

In a typical SDI application, a `CToolBar` object occupies the top portion of the `CMainFrame` client area and a `CStatusBar` object occupies the bottom portion. The view occupies the remaining (middle) part of the frame.

The Toolbar

A toolbar consists of a number of horizontally (or vertically) arranged graphical buttons that might be clustered in groups. The programming interface determines the grouping. The graphical images for the buttons are stored in a single bitmap that is attached to the application's resource file. When a button is clicked, it sends a command message, as do menus and keyboard accelerators. An update command UI message handler is used to update the button's state, which in turn is used by the application framework to modify the button's graphical image.

The Toolbar Bitmap

Each button on a toolbar appears to have its own bitmap, but actually a single bitmap serves the entire toolbar. The toolbar bitmap has a tile, 15 pixels high and 16 pixels wide, for each button. The application framework supplies the button borders, and it modifies those borders, together with the button's bitmap tile color, to reflect the current button state. Figure 14-1 shows the relationship between the toolbar bitmap and the corresponding toolbar.

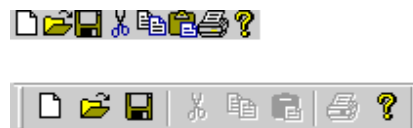


Figure 14-1. *A toolbar bitmap and an actual toolbar.*

The toolbar bitmap is stored in the file `Toolbar.bmp` in the application's `\res` subdirectory. The bitmap is identified in the resource script (RC) file as `IDR_MAINFRAME`. You don't edit the toolbar bitmap directly; instead you use Visual C++'s special toolbar-editing facility.

The Status Bar

The status bar window neither accepts user input nor generates command messages. Its job is simply to display text in panes under program control. The status bar supports two types of text panes—message line panes and status indicator panes. To use the status bar for application-specific data, you must first disable the standard status bar that displays the menu prompt and key-board status.

The Status Bar Definition

The static *indicators* array that AppWizard generates in the `MainFrm.cpp` file defines the panes for the application's status bar. The constant `ID_SEPARATOR` identifies a message line pane; the other constants are string resource IDs that identify indicator panes. Figure 14-3 shows the *indicators* array and its relationship to the standard framework status bar.

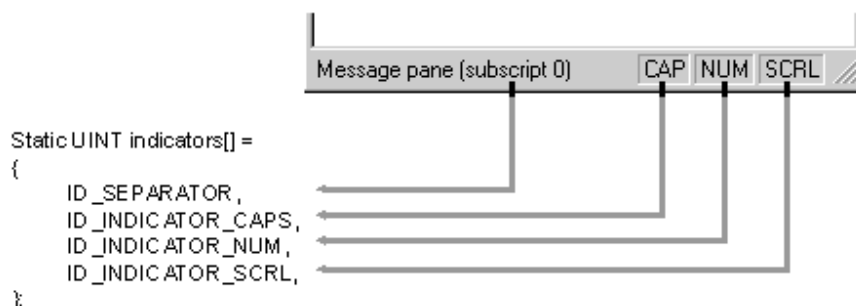


Figure 14-3. The status bar and the indicators array.

The `CStatusBar::SetIndicators` member function, called in the application's derived frame class, configures the status bar according to the contents of the *indicators* array.

The Message Line

A message line pane displays a string that the program supplies dynamically. To set the value of the message line, you must first get access to the status bar object and then you must call the `CStatusBar::SetPaneText` member function with a zero-based index parameter. Pane 0 is the leftmost pane, 1 is the next pane to the right, and so forth.

The following code fragment is part of a view class member function. Note that you must navigate up to the application object and then back down to the main frame window.

```
CMainFrame* pFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;  
CStatusBar* pStatus = &pFrame->m_wndStatusBar;  
pStatus->SetPaneText(0, "message line for first pane");
```

Normally, the length of a message line pane is exactly one-fourth the width of the display. If, however, the message line is the first (index 0) pane, it is a stretchy pane without a beveled border. Its minimum length is one-fourth the display width, and it expands if room is available in the status bar.

The Status Indicator

A status indicator pane is linked to a single resource-supplied string that is displayed or hidden by logic in an associated update command UI message handler function. An indicator is identified by a string resource ID, and that same ID is used to route update command UI messages. The Caps Lock indicator is handled in the

frame class by a message map entry and a handler function equivalent to those shown below. The *Enable* function turns on the indicator if the Caps Lock mode is set.

ON_UPDATE_COMMAND_UI(ID_INDICATOR_CAPS, OnUpdateKeyCapsLock)

```
void CMainFrame::OnUpdateKeyCapsLock(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(::GetKeyState(VK_CAPITAL) & 1);
}
```

The status bar update command UI functions are called during idle processing so that the status bar is updated whenever your application receives messages.

The length of a status indicator pane is the exact length of the corresponding resource string.

12. How will you register an application

If you've used Win16-based applications, you've probably seen INI files. You can still use INI files in Win32-based applications, but Microsoft recommends that you use the Windows Registry instead. The Registry is a set of system files, managed by Windows, in which Windows and individual applications can store and access permanent information. The Registry is organized as a kind of hierarchical database in which string and integer data is accessed by a multipart key.

For example, a text processing application, TEXTPROC, might need to store the most recent font and point size in the Registry. Suppose that the program name forms the root of the key (a simplification) and that the application maintains two hierarchy levels below the name. The structure looks something like this.

TEXTPROC

Text formatting

Font = Times Roman

Points = 10

The MFC library provides four *CWinApp* member functions, holdovers from the days of INI files, for accessing the Registry. Starting with Visual C++ version 5.0, AppWizard generates a call to *CWinApp::SetRegistryKey* in your application's *InitInstance* function as shown here.

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

If you remove this call, your application will not use the Registry but will create and use an INI file in the Windows directory. The *SetRegistryKey* function's string parameter establishes the top of the hierarchy, and the following Registry functions define the bottom two levels: called heading name and entry name.

- *GetProfileInt*
- *WriteProfileInt*
- *GetProfileString*
- *WriteProfileString*

These functions treat Registry data as either *CString* objects or unsigned integers. If you need floating-point values as entries, you must use the string functions and do the conversion yourself. All the functions take a heading name and an entry name as parameters. In the example shown above, the heading name is Text Formatting and the entry names are Font and Points.

To use the Registry access functions, you need a pointer to the application object. The global function *AfxGetApp* does the job. With the previous sample Registry, the Font and Points entries were set with the following code:

```
AfxGetApp()->WriteProfileString("Text formatting", "Font",  
                                "Times Roman");  
AfxGetApp()->WriteProfileInt("Text formatting", "Points", 10);
```

VISUAL PROGRAMMING

Unit – III

1.What is meant by serialization ?

Serialization—What Is It?

The term "serialization" might be new to you, but it's already seen some use in the world of object-oriented programming. The idea is that objects can be persistent, which means they can be saved on disk when a program exits and then can be restored when the program is restarted. This process of saving and restoring objects is called serialization. In the Microsoft Foundation Class (MFC) library, designated classes have a member function named *Serialize*. When the application framework calls *Serialize* for a particular object—for example, an object of class *CStudent*—the data for the student is either saved on disk or read from disk.

In the MFC library, serialization is not a substitute for a database management system. All the objects associated with a document are sequentially read from or written to a single disk file. It's not possible to access individual objects at random disk file addresses.

Disk Files and Archives

How do you know whether *Serialize* should read or write data? How is *Serialize* connected to a disk file? With the MFC library, objects of class *CFile* represent disk files. A *CFile* object encapsulates the binary file handle that you get through the Win32 function *CreateFile*. This is not the buffered *FILE* pointer that you'd get with a call to the C runtime *fopen* function; rather, it's a handle to a binary file. The application framework uses this file handle for Win32 *ReadFile*, *WriteFile*, and *SetFilePointer* calls.

If your application does no direct disk I/O but instead relies on the serialization process, you can avoid direct use of *CFile* objects. Between the *Serialize* function and the *CFile* object is an archive object (of class *CArchive*), as shown in Figure 17-1.

The *CArchive* object buffers data for the *CFile* object, and it maintains an internal flag that indicates whether the archive is storing (writing to disk) or loading (reading from disk). Only one active archive is associated with a file at any one time. The application framework takes care of constructing the *CFile* and *CArchive* objects, opening the disk file for the *CFile* object and associating the archive object with the file. All you have to do (in your *Serialize* function) is load data from or store data in the archive object. The application

framework calls the document's *Serialize* function during the File Open and File Save processes.

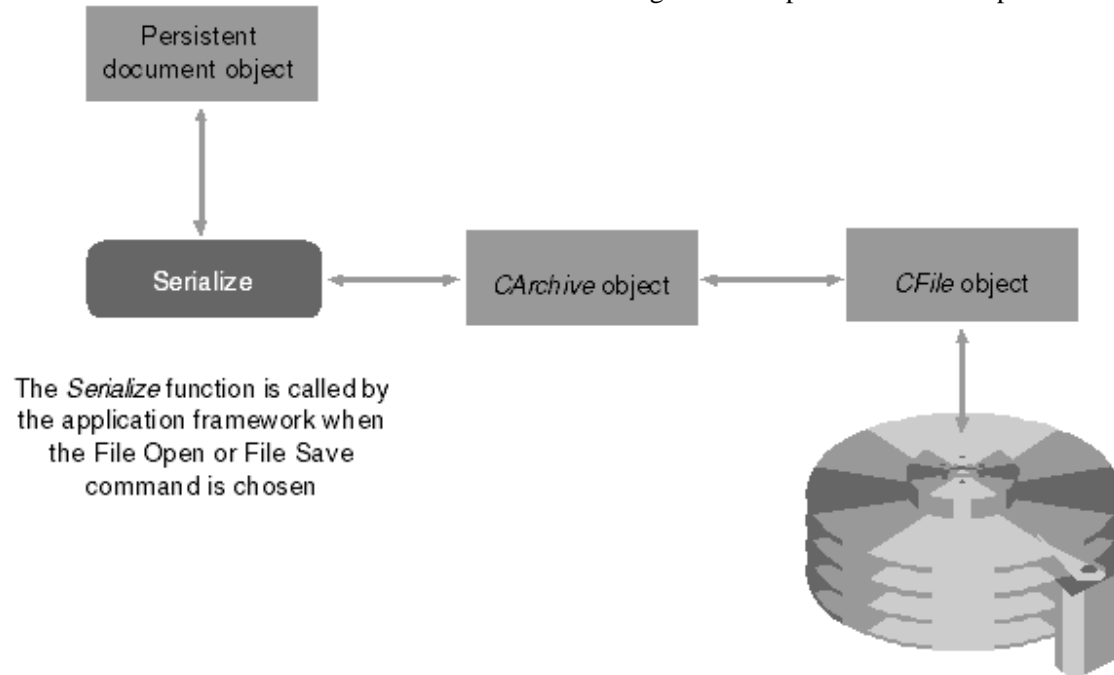


Figure 17-1. *The serialization process.*

Making a Class Serializable

A serializable class must be derived directly or indirectly from *CObject*. In addition (with some exceptions), the class declaration must contain the *DECLARE_SERIAL* macro call, and the class implementation file must contain the *IMPLEMENT_SERIAL* macro call. (See the *Microsoft Foundation Class Reference* for a description of these macros.)

Writing a *Serialize* Function

In [Chapter 16](#), you saw a *CStudent* class, derived from *CObject*, with these data members:

```
public:
    CString m_strName;
    int     m_nGrade;
```

Now your job is to write a *Serialize* member function for *CStudent*. Because *Serialize* is a virtual member function of class *CObject*, you must be sure that the return value and parameter types match the *CObject* declaration. The *Serialize* function for the *CStudent* class is below.

```
void CStudent::Serialize(CArchive& ar)
{
    TRACE("Entering CStudent::Serialize\n");
    if (ar.IsStoring()) {
        ar << m_strName << m_nGrade;
```

```
    }  
    else {  
        ar >> m_strName >> m_nGrade;  
    }  
}
```

Most serialization functions call the *Serialize* functions of their base classes. If *CStudent* were derived from *CPerson*, for example, the first line of the *Serialize* function would be

```
CPerson::Serialize(ar);
```

The *Serialize* function for *CObject* (and for *CDocument*, which doesn't override it) doesn't do anything useful, so there's no need to call it.

The insertion operators are overloaded for values; the extraction operators are overloaded for references. Sometimes you must use a cast to satisfy the compiler. Suppose you have a data member *m_nType* that is an enumerated type. Here's the code you would use:

```
ar << (int) m_nType;  
ar >> (int&) m_nType;
```

MFC classes that are not derived from *CObject*, such as *CString* and *CRect*, have their own overloaded insertion and extraction operators for *CArchive*.

```
void CStudent::Serialize(CArchive& ar)  
{  
    if (ar.IsStoring()) {  
        ar << m_strName << m_nGrade;  
    }  
    else {  
        ar >> m_strName >> m_nGrade;  
    }  
    m_transcript.Serialize(ar);  
}
```

Before the *CStudent::Serialize* function can be called to load a student record from the archive, a *CStudent* object must exist somewhere. The embedded *CTranscript* object *m_transcript* is constructed along with the *CStudent* object before the call to the *CTranscript::Serialize* function. When the virtual *CTranscript::Serialize* function does get called, it can load the archived transcript data into the embedded *m_transcript* object. If you're looking for a rule, here it is: always make a direct call to *Serialize* for embedded objects of classes derived from *CObject*.

Suppose that, instead of an embedded object, your *CStudent* object contained a *CTranscript* pointer data member such as this:

```
public:  
    CTranscript* m_pTranscript;
```

You could use the *Serialize* function, as shown below, but as you can see, you must construct a new *CTranscript* object yourself.

```
void CStudent::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_strName << m_nGrade;
    else {
        m_pTranscript = new CTranscript;
        ar >> m_strName >> m_nGrade;
    }
    m_pTranscript->Serialize(ar);
}
```

2.Explain in detail about splitter window and its types?

The Splitter Window

A splitter window appears as a special type of frame window that holds several views in panes. The application can split the window on creation, or the user can split the window by choosing a menu command or by dragging a splitter box on the window's scroll bar. After the window has been split, the user can move the splitter bars with the mouse to adjust the relative sizes of the panes. Splitter windows can be used in both SDI and MDI applications. You can see examples of splitter windows in this chapter.

An object of class *CSplitterWnd* represents the splitter window. As far as Windows is concerned, a *CSplitterWnd* object is an actual window that fully occupies the frame window (*CFrameWnd* or *CMDIChildWnd*) client area. The view windows occupy the splitter window pane areas. The splitter window does not take part in the command dispatch mechanism. The active view window (in a splitter pane) is connected directly to its frame window.

View Options

When you combine multiview presentation methods with application models, you get a number of permutations. Here are some of them:

- **SDI application with splitter window, single view class** This chapter's first example, EX20A, covers this case. Each splitter window pane can be scrolled to a different part of the document. The programmer determines the maximum number of horizontal and vertical panes; the user makes the split at runtime.
- **SDI application with splitter window, multiple view classes** The EX20B example illustrates this case. The programmer determines the number of panes and the sequence of views; the user can change the pane size at runtime.
- **SDI application with no splitter windows, multiple view classes** The EX20C example illustrates this case. The user switches view classes by making a selection from a menu.
- **MDI application with no splitter windows, single view class** This is the standard MDI application you've seen already in [Chapter 18](#). The New Window menu item lets the user open a new child window for a document that's open already.
- **MDI application with no splitter windows, multiple view classes** A small change to the standard MDI application allows the use of multiple views. As example EX20D shows, all that's necessary is to add a menu item and a handler function for each additional view class available.
- **MDI application with splitter child windows** This case is covered thoroughly in the online documentation. The SCRIBBLE example illustrates the splitting of an MDI child window.

Dynamic and Static Splitter Windows

A dynamic splitter window allows the user to split the window at any time by choosing a menu item or by dragging a splitter box located on the scroll bar. The panes in a dynamic splitter window generally use the same view class. The top left pane is initialized to a particular view when the splitter window is created. In a dynamic splitter window, scroll bars are shared among the views. In a window with a single horizontal split, for example, the bottom scroll bar controls both views. A dynamic splitter application starts with a single view object. When the user splits the frame, other view objects are constructed. When the user unsplit the frame, view objects are destroyed.

The panes of a static splitter window are defined when the window is first created and they cannot be changed. The user can move the bars but cannot unsplit or resplit the window. Static splitter windows can accommodate multiple view classes, with the configuration set at creation time. In a static splitter window, each pane has separate scroll bars. In a static splitter window application, all view objects are constructed when the frame is constructed and they are all destroyed when the frame is destroyed.

3.Explain about DLL

Basically, a DLL is a file on disk (usually with a DLL extension) consisting of global data, compiled functions, and resources, that becomes part of your process. It is compiled to load at a preferred base address, and if there's no conflict with other DLLs, the file gets mapped to the same virtual address in your process. The DLL has various exported functions, and the client program (the program that loaded the DLL in the first place) imports those functions. Windows matches up the imports and exports when it loads the DLL.

How Imports Are Matched to Exports

A DLL contains a table of exported functions. These functions are identified to the outside world by their symbolic names and (optionally) by integers called ordinal numbers. The function table also contains the addresses of the functions within the DLL. When the client program first loads the DLL, it doesn't know the addresses of the functions it needs to call, but it does know the symbols or ordinals. The dynamic linking process then builds a table that connects the client's calls to the function addresses in the DLL. If you edit and rebuild the DLL, you don't need to rebuild your client program unless you have changed function names or parameter sequences.

In the DLL code, you must explicitly declare your exported functions like this:

```
__declspec(dllexport) int MyFunction(int n);
```

(The alternative is to list your exported functions in a module-definition [DEF] file, but that's usually more troublesome.) On the client side, you need to declare the corresponding imports like this:

```
__declspec(dllimport) int MyFunction(int n);
```

If you're using C++, the compiler generates a decorated name for *MyFunction* that other languages can't use. These decorated names are the long names the compiler invents based on class name, function name, and parameter types. They are listed in the project's MAP file. If you want to use the plain name *MyFunction*, you have to write the declarations this way:

```
extern "C" __declspec(dllexport) int MyFunction(int n);  
extern "C" __declspec(dllimport) int MyFunction(int n);
```

By default, the compiler uses the `__cdecl` argument passing convention, which means that the calling program pops the parameters off the stack. Some client languages might require the `__stdcall` convention, which replaces the Pascal calling convention, and which means that the called function pops the stack. Therefore, you might have to use the `__stdcall` modifier in your DLL export declaration.

Just having import declarations isn't enough to make a client link to a DLL. The client's project must specify the import library (LIB) to the linker, and the client program must actually contain a call to at least one of the DLL's imported functions. That call statement must be in an executable path in the program.

How the Client Program Finds a DLL

If you link explicitly with `LoadLibrary`, you can specify the DLL's full pathname. If you don't specify the pathname, or if you link implicitly, Windows follows this search sequence to locate your DLL:

1. The directory containing the EXE file
2. The process's current directory
3. The Windows system directory
4. The Windows directory
5. The directories listed in the Path environment variable

Here's a trap you can easily fall into. You build a DLL as one project, copy the DLL file to the system directory, and then run the DLL from a client program. So far, so good. Next you rebuild the DLL with some changes, but you forget to copy the DLL file to the system directory. The next time you run the client program, it loads the old version of the DLL. Be careful!

Debugging a DLL

Visual C++ makes debugging a DLL easy. Just run the debugger from the DLL project. The first time you do this, the debugger asks for the pathname of the client EXE file. Every time you "run" the DLL from the debugger after this, the debugger loads the EXE, but the EXE uses the search sequence to find the DLL. This means that you must either set the Path environment variable to point to the DLL or copy the DLL to a directory in the search sequence.

4. Explain about MFC DLL's

AppWizard lets you build two kinds of DLLs with MFC library support: extension DLLs and regular DLLs. You must understand the differences between these two types before you decide which one is best for your needs.

Of course, Visual C++ lets you build a pure Win32 DLL without the MFC library, just as it lets you build a Windows program without the MFC library. This is an MFC-oriented book, however, so we'll ignore the Win32 option here.

An extension DLL supports a C++ interface. In other words, the DLL can export whole classes and the client can construct objects of those classes or derive classes from them. An extension DLL dynamically links to the code in the DLL version of the MFC library. Therefore, an extension DLL requires that your client program be dynamically linked to the MFC library (the AppWizard default) and that both the client program and the extension DLL be synchronized to the same version of the MFC DLLs (mfc42.dll, mfc42d.dll, and so

on). Extension DLLs are quite small; you can build a simple extension DLL with a size of 10 KB, which loads quickly.

If you need a DLL that can be loaded by any Win32 programming environment (including Visual Basic version 6.0), you should use a regular DLL. A big restriction here is that the regular DLL can export only C-style functions. It can't export C++ classes, member functions, or overloaded functions because every C++ compiler has its own method of decorating names. You can, however, use C++ classes (and MFC library classes, in particular) inside your regular DLL.

When you build an MFC regular DLL, you can choose to statically link or dynamically link to the MFC library. If you choose static linking, your DLL will include a copy of all the MFC library code it needs and will thus be self-contained. A typical Release-build statically linked regular DLL is about 144 KB in size. If you choose dynamic linking, the size drops to about 17 KB but you'll have to ensure that the proper MFC DLLs are present on the target machine. That's no problem if the client program is already dynamically linked to the same version of the MFC library.

When you tell AppWizard what kind of DLL or EXE you want, compiler *#define* constants are set as shown in the following table.

	Dynamically Linked to Shared MFC Library	Statically Linked* to MFC Library
Regular DLL	<code>_AFXDLL, _USRDLL</code>	<code>_USRDLL</code>
Extension DLL	<code>_AFXEXT, _AFXDLL</code>	unsupported option
Client EXE	<code>_AFXDLL</code>	no constants defined

* Visual C++ Learning Edition does not support the static linking option.

If you look inside the MFC source code and header files, you'll see a ton of *#ifdef* statements for these constants. This means that the library code is compiled quite differently depending on the kind of project you're producing.

The Shared MFC DLLs and the Windows DLLs

If you build a Windows Debug target with the shared MFC DLL option, your program is dynamically linked to one or more of these (ANSI) MFC DLLs:

mfc42d.dll	Core MFC classes
mfc42d.dll	ActiveX (OLE) classes
mfc42d.dll	Database classes (ODBC and DAO)
mfc42d.dll	Winsock, WinInet classes

When you build a Release target, your program is dynamically linked to mfc42.dll only. Linkage to these MFC DLLs is implicit via import libraries. You might assume implicit linkage to the ActiveX and ODBC DLLs in Windows, in which case you would expect all these DLLs to be linked to your Release-build client

when it loads, regardless of whether it uses ActiveX or ODBC features. However, this is not what happens. Through some creative thinking, MFC loads the ActiveX and ODBC DLLs explicitly (by calling *LoadLibrary*) when one of their functions is first called. Your client application thus loads only the DLLs it needs.

MFC Extension DLLs—Exporting Classes

If your extension DLL contains only exported C++ classes, you'll have an easy time building and using it. The steps for building the EX22A example show you how to tell AppWizard that you're building an extension DLL skeleton. That skeleton has only the *DllMain* function. You simply add your own C++ classes to the project. There's only one special thing you must do. You must add the macro *AFX_EXT_CLASS* to the class declaration, as shown here:

```
class AFX_EXT_CLASS CStudent : public CObject
```

This modification goes into the H file that's part of the DLL project, and it also goes into the H file that client programs use. In other words, the H files are exactly the same for both client and DLL. The macro generates different code depending on the situation—it exports the class in the DLL and imports the class in the client.

6.Implement a DLL function?

The EX22C Example—An MFC Regular DLL

This example creates a regular DLL that exports a single square root function. First you'll build the ex22c.dll file, and then you'll modify the test client program, EX22B, to test the new DLL.

Here are the steps for building the EX22C example:

1. **Run AppWizard to produce \vcpp32\ex22c\ex22c.** Proceed as you did for EX22A, but accept Regular DLL Using Shared MFC DLL (instead of choosing MFC Extension DLL) from the one and only AppWizard page.
2. **Examine the ex22c.cpp file.** AppWizard generates the following code, which includes a derived *CWinApp* class:

```
// ex22c.cpp : Defines the initialization routines for the DLL.  
//
```

```
#include "stdafx.h"  
#include "ex22c.h"
```

```
#ifdef _DEBUG  
#define new DEBUG_NEW  
#undef THIS_FILE  
static char THIS_FILE[] = __FILE__ ;  
#endif
```

(generated comment lines omitted)

```
////////////////////////////////////  
// CEx22cApp
```

```
BEGIN_MESSAGE_MAP(CEx22cApp, CWinApp)
//{{AFX_MSG_MAP(CEx22cApp)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
////////////////////////////////////
// CEx22cApp construction
```

```
CEx22cApp::CEx22cApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}
```

```
////////////////////////////////////
// The one and only CEx22cApp object
```

```
CEx22cApp theApp;
```

Add the code for the exported *Ex22cSquareRoot* function. It's okay to add this code in the *ex22c.cpp* file, although you can use a new file if you want to:

```
extern "C" __declspec(dllexport) double Ex22cSquareRoot(double d)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    TRACE("Entering Ex22cSquareRoot\n");
    if (d >= 0.0) {
        return sqrt(d);
    }
    AfxMessageBox("Can't take square root of a negative number.");
    return 0.0;
}
```

You can see that there's no problem with the DLL displaying a message box or another modal dialog. You'll need to include *math.h* in the file containing this code.

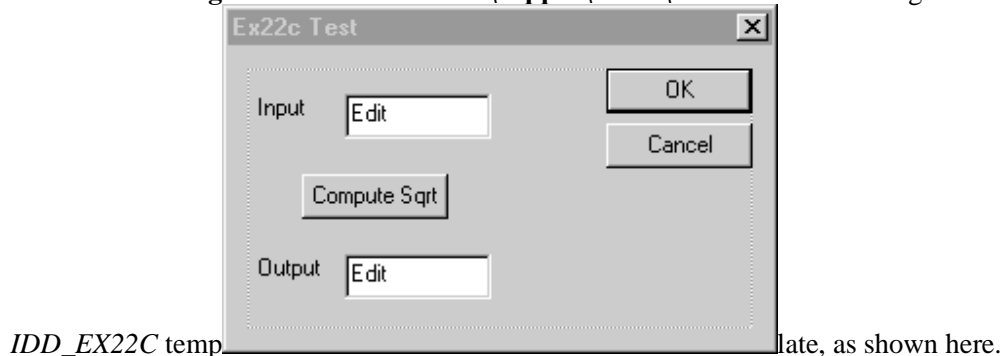
3. **Build the project and copy the DLL file.** Copy the file *ex22c.dll* from the *\vcpp32\ex22c\Debug* directory to your system directory.

Updating the EX22B Example—Adding Code to Test *ex22c.dll*

When you first built the EX22B program, it linked dynamically to the EX22A MFC extension DLL. Now you'll update the project to implicitly link to the EX22C MFC regular DLL and to call the DLL's square root function.

Following are the steps for updating the EX22B example.

1. Add a new dialog resource and class to `\vcpp32\ex22b\ex22b`. Use the dialog editor to create the



`IDD_EX22C` template, as shown here.

Then use ClassWizard to generate a class `CTest22cDialog`, derived from `CDialog`. The controls, data members, and message map function are shown in the following table.

Control ID	Type	Data Member	Message Map Function
<code>IDC_INPUT</code>	edit	<code>m_dInput</code> (double)	
<code>IDC_OUTPUT</code>	edit	<code>m_dOutput</code> (double)	
<code>IDC_COMPUTE</code>	button		<code>OnCompute</code>

2. Code the `OnCompute` function to call the DLL's exported function. Edit the ClassWizard-generated function in `Test22cDialog.cpp` as shown here:

```

3. void CTest22cDialog::OnCompute()
4. {
5.     UpdateData(TRUE);
6.     m_dOutput = Ex22cSquareRoot(m_dInput);
7.     UpdateData(FALSE);
8. }
```

You'll have to declare the `Ex22cSquareRoot` function as an imported function. Add the following line to the `Test22cDialog.h` file:

```
extern "C" __declspec(dllimport) double Ex22cSquareRoot(double d);
```

9. Integrate the `CTest22cDialog` class into the EX22B application. You'll need to add a top-level menu, Test, and an Ex22c DLL option with the ID `ID_TEST_EX22CDLL`. Use ClassWizard to map this option to a member function in the `CEx22bView` class, and then code the handler in `Ex22bView.cpp` as follows:

```

10. void CEx22bView::OnTestEx22cdll()
11. {
12.     CTest22cDialog dlg;
13.     dlg.DoModal();
14. }
```

Of course, you'll have to add this line to the `Ex22bView.cpp` file:

```
#include "Test22cDialog.h"
```

15. **Add the EX22C import library to the linker's input library list.** Choose Settings from Visual C++'s Project menu, and then add `\vcpp32\ex22c\Debug\ex22c.lib` to the Object/Library Modules control on the Link page. (Use a space to separate the new entry from the existing entry.) Now the program should implicitly link to both the EX22A DLL and the EX22C DLL. As you can see, the client doesn't care whether the DLL is a regular DLL or an extension DLL. You just specify the LIB name to the linker.
16. **Build and test the updated EX22B application.** Choose Ex22c DLL from the Test menu. Type a number in the Input edit control, and then click the Compute Sqrt button. The result should appear in the Output control.

VISUAL PROGRAMMING

Unit – IV

1. Explain about ActiveX controls?

ActiveX Controls vs. Ordinary Windows Controls

An ActiveX control is a software module that plugs into your C++ program the same way a Windows control does. At least that's the way it seems at first. It's worthwhile here to analyze the similarities and differences between ActiveX controls and the controls you already know.

Ordinary Controls—A Frame of Reference

In [Chapter 6](#), you used ordinary Windows controls such as the edit control and the list box, and you saw the Windows common controls that work in much the same way. These controls are all child windows that you use most often in dialogs, and they are represented by MFC classes such as *CEdit* and *CTreeCtrl*. The client program is always responsible for the creation of the control's child window.

Ordinary controls send notification command messages (standard Windows messages), such as `BN_CLICKED`, to the dialog. If you want to perform an action on the control, you call a C++ control class member function, which sends a Windows message to the control. The controls are all windows in their own right. All the MFC control classes are derived from *CWnd*, so if you want to get the text from an edit control, you call *CWnd::GetWindowText*. But even that function works by sending a message to the control.

Windows controls are an integral part of Windows, even though the Windows common controls are in a separate DLL. Another species of ordinary control, the so-called custom control, is a programmer-created control that acts as an ordinary control in that it sends `WM_COMMAND` notifications to its parent window and receives user-defined messages.

How ActiveX Controls Are Similar to Ordinary Controls

You can consider an ActiveX control to be a child window, just as an ordinary control is. If you want to include an ActiveX control in a dialog, you use the dialog editor to place it there, and the identifier for the control turns up in the resource template. If you're creating an ActiveX control on the fly, you call a *Create* member function for a class that represents the control, usually in the `WM_CREATE` handler for the parent window. When you want to manipulate an ActiveX control, you call a C++ member function, just as you do for a Windows control. The window that contains a control is called a container.

How ActiveX Controls Are Different from Ordinary Controls—Properties and Methods

The most prominent ActiveX Controls features are properties and methods. Those C++ member functions that you call to manipulate a control instance all revolve around properties and methods. Properties have symbolic names that are matched to integer indexes. For each property, the control designer assigns a property name, such as `BackColor` or `GridCellEffect`, and a property type, such as string, integer, or double. There's even a picture type for bitmaps and icons.

The client program can set an individual ActiveX control property by specifying the property's integer index and its value. The client can get a property by specifying the index and accepting the appropriate return value. In certain cases, ClassWizard lets you define data members in your client window class that are associated with the properties of the controls the client class contains. The generated Dialog Data Exchange (DDX) code exchanges data between the control properties and the client class data members.

ActiveX Controls methods are like functions. A method has a symbolic name, a set of parameters, and a return value. You call a method by calling a C++ member function of the class that represents the control. A control designer can define any needed methods, such as *PreviousYear*, *LowerControlRods*, and so forth.

An ActiveX control doesn't send WM_ notification messages to its container the way ordinary controls do; instead, it "fires events." An event has a symbolic name and can have an arbitrary sequence of parameters—it's really a container function that the control calls. Like ordinary control notification messages, events don't return a value to the ActiveX control. Examples of events are Click, KeyDown, and NewMonth. Events are mapped in your client class just as control notification messages are.

In the MFC world, ActiveX controls act just like child windows, but there's a significant layer of code between the container window and the control window. In fact, the control might not even have a window. When you call *Create*, the control's window isn't created directly; instead, the control code is loaded and given the command for "in-place activation." The ActiveX control then creates its own window, which MFC lets you access through a *CWnd* pointer. It's not a good idea for the client to use the control's *hWnd* directly, however.

A DLL is used to store one or more ActiveX controls, but the DLL often has an OCX filename extension instead of a DLL extension. Your container program loads the DLLs when it needs them, using sophisticated COM techniques that rely on the Windows Registry. For the time being, simply accept the fact that once you specify an ActiveX control at design time, it will be loaded for you at runtime. Obviously, when you ship a program that requires special ActiveX controls, you'll have to include the OCX files and an appropriate setup program.

Installing ActiveX Controls

Let's assume you've found a nifty ActiveX control that you want to use in your project. Your first step is to copy the control's DLL to your hard disk. You could put it anywhere, but it's easier to track your ActiveX controls if you put them in one place, such as in the system directory (typically \Windows\System for Microsoft Windows 95 or \Winnt\System32 for Microsoft Windows NT). Copy associated files such as help (HLP) or license (LIC) files to the same directory.

Your next step is to register the control in the Windows Registry.

After you register your ActiveX control, you must install it in each project that uses it. That doesn't mean that the OCX file gets copied. It means that ClassWizard generates a copy of a C++ class that's specific to the control, and it means that the control shows up in the dialog editor control palette for that project.

To install an ActiveX control in a project, choose Add To Project from the Project menu and then choose Components And Controls. Select Registered ActiveX Controls, as shown in the following illustration.

The Calendar Control

The MSCal.ocx control is a popular Microsoft ActiveX Calendar control that's probably already installed and registered on your computer.

Creating ActiveX Controls at Runtime

You've seen how to use the dialog editor to insert ActiveX controls at design time. If you need to create an ActiveX control at runtime without a resource template entry, here are the programming steps:

1. Insert the component into your project. ClassWizard will create the files for a wrapper class.
2. Add an embedded ActiveX control wrapper class data member to your dialog class or other C++ window class. An embedded C++ object is then constructed and destroyed along with the window object.
3. Choose Resource Symbols from Visual C++'s View menu. Add an ID constant for the new control.
4. If the parent window is a dialog, use ClassWizard to map the dialog's WM_INITDIALOG message, thus overriding *CDialog::OnInitDialog*. For other windows, use ClassWizard to map the WM_CREATE message. The new function should call the embedded control class's *Create* member function. This call indirectly displays the new control in the dialog. The control will be properly destroyed when the parent window is destroyed.
5. In the parent window class, manually add the necessary event message handlers and prototypes for your new control. Don't forget to add the event sink map macros.

2. What is meant by COM?

The Component Object Model (COM) is the foundation of much of the new Microsoft ActiveX technology, and after five years it's become an integral part of Microsoft Windows. So COM is now an integral part of *Programming Visual C++*. Soon, most Windows programming will involve COM, so you'd better start learning it now. But where do you begin? You could start with the Microsoft Foundation Class classes for ActiveX Controls, Automation, and OLE, but as useful as those classes are, they obscure the real COM architecture. You've got to start with fundamental theory, and that includes COM and something called an interface.

The Component Object Model

COM is an "industry-standard" software architecture supported by Microsoft, Digital Equipment Corporation, and many other companies. It's by no means the only standard. Indeed, it competes directly against other standards, such as Corba from the Open Software Foundation (OSF). Some people are working to establish interoperability between COM and other architectures, but my guess is that COM will become the leading standard.

The Problem That COM Solves

The "problem" is that there's no standard way for Windows program modules to communicate with one another. "But," you say "what about the DLL with its exported functions, Dynamic Data Exchange (DDE), the Windows Clipboard, and the Windows API itself, not to mention legacy standards such as VBX and OLE 1? Aren't they good enough?" Well, no. You can't build an object-oriented operating system for the future out of these ad hoc, unrelated standards. With the Component Object Model, however, you can, and that's precisely what Microsoft is doing.

The Essence of COM

What's wrong with the old standards? A lot. The Windows API has too large a programming "surface area"—more than 350 separate functions. VBXs don't work in the 32-bit world. DDE comes with a complicated system of applications, topics, and items. How you call a DLL is totally application-specific. COM provides a unified, expandable, object-oriented communications protocol for Windows that already supports the following features:

- A standard, language-independent way for a Win32 client EXE to load and call a Win32 DLL
- A general-purpose way for one EXE to control another EXE on the same computer (the DDE replacement)
- A replacement for the VBX control, called an ActiveX control
- A powerful new way for application programs to interact with the operating system
- Expansion to accommodate new protocols such as Microsoft's OLE DB database interface
- The distributed COM (DCOM) that allows one EXE to communicate with another EXE residing on a different computer, even if the computers use different microprocessor-chip families

So what is COM? That's an easier question to ask than to answer. At DevelopMentor (a training facility for software developers), the party line is that "COM is love." That is, COM is a powerful integrating technology that allows you to mix all sorts of disparate software parts together at runtime. COM allows developers to write software that runs together regardless of issues such as thread-awareness and language choice.

COM is a protocol that connects one software module with another and then drops out of the picture. After the connection is made, the two modules can communicate through a mechanism called an interface. Interfaces require no statically or dynamically linked entry points or hard-coded addresses other than the few general-purpose COM functions that start the communication process. An interface (more precisely, a COM interface) is a term that you'll be seeing a lot of.

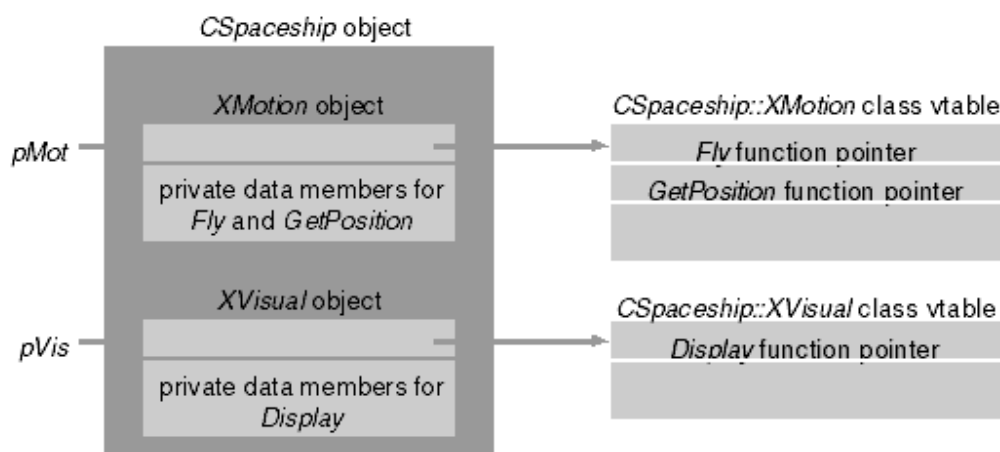
3. Explain COM interface with an example?- space ship example.

What Is a COM Interface?

Before digging into the topic of interfaces, let's re-examine the nature of inheritance and polymorphism in normal C++. We'll use a planetary-motion simulation (suitable for NASA or Nintendo) to illustrate C++ inheritance and polymorphism. Imagine a

spaceship that travels through our solar system under the influence of the sun's gravity. In ordinary C++, you could declare a *CSpaceship* class and write a constructor that sets the spaceship's initial position and acceleration. Then you could write a nonvirtual member function named *Fly* that implemented Kepler's laws to model the movement of the spaceship from one position to the next—say, over a period of 0.1 second. You could also write a *Display* function that painted an image of the spaceship in a window. The most interesting feature of the *CSpaceship* class is that the interface of the C++ class (the way the client talks to the class) and the implementation are tightly bound. One of the main goals of COM is to separate a class's interface from its implementation.

If we think of this example within the context of COM, the spaceship code could exist as a separate EXE or DLL (the component), which is a COM module. In COM the simulation manager (the client program) can't call *Fly* or any *CSpaceship* constructor directly: COM provides only a standard global function to gain access to the spaceship object, and then the client and the object use interfaces to talk to one another. Before we tackle real COM, let's build a COM simulation in which both the component and the client code are statically linked in the same EXE file. For our standard global function, we'll invent a function named *GetClassObject*.



If you want to map this process back to the way MFC works, you can look at *CRuntimeClass*, which serves as a class object for *CObject*-based classes. A class object is a meta-class (either in concept or in form).

In this COM simulation, clients will use this global single abstract function (*GetClassObject*) for objects of a particular class. In real COM, clients would get a class object first and then ask the class object to manufacture the real object in much the same way MFC does dynamic creation. *GetClassObject* has the following three parameters:

```
BOOL GetClassObject(int nClsid, int nIid, void** ppvObj);
```

The first *GetClassObject* parameter, *nClsid*, is a 32-bit integer that uniquely identifies the *CSpaceship* class. The second parameter, *nIid*, is the unique identifier of the interface that

we want. The third parameter is a pointer to an interface to the object. Remember that we're going to be dealing with interfaces now, (which are different from classes). As it turns out, a class can have several interfaces, so the last two parameters exist to manage interface selection. The function returns *TRUE* if the call is successful.

In C++, interfaces are declared as C++ *structs*, often with inheritance; in C, they're declared as C *typedef structs* with no inheritance. In C++, the compiler generates vtables for your derived classes; in C, you must "roll your own" vtables, and that gets tedious. It's important to realize, however, that in neither language do the interface declarations have data members, constructors, or destructors. Therefore, you can't rely on the interface having a virtual destructor—but that's not a problem because you never invoke a destructor for an interface.

The *IUnknown* Interface and the *QueryInterface* Member Function

Let's get back to the problem of how to obtain your interface pointers in the first place. COM declares a special interface named *IUnknown* for this purpose. As a matter of fact, all interfaces are derived from *IUnknown*, which has a pure virtual member function, *QueryInterface*, that returns an interface pointer based on the interface ID you feed it.

Once the interface mechanisms are hooked up, the client needs to get an *IUnknown* interface pointer (at the very least) or a pointer to one of the derived interfaces. Here is the new interface hierarchy, with *IUnknown* at the top:

```
struct IUnknown
{
    virtual BOOL QueryInterface(int nIid, void** ppvObj) = 0;
};

struct IMotion : public IUnknown
{
    virtual void Fly() = 0;
    virtual int& GetPosition() = 0;
};

struct IVisual : public IUnknown
{
    virtual void Display() = 0;
};
```

To satisfy the compiler, we must now add *QueryInterface* implementations in both *CSpaceship::XMotion* and *CSpaceship::XVisual*. What do the vtables look like after this is done? For each derived class, the compiler builds a vtable with the base class function pointers on top, as shown here.

GetClassObject can get the interface pointer for a given *CSpaceship* object by getting the address of the corresponding embedded object. Here's the code for the *QueryInterface* function in *XMotion*:

```
BOOL CSpaceship::XMotion::QueryInterface(int nIid,
                                         void** ppvObj)
{
    METHOD_PROLOGUE(CSpaceship, Motion)
    switch (nIid) {
        case IID_IUnknown:
        case IID_IMotion:
            *ppvObj = &pThis->m_xMotion;
            break;
        case IID_IVisual:
            *ppvObj = &pThis->m_xVisual;
            break;
        default:
            *ppvObj = NULL;
            return FALSE;
    }
    return TRUE;
}
```

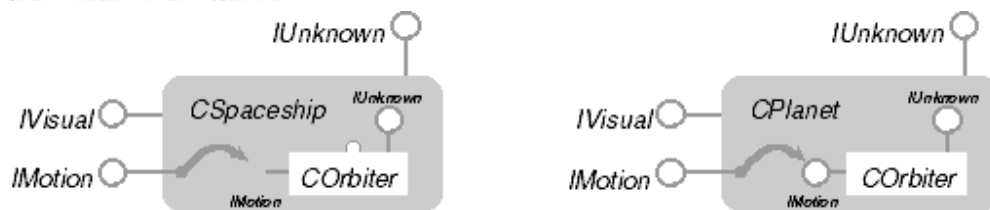
Because *IMotion* is derived from *IUnknown*, an *IMotion* pointer is a valid pointer if the caller asks for an *IUnknown* pointer.

The COM standard demands that *QueryInterface* return exactly the same *IUnknown* pointer value for *IID_IUnknown*, no matter which interface pointer you start with. Thus, if two *IUnknown* pointers match, you can assume that they refer to the same object. *IUnknown* is sometimes known as the "void*" of COM because it represents the object's identity.

4. Explain about Containment and Aggregation vs. Inheritance

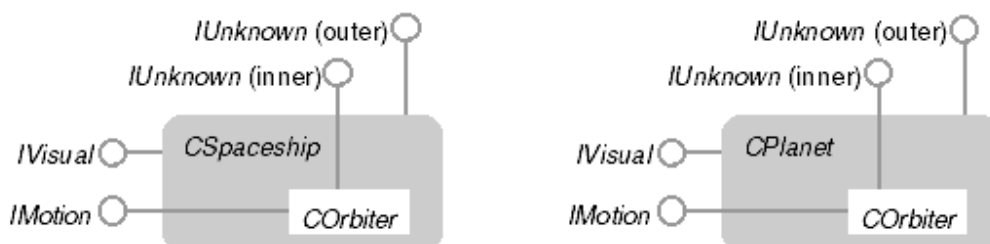
In normal C++ programming, you frequently use inheritance to factor out common behavior into a reusable base class. The *CPersistentFrame* class) is an example of reusability through inheritance.

COM uses containment and aggregation instead of inheritance. Let's start with containment. Suppose you extended the spaceship simulation to include planets in addition to spaceships. Using C++ by itself, you would probably write a *COrbiter* base class that encapsulated the laws of planetary motion. With COM, you would have "outer" *CSpaceship* and *CPlanet* classes plus an "inner" *COrbiter* class. The outer classes would implement the *IVisual* interface directly, but those outer classes would delegate their *IMotion* interfaces to the inner class. The result would look something like this.



Note that the *CO orbiter* object doesn't know that it's inside a *CS spaceship* or *CP planet* object, but the outer object certainly knows that it has a *CO orbiter* object embedded inside. The outer class needs to implement all its interface functions, but the *IMotion* functions, including *QueryInterface*, simply call the same *IMotion* functions of the inner class.

A more complex alternative to containment is aggregation. With aggregation, the client can have direct access to the inner object's interfaces. Shown here is the aggregation version of the space simulation.



The orbiter is embedded in the spaceship and planet, just as it was in the containment case. Suppose the client obtains an *IVisual* pointer for a spaceship and then calls *QueryInterface* for an *IMotion* pointer. Using the outer *IUnknown* pointer will draw a blank because the *CS spaceship* class doesn't support *IMotion*. The *CS spaceship* class keeps track of the inner *IUnknown* pointer (of its embedded *CO orbiter* object), so the class uses that pointer to obtain the *IMotion* pointer for the *CO orbiter* object.

Now suppose the client obtains an *IMotion* pointer and then calls *QueryInterface* for *IVisual*. The inner object must be able to navigate to the outer object, but how? Take a close look at the *CreateInstance* call back in Figure 24-10. The first parameter is set to *NULL* in that case. If you are creating an aggregated (inner) object, you use that parameter to pass an *IUnknown* pointer for the outer object that you have already created. This pointer is called the controlling unknown. The *CO orbiter* class saves this pointer in a data member and then uses it to call *QueryInterface* for interfaces that the class itself doesn't support.

The MFC library supports aggregation. The *CCmdTarget* class has a public data member *m_pOuterUnknown* that holds the outer object's *IUnknown* pointer (if the object is aggregated). The *CCmdTarget* member functions *ExternalQueryInterface*, *ExternalAddRef*, and *ExternalRelease* delegate to the outer *IUnknown* if it exists. Member functions *InternalQueryInterface*, *InternalAddRef*, and *InternalRelease* do not delegate. See Technical Note #38 in the online documentation for a description of the MFC macros that support aggregation.

6. Explain about MFC Drag and Drop

Drag and drop was the ultimate justification for the data object code you've been looking at. OLE supports this feature with its *IDropSource* and *IDropTarget* interfaces plus some library code that manages the drag-and-drop process. The MFC library offers good drag-and-drop support at the view level, so we'll use it. Be aware that drag-and-drop transfers are immediate and independent of the clipboard. If the user cancels the operation, there's no "memory" of the object being dragged.

Drag-and-drop transfers should work consistently between applications, between windows of the same application, and within a window. When the user starts the operation, the cursor should change to an arrow_rectangle combination. If the user holds down the Ctrl key, the cursor turns into a plus sign (+), which indicates that the object is being copied rather than moved.

MFC also supports drag-and-drop operations for items in compound documents. This is the next level up in MFC OLE support, and it's not covered in this chapter. Look up the OCLIENT example in the online documentation under Visual C++ Samples.

The Source Side of the Transfer

When your source program starts a drag-and-drop operation for a data object, it calls *COleDataSource::DoDragDrop*. This function internally creates an object of MFC class *COleDropSource*, which implements the *IOleDropSource* interface. *DoDragDrop* is one of those functions that don't return for a while. It returns when the user drops the object or cancels the operation or when a specified number of milliseconds have elapsed.

If you're programming drag-and-drop operations to work with a *CRectTracker* object, you should call *DoDragDrop* only when the user clicks inside the tracking rectangle, not on its border. *CRectTracker::HitTest* gives you that information. When you call *DoDragDrop*, you need to set a flag that tells you whether the user is dropping the object into the same view (or document) that it was dragged from.

The Destination Side of the Transfer

If you want to use the MFC library's view class drag-and-drop support, you must add a data member of class *COleDropTarget* to your derived view class. This class implements the *IDropTarget* interface, and it holds an *IDropSource* pointer that links back to the *COleDropSource* object. In your view's *OnInitialUpdate* function, you call the *Register* member function for the embedded *COleDropTarget* object.

After you have made your view a drop target, you must override four *CView* virtual functions, which the framework calls during the drag-and-drop operation. Here's a summary of what they should do, assuming that you're using a tracker.

- OnDragEnter* Adjusts the focus rectangle and then calls *OnDragOver*
- OnDragOver* Moves the dotted focus rectangle and sets the drop effect (determines cursor shape)
- OnDragLeave* Cancels the transfer operation; returns the rectangle to its original position and size
- OnDrop* Adjusts the focus rectangle and then calls the *DoPaste* helper function to get formats from the data object

The Drag-and-Drop Sequence

Figure 26-4 illustrates the MFC drag-and-drop process.

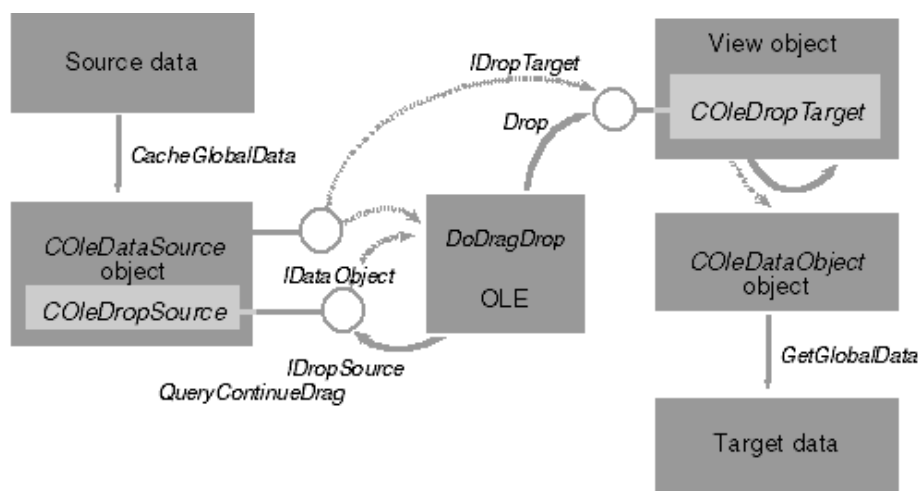


Figure 26-4. MFC OLE drag-and-drop processing.

Here's a summary of what's going on:

1. User presses the left mouse button in the source view window.
2. Mouse button handler calls *CRectTracker::HitTest* and finds out that the cursor was inside the tracker rectangle.
3. Handler stores formats in a *COleDataSource* object.
4. Handler calls *COleDataSource::DoDragDrop* for the data source.
5. User moves the cursor to the view window of the target application.
6. OLE calls *IDropTarget::OnDragEnter* and *OnDragOver* for the *COleDropTarget* object, which calls the corresponding virtual functions in the target's view. The *OnDragOver* function is passed a *COleDataObject* pointer for the source object, which the target tests for a format it can understand.
7. *OnDragOver* returns a drop effect code, which OLE uses to set the cursor.

8. OLE calls *IDataSource::QueryContinueDrag* on the source side to find out whether the drag operation is still in progress. The MFC *COleDataSource* class responds appropriately.
9. User releases the mouse button to drop the object in the target view window.
10. OLE calls *IDropTarget::OnDrop*, which calls *OnDrop* for the target's view. Because *OnDrop* is passed a *COleDataObject* pointer, it can retrieve the desired format from that object.
11. When *OnDrop* returns in the target program, *DoDragDrop* can return in the source program.

7. Explain about Container-Component Interactions

Analyzing the component and the container separately won't help you to understand fully how they work. You must watch them working together to understand their interactions. Let's reveal the complexity one step at a time. Consider first that you have a container EXE and a component EXE, and the container must manage the component by means of OLE interfaces.

The client program called *CoGetClassObject* and *IClassFactory::CreateInstance* to load the spaceship component and to create a spaceship object, and then it called *QueryInterface* to get *IMotion* and *IVisual* pointers. An embedding container program works the same way that the space simulation client works. It starts the component program based on the component's class ID, and the component program constructs an object. Only the interfaces are different.

Figure 28-2 shows a container program looking at a component. You've already seen all the interfaces except one—*IOleObject*.

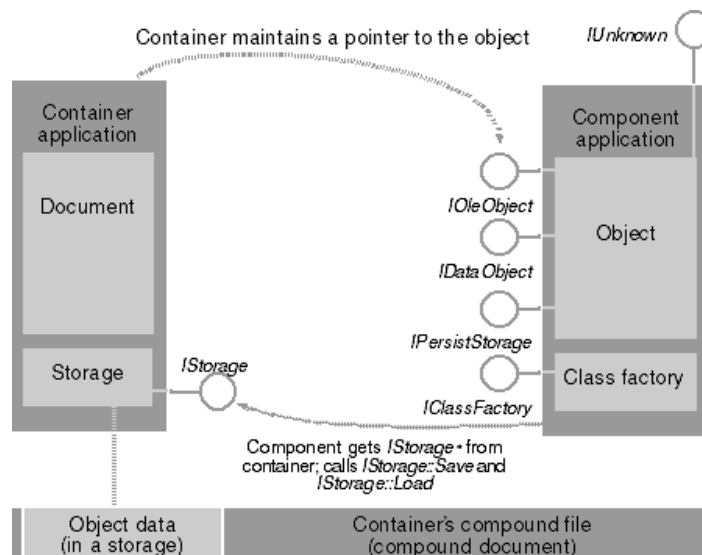


Figure 28-2. A container program's view of the component.

Using the Component's *IOleObject* Interface

Loading a component is not the same as activating it. Loading merely starts a process, which then sits waiting for further instructions. If the container gets an *IOleObject* pointer to the component

object, it can call the *DoVerb* member function with a verb parameter such as *OLEIVERB_SHOW*. The component should then show its main window and act like a Windows-based program. If you look at the *IObject::DoVerb* description, you'll see an *IObjectSite** parameter. We'll consider client sites shortly, but for now you can simply set the parameter to *NULL* and most components will work okay.

Another important *IObject* function, *Close*, is useful at this stage. As you might expect, the container calls *Close* when it wants to terminate the component program. If the component process is currently servicing one embedded object (as is the case with MFC components), the process exits.

Loading and Saving the Component's Native Data—Compound Documents

Figure 28-2 demonstrates that the container manages a storage through an *IStorage* pointer and that the component implements *IPersistStorage*. That means that the component can load and save its native data when the container calls the *Load* and *Save* functions of *IPersistStorage*. You've seen the *IStorage* and *IPersistStorage* interfaces used in [Chapter 27](#), but this time the container is going to save the component's class ID in the storage. The container can read the class ID from the storage and use it to start the component program prior to calling *IPersistStorage::Load*.

Actually, the storage is very important to the embedded object. Just as a virus needs to live in a cell, an embedded object needs to live in a storage. The storage must always be available because the object is constantly loading and saving itself and reading and writing temporary data.

A compound document appears at the bottom of Figure 28-2. The container manages the whole file, but the embedded components are responsible for the storages inside it. There's one main storage for each embedded object, and the container doesn't know or care what's inside those storages.

Clipboard Data Transfers

There's actually less here than meets the eye. The only thing inside the *CF_EMBEDDEDOBJECT* format is an *IStorage* pointer. The clipboard copy program verifies that *IPersistStorage::Save* has been called to save the embedded object's data in the storage, and then it passes off the *IStorage* pointer in a data object. The clipboard paste program gets the class ID from the source storage, loads the component program, and then calls *IPersistStorage::Load* to load the data from the source storage.

The data objects for the clipboard are generated as needed by the container program. The component's *IDataObject* interface isn't used for transferring the objects' native data.

Getting the Component's Metafile

You already know that a component program is supposed to draw in a metafile and that a container is supposed to play it. But how does the component deliver the metafile? That's what the *IDataObject* interface, shown in Figure 28-2, is for. The container calls *IDataObject::GetData*, asking for a *CF_METAFILEPICT* format. But wait a minute. The container is supposed to get the metafile even if the component program isn't running. So now you're ready for the next complexity level.

The Role of the In-Process Handler

If the component program is running, it's in a separate process. Sometimes it's not running at all. In either case, the OLE32 DLL is linked into the container's process. This DLL is known as the object handler.

Figure 28-3 shows the new picture. The handler communicates with the component over the RPC link, marshaling all interface function calls. But the handler does more than act as the component's proxy for marshaling; it maintains a cache that contains the component object's metafile. The handler saves and loads the cache to and from storage, and it can fill the cache by calling the component's *IDataObject::GetData* function.

When the container wants to draw the metafile, it doesn't do the drawing itself; instead, it asks the handler to draw the metafile by calling the handler's *IViewObject2::Draw* function. The handler tries to satisfy as many container requests as it can without bothering the component program. If the handler needs to call a component function, it takes care of loading the component program if it is not already loaded.

The *IViewObject2* interface is an example of OLE's design evolution. Someone decided to add a new function—in this case, *GetExtent*—to the *IViewObject* interface. *IViewObject2* is derived from *IViewObject* and contains the new function. All new components should implement the new interface and should return an *IViewObject2* pointer when *QueryInterface* is called for either *IID_IViewObject* or *IID_IViewObject2*. This is easy with the MFC library because you write two interface map entries that link to the same nested class.

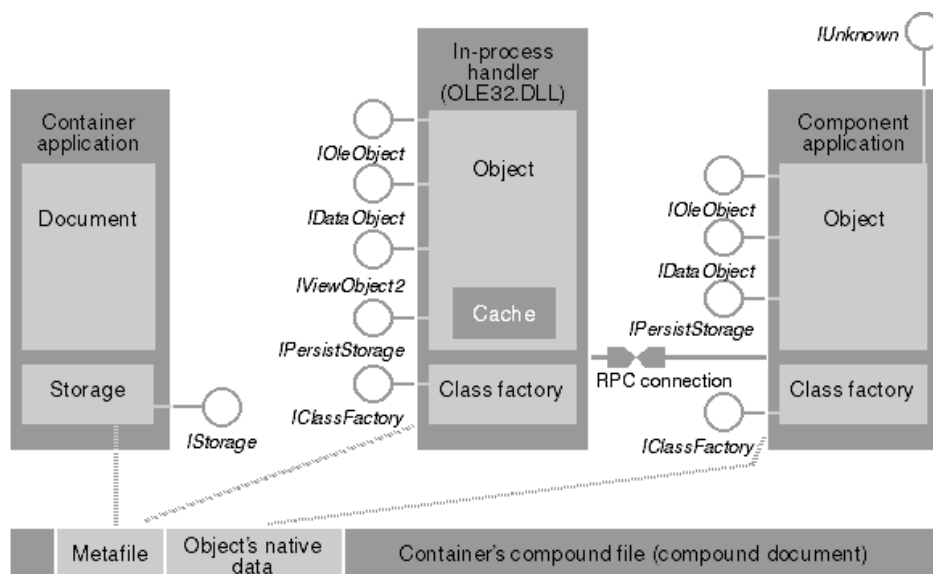


Figure 28-3. The in-process handler and the component.

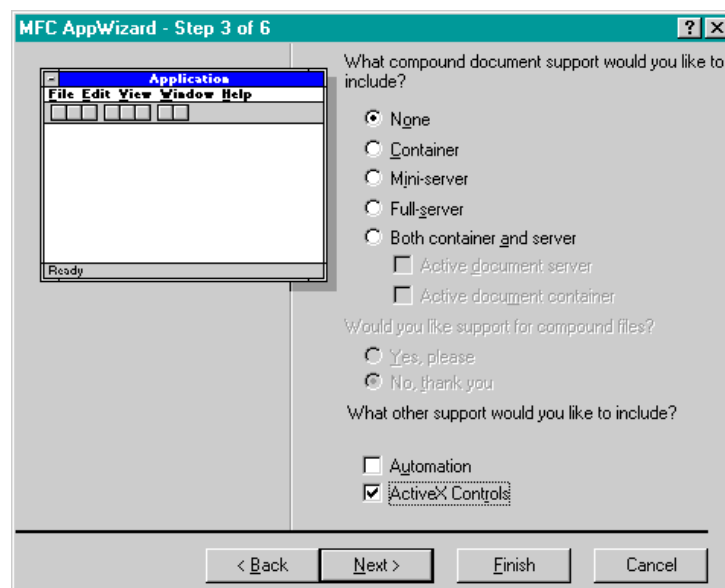
Figure 28-3 shows both object data and metafile data in the object's storage. When the container calls the handler's *IPersistStorage::Save* function, the handler writes the cache (containing the metafile) to the storage and then calls the component's *IPersistStorage::Save* function, which writes the object's native data to the same storage. The reverse happens when the object is loaded.

8. Implement calendar control using ACTIVEX

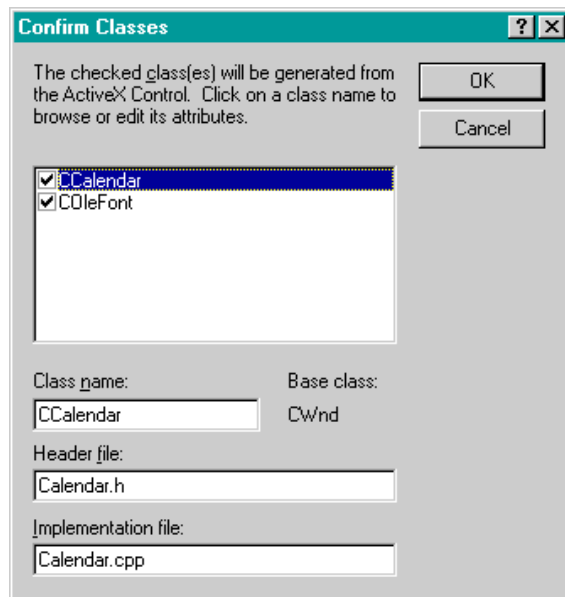
The EX08A Example—An ActiveX Control Dialog Container

Now it's time to build an application that uses a Calendar control in a dialog. Here are the steps to create the EX08A example:

1. **Verify that the Calendar control is registered.** If the control does not appear in the Visual C++ Gallery's Registered ActiveX Controls page, copy the files MSCal.ocx, MSCal.hlp, and MSCal.cnt to your system directory and register the control by running the REGCOMP program.
2. **Run AppWizard to produce \vcpp32\ex08a\ex08a.** Accept all of the default settings but two: select Single Document and deselect Printing And Print Preview. In the AppWizard Step 3 dialog, make sure the ActiveX Controls option is selected, as shown below.



3. **Install the Calendar control in the EX08A project.** Choose Add To Project from Visual C++'s Project menu, and then choose Components And Controls. Choose Registered ActiveX Controls, and then choose Calendar Control 8.0. ClassWizard generates two classes in the EX08A directory, as shown here.



4. **Edit the Calendar control class to handle help messages.** Add Calendar.cpp to the following message map code:

```
BEGIN_MESSAGE_MAP(CCalendar, CWnd)
    ON_WM_HELPINFO()
END_MESSAGE_MAP()
```

In the same file, add the *OnHelpInfo* function:

```
BOOL CCalendar::OnHelpInfo(HELPINFO* pHelpInfo)
{
    // Edit the following string for your system
    ::WinHelp(GetSafeHwnd(), "c:\\winnt\\system32\\mscal.hlp",
        HELP_FINDER, 0);
    return FALSE;
}
```

In Calendar.h, add the function prototype and declare the message map:

```
protected:
    afx_msg BOOL OnHelpInfo(HELPINFO* pHelpInfo);
    DECLARE_MESSAGE_MAP()
```

The *OnHelpInfo* function is called if the user presses the F1 key when the Calendar control has the input focus. We have to add the message map code by hand because ClassWizard doesn't modify generated ActiveX classes.

The *ON_WM_HELPINFO* macro maps the WM_HELP message, which is new to Microsoft Windows 95 and Microsoft Windows NT 4.0. You can use *ON_WM_HELPINFO* in any view or dialog class and then code the handler to

activate any help system. [Chapter 21](#) describes the MFC context-sensitive help system, some of which predates the WM_HELP message.

5. **Use the dialog editor to create a new dialog resource.** Choose Resource from Visual C++'s Insert menu, and then choose Dialog. The dialog editor assigns the ID *IDD_DIALOG1* to the new dialog. Next change the ID to *IDD_ACTIVEXDIALOG*, change the dialog caption to *ActiveX Dialog*, and set the dialog's Context Help property (on the More Styles page). Accept the default OK and Cancel buttons with the IDs *IDOK* and *IDCANCEL*, and then add the other controls as shown in Figure 8-1. Make the Select Date button the default button. Drag the Calendar control from the control palette. Then set an appropriate tab order. Assign control IDs as shown in the following table.

Control	ID
Calendar control	<i>IDC_CALENDAR1</i>
Select Date button	<i>IDC_SELECTDATE</i>
Edit control	<i>IDC_DAY</i>
Edit control	<i>IDC_MONTH</i>
Edit control	<i>IDC_YEAR</i>
Next Week button	<i>IDC_NEXTWEEK</i>

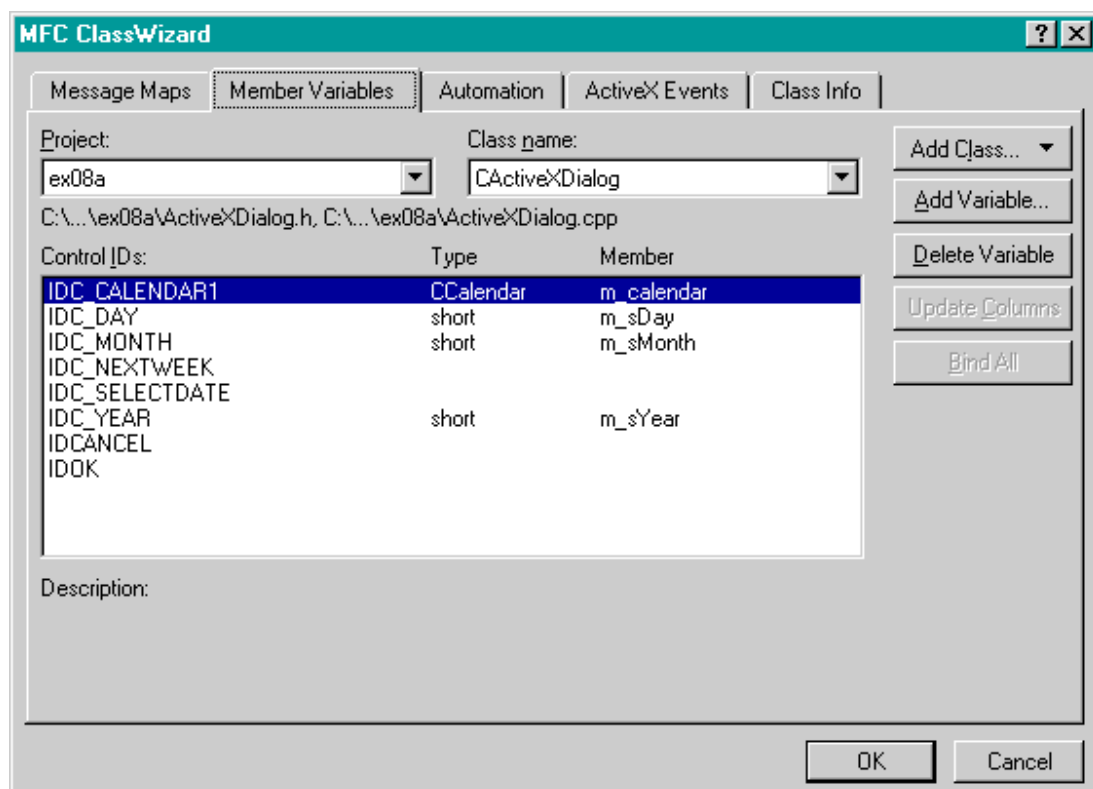
6. **Use ClassWizard to create the *CActiveXDialog* class.** If you run ClassWizard directly from the dialog editor window, it will know that you want to create a *CDialog*-derived class based on the *IDD_ACTIVEXDIALOG* template. Simply accept the default options, and name the class *CActiveXDialog*.

Click on the ClassWizard Message Maps tab, and then add the message handler functions shown in the table below. To add a message handler function, click on an object ID, click on a message, and click the Add Function button. If the Add Member Function dialog box appears, type the function name and click the OK button.

Object ID	Message	Member Function
<i>CActiveXDialog</i>	WM_INITDIALOG	<i>OnInitDialog</i> (virtual function)
<i>IDC_CALENDAR1</i>	NewMonth (event)	<i>OnNewMonthCalendar1</i>
<i>IDC_SELECTDATE</i>	BN_CLICKED	<i>OnSelectDate</i>
<i>IDC_NEXTWEEK</i>	BN_CLICKED	<i>OnNextWeek</i>
<i>IDOK</i>	BN_CLICKED	<i>OnOK</i> (virtual function)

7. Use **ClassWizard** to add data members to the *CActiveXDialog* class. Click on the Member Variables tab, and then add the data members as shown in the illustration below.

You might think that the ClassWizard ActiveX Events tab is for mapping ActiveX control events in a container. That's not true: it's for ActiveX control developers who are defining events for a control.



8. **Edit the CActiveXDialog class.** Add the *m_varValue* and *m_BackColor* data members, and then edit the code for the five handler functions *OnInitDialog*, *OnNewMonthCalendar1*, *OnSelectDate*, *OnNextWeek*, and *OnOK*. Figure 8-2 shows all the code for the dialog class, with new code in boldface.

ACTIVEXDIALOG.H

```
// {AFX_INCLUDES()
#include "calendar.h"
// }AFX_INCLUDES
#if
!defined(AFX_ACTIVEXDIALOG_H__1917789D_6F24_11D0_8FD9_00C04FC2A0C2__INCLUDED_)
#define AFX_ACTIVEXDIALOG_H__1917789D_6F24_11D0_8FD9_00C04FC2A0C2__INCLUDED_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// ActiveXDialog.h : header file
//

/////////////////////////////////////////////////////////////////
// CActiveXDialog dialog
class CActiveXDialog : public CDialog
{
// Construction
public:
    CActiveXDialog(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
   //{{AFX_DATA(CActiveXDialog)
    enum { IDD = IDD_ACTIVEXDIALOG };
    CCalendar    m_calendar;
    short        m_sDay;
    short        m_sMonth;
    short        m_sYear;
    //}}AFX_DATA
    COleVariant m_varValue;
    unsigned long m_BackColor;

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CActiveXDialog)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
                                                    // support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
   //{{AFX_MSG(CActiveXDialog)
    virtual BOOL OnInitDialog();
    afx_msg void OnNewMonthCalendar1();
    afx_msg void OnSelectDate();
    afx_msg void OnNextWeek();
    virtual void OnOK();
    DECLARE_EVENTSINK_MAP()
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional
// declarations immediately before the previous line.

#endif //
#ifndef(AFX_ACTIVEXDIALOG_H__1917789D_6F24_11D0_8FD9_00C04FC2A0C2__INCLUDED_)
```

ACTIVEXDIALOG.CPP

```
// ActiveXDialog.cpp : implementation file
//

#include "stdafx.h"
#include "ex08a.h"
#include "ActiveXDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CActiveXDialog dialog

CActiveXDialog::CActiveXDialog(CWnd* pParent /*=NULL*/)
: CDialog(CActiveXDialog::IDD, pParent)
{
   //{{AFX_DATA_INIT(CActiveXDialog)
    m_sDay = 0;
    m_sMonth = 0;
    m_sYear = 0;
    //}}AFX_DATA_INIT
    m_BackColor = 0x8000000F;
}

void CActiveXDialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CActiveXDialog)
    DDX_Control(pDX, IDC_CALENDAR1, m_calendar);
    DDX_Text(pDX, IDC_DAY, m_sDay);
    DDX_Text(pDX, IDC_MONTH, m_sMonth);
    DDX_Text(pDX, IDC_YEAR, m_sYear);
    //}}AFX_DATA_MAP
    DDX_OcColor(pDX, IDC_CALENDAR1, DISPID_BACKCOLOR, m_BackColor);
}

BEGIN_MESSAGE_MAP(CActiveXDialog, CDialog)
   //{{AFX_MSG_MAP(CActiveXDialog)
    ON_BN_CLICKED(IDC_SELECTDATE, OnSelectDate)
    ON_BN_CLICKED(IDC_NEXTWEEK, OnNextWeek)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CActiveXDialog message handlers

BEGIN_EVENTSINK_MAP(CActiveXDialog, CDialog)
   //{{AFX_EVENTSINK_MAP(CActiveXDialog)
```



```

        ON_EVENT(CActiveXDialog, IDC_CALENDAR1, 3 /* NewMonth */,
OnNewMonthCalendar1, VTS_NONE)
        //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()

BOOL CActiveXDialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    m_calendar.SetValue(m_varValue); // no DDX for VARIANTS
    return TRUE; // return TRUE unless you set the focus to a control
                // EXCEPTION: OCX Property Pages should return FALSE
}

void CActiveXDialog::OnNewMonthCalendar1()
{
    AfxMessageBox("EVENT: CActiveXDialog::OnNewMonthCalendar1");
}

void CActiveXDialog::OnSelectDate()
{
    CDataExchange dx(this, TRUE);
    DDX_Text(&dx, IDC_DAY, m_sDay);
    DDX_Text(&dx, IDC_MONTH, m_sMonth);
    DDX_Text(&dx, IDC_YEAR, m_sYear);
    m_calendar.SetDay(m_sDay);
    m_calendar.SetMonth(m_sMonth);
    m_calendar.SetYear(m_sYear);
}

void CActiveXDialog::OnNextWeek()
{
    m_calendar.NextWeek();
}

void CActiveXDialog::OnOK()
{
    CDialog::OnOK();
    m_varValue = m_calendar.GetValue(); // no DDX for VARIANTS
}

```

9. **Figure 8-2.** Code for the CActiveXDialog class.

10. The *OnSelectDate* function is called when the user clicks the Select Date button. The function gets the day, month, and year values from the three edit controls and transfers them to the control's properties. ClassWizard can't add DDX code for the BackColor property, so you must add it by hand. In addition, there's no DDX code for VARIANT types, so you must add code to the *OnInitDialog* and *OnOK* functions to set and retrieve the date with the control's Value property.
11. **Connect the dialog to the view.** Use ClassWizard to map the WM_LBUTTONDOWN message, and then edit the handler function as follows:

```

void CEx08aView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CActiveXDialog dlg;

```

```
dlg.m_BackColor = RGB(255, 251, 240); // light yellow
COleDateTime today = COleDateTime::GetCurrentTime();
dlg.m_varValue = COleDateTime(today.GetYear(), today.GetMonth(),
                             today.GetDay(), 0, 0, 0);

if (dlg.DoModal() == IDOK) {
    COleDateTime date(dlg.m_varValue);
    AfxMessageBox(date.Format("%B %d, %Y"));
}
}
```

The code sets the background color to light yellow and the date to today's date, displays the modal dialog, and reports the date returned by the Calendar control. You'll need to include `ActiveXDialog.h` in `ex08aView.cpp`.

12. **Edit the virtual *OnDraw* function in the file `ex08aView.cpp`.** To prompt the user to press the left mouse button, replace the code in the view class *OnDraw* function with this single line:

```
pDC->TextOut(0, 0, "Press the left mouse button here.");
```

13. **Build and test the EX08A application.** Open the dialog, enter a date in the three edit controls, and then click the Select Date button. Click the Next Week button. Try moving the selected date directly to a new month, and observe the message box that is triggered by the `NewMonth` event. Watch for the final date in another message box when you click OK. Press the F1 key for help on the Calendar control

Unit 5

1.Explain about Process and Threads

Process:

A process is an executing instance of an application. A process maintains its own memory and other system resources. It also has structures that keep track of code as well as data used by the application. Both WIN95 and WINNT allow multiple processes to run simultaneously. The operating system takes care of switching between various running processes at specific time intervals. The operating system allocates resources to the process.

THREAD:

A thread is a path of execution within a process. It is the smallest unit of code that the operating system can execute. Every process has a minimum of one thread of execution. When you start an application, the operating system creates a process and begins executing the primary thread of that process. When this thread terminates, so does the process.

A thread cannot request for resources for resources nor will it be allocated any resources, the thread has to use the resources of the process.

Applications are of two types – Single threaded applications and multithreaded applications.

SINGLE THREAD APPLICATIONS:

Applications that have only one thread are called single threaded applications. This thread is also known as the main thread of the application. In these applications, all the processing is done in a linear fashion. Such applications take more time for execution also. The applications cannot perform any background processing.

MULTI-THREADED APPLICATIONS:

Applications that use more than one thread are called multithreaded applications. You can create additional threads in your applications if you wish. Such applications make full use of the resources even if the application is running in a multiprocessor environment.

Some illustrations of a multithreaded applications are as follows:

- Printing multiple copies of a document while you continue to edit your document.
- Recalculating spreadsheet entries automatically when a cell value is updated.
- Compiling and building your application while you continue to work in the Microsoft Developer studio.

HOW WIN32 VIEWS PROCESSES AND THREADS:

A process is a program that is loaded into memory and prepared for execution. Each process has a private virtual address space. A process consists of the code, data and other system resources, such as files, pipes that are accessible to the threads of the process.

Each process is started with a single thread, but additionally executing threads can be created.

A thread can execute any part of the program's code, including a part executed by another thread. Threads are the basic entity to which the operating system allocates CPU time. All threads of a

process share the virtual address space and can access the global variables and system resources of the process.

In Windows, the Win32API is designed for preemptive multitasking, this means that the system allocates small slices of CPU time among the competing threads. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When the system switches from one thread to another, it saves the context of the suspended thread and restores the saved context of the next thread in the queue.

Because each time slice is small, it appears that multiple threads are executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors.

On a single processor system, however, using multiple threads does not result in more instructions being executed. In fact, the system can slow down if it is forced to keep track of too many threads.

To the application developer, the advantage of multitasking is the ability to create applications that use more than one process and to create processes that use more than one thread of execution.

TYPES OF THREAD:

MFC supports two kinds of threads:

- User-interface threads
- Worker threads

USER-INTERFACE THREADS:

User-interface threads are used to handle user input and respond to events generated by the user as well as windows messages. User-interface threads have message handler functions. Example the application object. The CWinThread class is used for thread processing in MFC. Thus, in our MFC applications the primary thread of execution is a user interface thread.

WORKER THREADS:

Worker threads are commonly used to complete tasks, such as recalculation and background printing, that do not require user input. Since worker threads do not handle user-inputs, they do not have any message handler functions associated with them. An application can create any number of worker threads.

FUNCTIONING OF THREADS:

All threads in MFC applications are represented by CWinThread objects. In most situations, you do not have to explicitly create these objects. Instead, you can call the AfxBeginThread function, which creates CWinThread object.

CREATING WORKER THREAD:

A worker thread is commonly used to handle background tasks that do not require user interaction. So they do not handle any messages. Programming using the worker threads includes the following steps:

- Implementing the controlling function.

- Starting the thread.

The controlling function specifies the code to be executed when the thread starts. In fact, when you start thread, the address of the controlling function is passed as a parameter to the thread. The `AfxBeginThread` function is used to create threads. This function is overloaded, one for user interface thread and other for worker thread.

AfxBeginThread Function:

Following is the syntax of the `AfxBeginThread` function to create a worker thread:

```
CWinThread* AfxBeginThread(AFX_THREADPROC pfnThreadProc, LPVOID pParam, int  
nPriority=THREAD_PRIORITY_NORMAL, UINT nStackSize=0, DWORD dwCreateFlags=0  
, LPSECURITY_ATTRIBUTES lpSecurityAttrs=NULL);
```

Where,

`pfnThreadProc` is the name of the function that will run the thread.

`pParam` is a parameter passed to the thread to the thread function. The data type of this pointer is `void *`, can pass any data type.

`nPriority` specifies the priority of the thread, default is normal priority.

`nStackSize` specifies the size of the stack for the threaded function.

`dwCreateFlags` additional flags that control thread creation.

`lpSecurityAttributes` any security attributes for the thread.

The following code segment illustrates the implementation of a worker thread.

```
CWinThread *pThread=AfxBeginThread(MyThreadProcedure, GetSafeHwnd(),  
THREAD_PRIORITY_NORMAL);
```

The first parameter is the address of the function to be executed in the thread. You can write any instructions in the controlling function and they will be executed by the thread.

The second parameter is the handle to the application window. The member function `GetSafeHwnd` of the `CWnd` class returns a handle to the application window. The thread uses this handle when it has to communicate to the main thread of the application.

The third parameter specifies the thread's priority code. This parameter is optional. By default it is `THREAD_PRIORITY_NORMAL`. All the symbols and macros regarding this processes and threads are defined in `winbase.h` file.

The function `AfxBeginThread` returns a pointer of the type `CWinThread` to the newly created thread object. You can use this pointer to suspend or resume the thread while your application is running using `SuspendThread` and `ResumeThread`, which are member functions of `CWinThread`.

SetThreadPriority:

Every thread has a base priority level. This priority level is determined by the thread's priority value and the priority of its application. A thread's priority value is specified at the time of creation of the thread. The priority of the application is determined at the operating system level.

The SetThreadPriority member function of the CWinThread class sets the priority value for a specified thread. Every running application has a runtime priority assigned to it by the operating system. This function takes an integer as the priority value. The following code segment illustrates the usage of the function.

```
BOOL status;  
Status= pThread->SetThreadPriority(THREAD_PRIORITY_NORMAL);
```

The following are the priority values that can be set using the function as follows:

- THREAD_PRIORITY_NORMAL—indicates normal priority.
- THREAD_PRIORITY_ABOVE_NORMAL—indicates one point above normal priority.
- THREAD_PRIORITY_BELOW_NORMAL—indicates one point below normal priority.
- THREAD_PRIORITY_HIGHEST—indicates two points above normal priority.
- THREAD_PRIORITY_LOWEST—indicates two points below normal priority.

CREATING USER INTERFACE THREADS:

A user-interface thread is commonly used to handle user input and respond to user events independently of other threads executing in other portions of the application. The main application thread is a user-interface thread and is derived from the CWinApp class. The application should make a call to the following overridden version of AfxBeginThread to create a user-interface thread.

```
CWinThread* AfxBeginThread(CRuntimeClass* pThreadClass, int nPriority= int  
THREAD_PRIORITY_NORMAL, UINT nStackSize =0 , DWORD dwCreateFlags =0 ,  
LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
```

Where,
pThreadClass is the RUNTIME_CLASS of an object derived from CWinThread.

nPriority specifies the priority of the thread, default is normal priority.

nStackSize specifies the size of the stack for the threaded function.

dwCreateFlags additional flags that control thread creation.

lpSecurityAttributes any security attributes for the thread.

The first thing you must do when creating a user-interface thread is derive a class from CWinThread. You must implement this class using the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros. The class must override certain functions and can override others. These functions are:

- ExitInstance
- InitInstance
- OnIdle – perform thread specific idle-time processing . Not usually overridden.

TERMINATING THREADS:

Two normal situations cause a thread to terminate—the controlling function exits or the thread is not be allowed to run to completion. Should the user wish to cancel the printing, however, the background printing thread would have to be terminated prematurely.

Normal Thread Termination:

For a worker thread, normal thread termination is simple, exit the controlling function and return value that signifies the reason for termination . Typically the return value 0 signifies successful completion.

For a user-interface thread , the process is just as simple-from within the user-interface thread, call ::PostQuitMessage function. The exit code 0 typically signifies successful completion.

Premature Thread Termination:

Terminating a thread prematurely is almost as simple- call AfxEndThread from within the thread. Pass the desired exit code as the only parameter. This stops execution of the thread, releases the thread's resources, detaches all DLLs attaches to the thread and deletes the thread object from memory.

2.RDBMS:

A relational database is a collection of related information organized into one or more tables. Tables are visually represented as a two dimensional array, wit rows representing records and columns representing fields.

DATABASE OPERATIONS:

- Query: Retrieval of information from a database. A query is often qualified by a filter(a WHERE clause).
- Join: Combination of two tables to form a new table. While joining two tables, you can set the criteria for selecting records.
- Update: Updation of one or more records in a single or multiple tables in a database. This changes the state a table.
- recordation: Addition of records to a table.
- Record Deletion: Removal of records from a table.

For performing these operations, all RDBMS –based packages provide a language called Structured Query Language(SQL).

SQL:

SQL stands for Structured Query Language . Most DBMS's support SQL. SQL standards are, however, constantly evolving , and SQL grammar varies between products. ODBC allows a user to connect to a data source and execute SQL statements on the data source.

DATABASE MANAGEMENT WITH ODBC:

In the previous sessions , we had seen the use of Document-View architecture to code applications in MFC. Now we will see how to apply the same to programs for a different kind of interface—The ODBC interface. The applications required fast access to individual records in a large database. Today Microsoft Windows programmers have a wide choice of programming DBMS, including Powersoft PowerBuilder, Borland Paradox, Microsoft Access and Microsoft FoxPro.

VC++ product contains two separate database access systems.
ODBC and DAO(Data Access Objects).

ODBC is based on a standardized version of SQL . With ODBC and SQL, you can write database access code that is independent of database products.

FEATURES OF ODBC:

ODBC offers a structured approach for the construction of such an application interface. There are two tasks that have to be provided for:

- Extracting database information.
- Importing data into the application.

It provides a standard interface that allows both application and providers of libraries to move data between applications and data sources.

ODBC Interface:

ODBC interface is an open, vendor –neutral interface for database connectivity that provides access to a variety of personal computer, minicomputer, and mainframe systems, including Windows-based systems and the Apple Macintosh.

The ODBC interface of MFC includes:

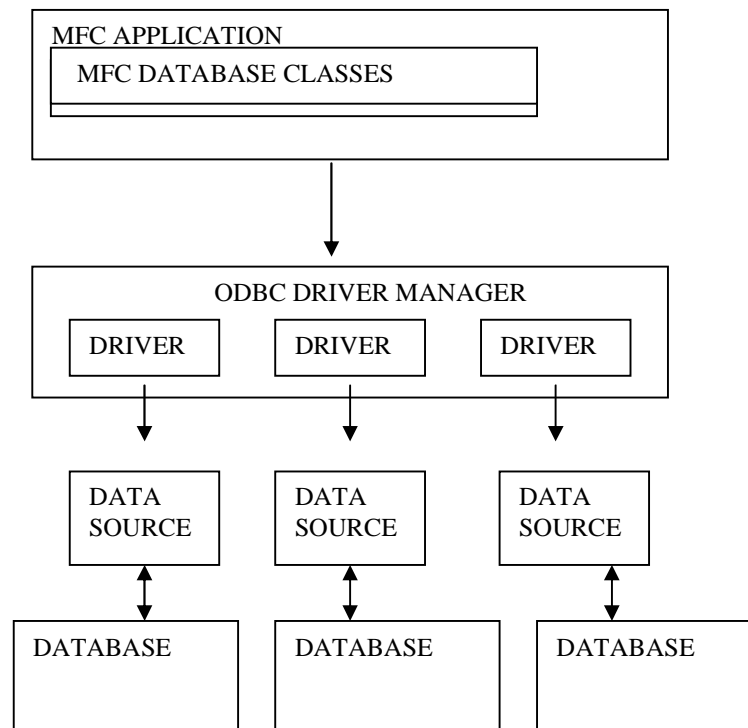
- A library of ODBC function calls that allow an application to logon and connect to a DBMS, execute SQL statements, and retrieve results.
- A standard set of error codes.
- A standard representation for data types.

ODBC Architecture:

ODBC has four components- The application, the driver manager, the drive and the data source.

- **Application:** This is the application program which perform all the processing and calls ODBC functions to submit statements and retrieve results.
- **Driver Manager:** This routine loads drivers on behalf of an application.

- **Driver:** This routine processes ODBC function calls, submits SQL requests to a specific data source and returns results to the application. If necessary, the driver modifies an application request so that the request conforms to the syntax supported by the associated DBMS.
- **Data Source:** This consists of the data the user wants to access and the associated operating system DBMS, and network platform used to access the DBMS.



DATA SOURCE:

A data source consists of the data a user wants to access, its associated DBMS, the platform on which the DBMS resides, and the network used to access that platform. Each data source requires that a driver provide certain information in order to connect to it.

ODBC CLASSES OF MFC:

An ODBC data source contains a specific set of data, the information required to access that data, and the location of the data source, which data can be described using a data source name.

To access data provided by a data source, your program must first establish a connection to the data source. All data access is managed through that connection. Data source connections are encapsulated by the class `CDatabase`.

CDatabase Class:

The `CDatabase` class represents a connection to a data source through which you can operate on the data source. The `CDatabase` class treats a data source as a specific instance of data hosted by a

database management system(DBMS). You can connect to multiple databases by creating multiple CDatabase objects.

The simplest way to use the CDatabase class is to construct a CDatabase object call its Open member function. This opens connection to the data source. When you finish using the connection, you call the Close member function and destroy CDatabase object.

Once a CDatabase object is connected to a data source, you can:

- Construct recordsets, which select records from tables or queries.
- Manage transactions, that are committed to the data source at once.
- Execute SQL statements.

CRecordset Class:

The CRecordset class encapsulates a set of records selected from a data source. Record sets make use of RFX mechanism to exchange data between field data members of the recordset object and corresponding table-columns of the data source. Recordsets enable scrolling from record to record, updating records, qualifying the selection with a filter and sorting the selection.

Using a CRecordset object at run time , you can:

- Examine the data fields of the current record.
- Filter or sort the recordset.
- Customize the default SQL SLECT statement.
- Scroll through the selected records.
- Add, update, or delete records .

Once you have finished using the recordset object , it must be closed and them destroyed.

DISPLAYING AND MANIPULATING DATA IN A FORM:

Many data-access applications select data and display it in the fields in a form. The user-interface is a form containing controls in which the user examines, enters or edits data. To make your application for-based use CRecordView class. The database class CRecordView gives you a CFormView object directly connected to a recordset object.

CRecordView Class:

The CRecordView class provides a form view that is directly connected to a record set object. The Dialog Data Exchange(DDX) mechanism exchange data between the recordset and the controls of the record view. Record views also support moving from record to record in the recordset, updating records and closing the associated recordset when the record view closes.

RFX Mechanism:

The Record Field eXchange (RFX) Mechanism exchanges data between the field data members of your recordset object and the corresponding columns of the record in the data source. The RFX mechanism is implemented by the DoFieldExchange member function of the CRecordset class.

The framework calls this function implicitly to automatically exchange data between the field data members of your recordset object and the corresponding columns of the current record on the data source.

The exchange happens in both the directions.

The data types supported by RFX mechanism are as below:

RFX FUNCTION	FUNCTION
RFX_Text	Transfers string data
RFX_Long	Transfer long int data
RFX_Int	Transfer int data
RFX_Byte	Transfers a single byte of data
RFX_Double	Transfers double precision float data
RFX_Date	Transfers time and date data
RFX_Binary	Transfers arrays of bytes of type CByteArray
RFX_LongBinary	Transfers binarylarge object (BLOB) data

Let Us see a code segment to demonstrate transfer of data between the data source field members and the corresponding CRecordset class members.

```
class CRecordSet : public CRecordset
{
public:
    CString Manama;
    Costarring    madders;
    long          mage;
public:
    virtual void DoFieldExchange(CFieldExchange* pFX);

    void CRecordSet::DoFieldExchange(CFieldExchange* pFX)
    {
        AFX_FIELD_MAP(CRecordSet)
        pFX->SetFieldType(CFieldExchange::outputColumn);
        RFX_Text(pFX, _T("[NAME]"), m_name);
        RFX_Text(pFX, _T("[ADDRESS]"), madders);
        RFX_Long(pFX, _T("[AGE]"), mage);
        //}}AFX_FIELD_MAP
    }
}
```

The DoFieldExchange function takes one parameter to the CfieldExchange class. The CFieldExchange class is responsible for the flow of data between the CDatabase class and the CRecordset class.

The first statement in the function SetFieldType, specifies that the data exchange is for the field data members. Then next set of RFX statements makes the data exchange is from the current record of the data source of the corresponding variables in the class CRecordset.

FROM DATABASE TO RECORDVIEW – THE PICTURE:

- ❖ The CDatabase class encapsulates the data source and provides a connection for you to operate on the data source.
 - The RFX mechanism exchanges data between field data members of the recordset object and corresponding table columns of the data source.
- ❖ The CRecordset class encapsulates a set of records from a data source.
 - Recordsets allow you to scroll from record to record, update records, qualify selection with a filter and sort the selection.
 - In the case of the form-view , the DDX mechanism exchanges data between the form-view and the recordset object, and is usually used to display the current record

The framework's Record Field eXchange (RFX) mechanism exchanges data between the recordset and the data source. The dialog data exchange (DDX) mechanism exchanges data between the field data members of the recordset object and the controls on the form. CRecordView also provides default command handler functions for navigating from record to record on the form. The DDX mechanism applies only to the CRecordView class.

CONNECTING TO A DATA SOURCE:

There are two ways to connect to a data source:

- By creating a CDatabase object and then calling its Open member function.
- Using the GetDefaultConnect member function of the CRecordset class.

Example of usage of GetDefaultConnect function as shown under:

```
class CRecordSet : public CRecordset
{
public:

    CString CRecordSet::GetDefaultConnect()
    {
        return _T("ODBC;DSN=xyz");
    }
};
```

In the above example , the GetDefaultConnect is called by the framework. The function returns a string containing information about the data source—the data source name , the user-id and the password.

In the above function, _T macro is used to convert CString object to its UNICODE counterpart if the platform supports UNICODE applications.

GETDEFAULTSQL FUNCTION:

Once the connection between the data source and the application program has been established, the next step is to select a default table. This operation can be done using the GetDefaultSQL member function of the CRecordset class. TH frame work calls this member function implicitly to get the default SQL statement on which the recordset is based. The return value might be a table name .

If the GetDefaultSQL function returns the names of two tables the recordset will retrieve all the records from two tables.

Ex:

```
CString CRecordSet::GetDefaultSQL()
{
    return _T("[Table2]");
}
```

SNAPSHOTS AND DYNASETS:

A dynaset is a recordset with dynamic properties. A recordset object in dynaset mode changes for every updations or deletions or insertions of data . This is helpful in a multiuser environment.

Unlike dynaset, a snapshot type recordset object is static set records cannot be updated.

The default recordset is snapshot.

INITIALIZING A CRECORDSET OBJECT:

The recordset data members are explicitly initialized usually in the constructor of the CRecordset object. The data members of the CRecordset object are :

- m_nDefaultType- Used to specify the type of the recordset as either a snapshot or dynaset.
- m_strFilter—It is used to manipulate the WHERE clause of the SQL statement. The recordset uses this string to filter the records it selects during the Open call.
- m_strFilter data member can also be used to set join conditions.
- m_strSort- Used to set the ORDER BY clause of the SQL statement. You can use this feature to sort a recordset on one or more columns.
 - m_nFields:-specifies the number of fields, this value to the m_nFields data member is assigned by the framework. m_nFields data member assigned by the framework.

DOCUMENT_VIEW ARCHITECTURE AND ODBC:

To implement the document-view architecture for ODBC:

- Create a single document template using the user-defined Document, View and Frame.
- Construct a recordset object in the class derived from the CDocument class. This ensures that the recordset is created when the document is created.
- In the class derived from CRecordView, declare a pointer to the recordset object.
- The constructor of the class derived from CRecordset is designed to accept a parameter-a pointer to CDatabase class.
- Initialize all variables for the recordset object which correspond to field members of the data source , set the values to members like m_n_DefaultType, m_strFilter, m_strSort, m_nFields.
- Use the GetDefaultConnect function to connect to the data source and GetDefaultSQL to set default SQL/Table.
- Use the GetDocument in the OnInitialUpdate member function of your View class to obtain a pointer to the document object.
- Using the pointer obtained, access the recordset data member of the document object and assign it to pointer to the recordset object.
- Associate the recordset pointer of the view, In the View class's OnInitialUpdate member function . If the recordset is already open, close it using the Close member function and open it again.

Let us understand what each class does in the Document-View architecture:

- CRecordset class – This class contains the data members corresponding to the data source's fields. The constructor of the recordset class takes a parameter as a pointer to the CDatabase class and passes it to its base class constructor.
- The GetDefaultConnect function – specifies the data source to connect to and the GetDefaultSQL function returns the SQL statement which generates the record set. The DoFieldExchange transfers the contents of the data sourcefield members with the recordset's fields.
- CDocument class—This class constructs the recordset object.
- CRecordView class—This declares a pointer to the recordset class. The OnInitialUpdate function, which is called implicitly by the frame work, sets and view and opens the recordset. The following are few functions are used by the framework to display the records.
 - IsEOF – Checks if the end of the recordset object is reached.
 - IsBOF - Checks if the beginning of the recordset object is reached.
 - MoveFirst- Moves the record pointer to the first record in the recordset object.
 - MoveLast- Moves the record pointer to the last record in the recordset object.

- **MoveNext**—Moves the record pointer to the next record in the recordset.
- **MovePrev**—Moves the record pointer to the previous record in the recordset.
- **CFrameWnd Class** – Is the same as in all Document-view programs
- **CWinApp Class** – Is same as in all Document-View programs.

CRecordView Class:

A record-view is a from-view with controls that are mapped directly to the field data members of a recordset object. A **CRecordView** object is a view that displays database records in controls. The view is created from a dialog template resource and displays the fields of the **CRecordset** object in the dialog template's controls. The **CRecordView** object uses the **DDX** and **RFX** mechanism to automate the transfer of data between the controls in the form and the fields of the recordset.

The steps involved in manipulating the **CRecordView** class.

The first step towards handling the **CRecordView** class is to code constructor of the class derived from **CRecordView**. The dialog resource-id must be passed to the base class constructor.

Ex:

```
class CMyRecordView : public CRecordView
{
protected:
    CRecordSet* m_pSet;

public:
    CMyRecordView::CMyRecordView() :
        CRecordView(CMyRecordView::IDD)
    {
        m_pSet = NULL;
    }

    CRecordset* CMyRecordView::OnGetRecordset()
    {
```

```
        return m_pSet;  
    }  
  
};
```

BROWSING RECORDS THROUGH A RECORD VIEW:

Let us assume that the menu developed for the application has options for scrolling from one record to another, either in the forward or reverse direction.

The first step is to write handlers for the menu options in the class derived from CFrameWnd.

Ex:

```
IMPLEMENT_DYNCREATE(CMyRecordView, CRecordView)
```

```
BEGIN_MESSAGE_MAP(CMyRecordView, CRecordView)
```

```
    ON_BN_CLICKED(IDC_FIRST, OnFirst)
```

```
    ON_BN_CLICKED(IDC_NEXT, OnNext)
```

```
    ON_BN_CLICKED(IDC_PREVIOUS, OnPrevious)
```

```
    ON_BN_CLICKED(IDC_LAST, OnLast)
```

```
    END_MESSAGE_MAP()
```

The following member functions of the CRecordset class are used to move to the corresponding record.

- MovePrev – Moves to the previous record
- MoveNext – Moves to the next record
- MoveLast – Moves to the Last record
- MoveFirst – Moves to the First record

The sample code to move from one record to the next record

```
void CMyRecordView::OnNext()  
{  
    if(m_pSet->IsEOF())  
    {
```



```
        MessageBox("Attempt to scroll beyond last record","scroll error");  
        return;  
    }  
    m_pSet->MoveNext();  
    UpdateData(FALSE);  
}
```

Similarly the code for move record to first , last , and prev are as below.

```
void CMyRecordView::OnFirst()  
{  
    if(m_pSet->IsBOF())  
    {  
        MessageBox("Attempt to scroll before first record","scroll error");  
        return;  
    }  
    m_pSet->MoveFirst();  
    UpdateData(FALSE);  
  
}
```

```
void CMyRecordView::OnPrevious()  
{  
    if(m_pSet->IsBOF())  
    {  
        MessageBox("Attempt to scroll before first record","scroll error");  
        return;  
    }  
    m_pSet->MovePrev();  
    UpdateData(FALSE);  
  
}
```

```
void CMyRecordView::OnLast()
{
    if(m_pSet->IsEOF())
    {
        MessageBox("Attempt to scroll before last record","scroll error");
        return;
    }
    m_pSet->MoveLast();
    UpdateData(FALSE);
}
```

ADDING RECORDS TO A DATABASE:

Adding records to the recordset can be done using AddNew member function of the CRecordset class.

The changes done are not reflected in the recordset immediately the data is not updated. In order to update the data you need to perform the following tasks:

- Call the Update member function of the CRecordset class. This updates the record currently added by the user.
- Reconstruct the recordset by calling the Requery member function of the CRecordset class.

And clear button should clear the contents of the edit controls in the dialog box.

```
void CMyRecordView::OnClear()
{
    SetDlgItemInt(IDC_AGE,0,TRUE);
    SetDlgItemText(IDC_NAME,"");
    SetDlgItemText(IDC_ADDRESS,"");
    SetDlgItemText(IDC_ID,0);
}
```

```
void CMyRecordView::OnAdd()
{
```

```
m_pSet->MoveLast();
m_pSet->AddNew();
UpdateData(TRUE);
if(!m_pSet->Update())
    MessageBox("cannot perform updation");
if(m_pSet->Requery()==0)
    return;
UpdateData(FALSE);
}
```

MODIFYING RECORDS IN A DATABASE:

Modification is another kind of update operation performed on a data source. The only difference between addition and modification of records using MFC database classes is that no new records are added to the data source. The Edit member function of the CRecordset class is used to bring about modification.

```
void CMyRecordView::OnModify()
{

    m_pSet->Edit();
    UpdateData(TRUE);
    if(!m_pSet->Update())
        MessageBox("cannot perform updation");
    if(m_pSet->Requery()==0)
        return;
    UpdateData(FALSE);
}
```

DELETING RECORDS FROM A DATABASE :

To delete records in the recordset, you use the Delete member function of the CRecordset class. The following code deletes the current record.

```
void CMyRecordView::OnDelete()
{
    m_pSet->Delete();
    m_pSet->MoveNext();
    if(m_pSet->IsEOF())
    {
        m_pSet->MoveLast();
        m_pSet->SetFieldNull(NULL);
    }
    m_pSet->Requery();
    UpdateData(FALSE);
}
```

SetFieldNull(NULL) function is a member function of **CRecordset** class. It will set the recordset to **NULL**.

ADMINISTERING THE ODBC SUPPORT:

Step 1: **Open Control panel window and click on 32bit ODBC icon.**

Step 2: **Click on add button.**

Step 3 : **Select the appropriate driver according to the data source or back end .**

Step 4 : **Enter the data source name and click on select button and select the database.**

Step 5 : **Click on OK.**

CREATING AN MFC APPLICATION WITH DATABASE SUPPORT:

Step 1: **Click on File menu item and then click on New and then click on project tab.**

Step 2: **Enter the project name and select the directory where ever you want to create the project.**

Step 3: Click on Single document interface radio button.

Step 4: Click on the option database without file support.

Step 5: Click on data source button and enter the data source name created through ODBC then click on OK and select the table or tables.

Step 6: Click on Finish button.

The AppWizard creates the following code:

```
class CRecordDoc : public CDocument
{
protected:
    CRecordDoc();
    DECLARE_DYNCREATE(CRecordDoc)
public:
    CRecordSet m_recordSet;

public:
    virtual BOOL OnNewDocument();
    virtual ~CRecordDoc();
    DECLARE_MESSAGE_MAP()
};
```

In the above code CRecordSet is the recordset object derived from CRecordset to the data source and performs all data management.

The AppWizard also generates a dialog box template that will be displayed in the CRecordView object. Following is the listing of the source code generated by AppWizard.

Rest of the code in document is the same as other document-view applications.

```
class CMyRecordView : public CRecordView
{
protected:
    CMyRecordView();
    DECLARE_DYNCREATE(CMyRecordView)
```

public:

CRecordSet* m_pSet;

public:

CRecordDoc* GetDocument();

public:

virtual CRecordset* OnGetRecordset();

virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

protected:

virtual void DoDataExchange(CDataExchange* pDX);

virtual void OnInitialUpdate();

virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);

virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);

virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

//}}AFX_VIRTUAL

public:

virtual ~CMyRecordView();

protected:

//{{AFX_MSG(CMyRecordView)

afx_msg void OnFirst();

afx_msg void OnNext();

afx_msg void OnPrevious();

afx_msg void OnLast();

afx_msg void OnAdd();

afx_msg void OnModify();

afx_msg void OnDelete();

afx_msg void OnClear();

afx_msg void OnExit();

DECLARE_MESSAGE_MAP()

};

void CMyRecordView::OnInitialUpdate()

{

m_pSet = &GetDocument()->m_recordSet;

```
CRecordView::OnInitialUpdate();  
}
```

Rest of the code in view is the same as other document-view applications.

```
class CRecordSet : public CRecordset  
{  
public:  
  
    CRecordSet(CDatabase* pDatabase = NULL);  
    DECLARE_DYNAMIC(CRecordSet)  
  
  
   //{{AFX_FIELD(CRecordSet, CRecordset)  
    CString      m_NAME;  
    CString      m_ADDRESS;  
    long   m_AGE;  
    long   m_ID;  
    //}}AFX_FIELD  
  
   //{{AFX_VIRTUAL(CRecordSet)  
public:  
    virtual CString GetDefaultConnect();  
    virtual CString GetDefaultSQL();  
    virtual void DoFieldExchange(CFieldExchange* pFX);  
    //}}AFX_VIRTUAL  
  
    IMPLEMENT_DYNAMIC(CRecordSet, CRecordset)  
  
    CRecordSet::CRecordSet(CDatabase* pdb)  
    : CRecordset(pdb)  
    {  
        //{{AFX_FIELD_INIT(CRecordSet)  
        m_NAME = _T("");  
        m_ADDRESS = _T("");  
    }  
}
```

```
        m_AGE = 0;
        m_ID = 0;
        m_nFields = 4;
        //}}AFX_FIELD_INIT
        m_nDefaultType = snapshot;
    }
```

```
CString CRecordSet::GetDefaultConnect()
{
    return _T("ODBC;DSN=xyz");
}
```

```
CString CRecordSet::GetDefaultSQL()
{
    return _T("[Table2]");
}
```

```
void CRecordSet::DoFieldExchange(CFieldExchange* pFX)
{
    //{{AFX_FIELD_MAP(CRecordSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, _T("[NAME]"), m_NAME);
    RFX_Text(pFX, _T("[ADDRESS]"), m_ADDRESS);
    RFX_Long(pFX, _T("[AGE]"), m_AGE);
    RFX_Long(pFX, _T("[ID]"), m_ID);
    //}}AFX_FIELD_MAP
}
```

The application compiles to produce the output where dialog box is empty since no controls are there.

The next step is to add controls to the dialog box and associate the controls with the variables of the record set object with the help of Class Wizard .

The ClassWizard generates a code in the record view, which is below.


```
void CMyRecordView::DoDataExchange(CDataExchange* pDX)
{
    CRecordView::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CMyRecordView)
    DDX_FieldText(pDX, IDC_ID, m_pSet->m_ID, m_pSet);
        DDX_FieldText(pDX, IDC_ADDRESS, m_pSet->m_ADDRESS, m_pSet);
    DDX_FieldText(pDX, IDC_NAME, m_pSet->m_NAME, m_pSet);
    DDX_FieldText(pDX, IDC_AGE, m_pSet->m_AGE, m_pSet);
   //}}AFX_DATA_MAP
}
```

The following code implements the RFX mechanism for the variables in the recordset object

```
void CRecordSet::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CRecordSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, _T("[NAME]"), m_NAME);
    RFX_Text(pFX, _T("[ADDRESS]"), m_ADDRESS);
    RFX_Long(pFX, _T("[AGE]"), m_AGE);
    RFX_Long(pFX, _T("[ID]"), m_ID);
   //}}AFX_FIELD_MAP
}
```

REGISTRY:

Introduction:

Applications running under DOS environment depend on the information in CONFIG.SYS and AUTOEXEC.BAT files for their configuration and initialization. Similarly, under Windows environment the applications depend on WIN.INI and SYSTEM.INI files. These .INI files store information that control application startup as well as various resources used by applications like cursor and icon.

These files provide an ideal solution in single user environment. But these files are incapable of storing separate information for individual users in a multi-user environment . Both WIN95 and WinNT centralize program information in a database called registry. They still support the above files for compatibility.

REGISTRY:

The registry is a hierarchical database . Users can edit registry values by running regedit utility. The registry serves as a central configuration database stored in it can be viewed as a tree structure. Each node in the tree is called a key. The registry stores all the information about a computer and its users by dividing them into five sub-trees. These trees are described as follows:

HKEY_LOCAL_MACHINE:

This subtree contains information about hardware currently installed, programs and applications running on the machine.

HKEY_CURRENT_CONFIG:

This sub-tree window contains information about the hardware properties and also the hardware configuration used by the operating system during booting.

HKEY_CLASSES_ROOT:

This sub-tree contains the information about file associations. Most of the information under this tree is also found in the HKEY_LOCAL_MACHINE sub-hive.

HKEY_USERS:

This sub-tree contains user profiles. This information includes user properties as well as their privileges used in a multi user environment. The users who are using the system for the first time are also given a default profile.

HKEY_CURRENT_USER:

Contains the profile information, which includes environment profiles, personal program groups, desktop settings, network connections and printers.

ACCESSING REGISTRY:

All entries in the registry are of the following format.

SECTION:ENTRY=Value

Applications require to get information from the registry. Also in some cases applications should be able to write information to the registry.

READING FROM REGISTRY:

The GetProfileString function is be used for accessing the registry. This function is a member of the CWinApp class. The usage of the function is as follows:

CString GetProfileString(CString Section, CString Entry);

This function returns a string which has as entry as specified in the Entry from the section specified by Section from the registry.

WRITING FROM REGISTRY:

The other function frequently used for storing information in registry is WriteProfileString. This is also a member function of the CWinApp Class. The usage of the function is as follows.

BOOL WriteProfileString(CString Section, CString Entry, CString Value);

This function writes the Value for the entry under the section in the registry. This function returns TRUE if successful, FALSE otherwise.

Winsock

Winsock is the lowest level Windows API for TCP/IP programming. Part of the code is located in wsock32.dll (the exported functions that your program calls), and part is inside the Windows kernel. You can write both internet server programs and internet client programs using the Winsock API. This API is based on the original Berkely Sockets API for UNIX.

Synchronous vs. Asynchronous Winsock Programming

Winsock was introduced first for Win16, which did not support multithreading. Consequently, most developers used Winsock in the asynchronous mode. In that mode, all sorts of hidden windows and *PeekMessage* calls enabled single-threaded programs to make Winsock send and receive calls without blocking, thus keeping the user interface (UI) alive. Asynchronous Winsock programs were complex, often implementing "state machines" that processed callback functions, trying to figure out what to do next based on what had just happened.

The MFC Winsock Classes

1. *CAsyncSocket*

2. *CSocket* classes

The Blocking Socket Classes

Since we couldn't use MFC, we had to write our own Winsock classes. *CBlockingSocket* is a thin wrapping of the Winsock API, designed only for synchronous use in a worker thread. The only fancy features are exception-throwing on errors and time-outs for sending and receiving data. The exceptions help you write cleaner code because you don't need to have error tests after every Winsock call. The time-outs (implemented with the Winsock *select* function) prevent a communication fault from blocking a thread indefinitely.

CHttpBlockingSocket is derived from *CBlockingSocket* and provides functions for reading HTTP data. *CSockAddr* and *CBlockingSocketException* are helper classes.

The CSockAddr Helper Class

Many Winsock functions take socket address parameters. As you might remember, a socket address consists of a 32-bit IP address plus a 16-bit port number. The actual Winsock type is a 16-byte *sockaddr_in* structure, which looks like this:

```
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

The IP address is stored as type *in_addr*, which looks like this:

```
struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
```

} The **CBlockingSocketException** Class

All the *CBlockingSocket* functions throw a *CBlockingSocketException* object when Winsock returns an error. This class is derived from the MFC *CException* class and thus overrides the *GetErrorMessage* function. This function gives the Winsock error number and a character string that *CBlockingSocket* provided when it threw the exception.

The **CBlockingSocket** Class

It shows an excerpt from the header file for the *CBlockingSocket* class.

BLOCKSOCK.H

```
class CBlockingSocket : public CObject
{
    DECLARE_DYNAMIC(CBlockingSocket)
public:
    SOCKET m_hSocket;
    CBlockingSocket(); { m_hSocket = NULL; }
    void Cleanup();
    void Create(int nType = SOCK_STREAM);
    void Close();
    void Bind(LPCSOCKADDR psa);
    void Listen();
    void Connect(LPCSOCKADDR psa);
    BOOL Accept(CBlockingSocket& s, LPCSOCKADDR psa);
    int Send(const char* pch, const int nSize, const int
nSecs);
    int Write(const char* pch, const int nSize, const int
nSecs);
    int Receive(char* pch, const int nSize, const int nSecs);
    int SendDatagram(const char* pch, const int nSize,
LPCSOCKADDR psa,
        const int nSecs);
    int ReceiveDatagram(char* pch, const int nSize,
LPCSOCKADDR psa,
        const int nSecs);
    void GetPeerAddr(LPCSOCKADDR psa);
    void GetSockAddr(LPCSOCKADDR psa);
    static CSockAddr GetHostByName(const char* pchName,
        const USHORT ushPort = 0);
    static const char* GetHostByAddr(LPCSOCKADDR psa);
    operator SOCKET();
    { return m_hSocket; }
};
```

Following is a list of the *CBlockingSocket* member functions, starting with the constructor:

- **Constructor**—The *CBlockingSocket* constructor makes an uninitialized object. You must call the *Create* member function to create a Windows socket and connect it to the C++ object.
- **Create**—This function calls the Winsock *socket* function and then sets the *m_hSocket* data member to the returned 32-bit *SOCKET* handle.

Parameter Description

nType Type of socket; should be *SOCK_STREAM* (the default value) or *SOCK_DGRAM*

- **Close**—This function closes an open socket by calling the Winsock *closesocket* function. The *Create* function must have been called previously. The destructor does not call this function because it would be impossible to catch an exception for a global object. Your server program can call *Close* anytime for a socket that is listening.
- **Bind**—This function calls the Winsock *bind* function to bind a previously created socket to a specified socket address. Prior to calling *Listen*, your server program calls *Bind* with a socket address containing the listening port number and server's IP address. If you supply *INADDR_ANY* as the IP address, Winsock deciphers your computer's IP address.

Parameter Description

psa A *CSockAddr* object or a pointer to a variable of type *sockaddr*

- **Listen**—This TCP function calls the Winsock *listen* function. Your server program calls *Listen* to begin listening on the port specified by the previous *Bind* call. The function returns immediately.
- **Accept**—This TCP function calls the Winsock *accept* function. Your server program calls *Accept* immediately after calling *Listen*. *Accept* returns when a client connects to the socket, sending back a new socket (in a *CBlockingSocket* object that you provide) that corresponds to the new connection.

Parameter Description

s A reference to an existing *CBlockingSocket* object for which *Create* has not been called

psa A *CSockAddr* object or a pointer to a variable of type *sockaddr* for the connecting socket's address

Return value *TRUE* if successful

- **Connect**—This TCP function calls the Winsock *connect* function. Your client program calls *Connect* after calling *Create*. *Connect* returns when the connection has been made.

Parameter	Description
-----------	-------------

<i>psa</i>	A <i>CSockAddr</i> object or a pointer to a variable of type <i>sockaddr</i>
------------	--

- **Send**—This TCP function calls the Winsock *send* function after calling *select* to activate the time-out. The number of bytes actually transmitted by each *Send* call depends on how quickly the program at the other end of the connection can receive the bytes. *Send* throws an exception if the program at the other end closes the socket before it reads all the bytes.

Parameter	Description
-----------	-------------

<i>pch</i>	A pointer to a buffer that contains the bytes to send
------------	---

<i>nSize</i>	The size (in bytes) of the block to send
--------------	--

<i>nSecs</i>	Time-out value in seconds
--------------	---------------------------

Return value	The actual number of bytes sent
--------------	---------------------------------

- **Write**—This TCP function calls *Send* repeatedly until all the bytes are sent or until the receiver closes the socket.

Parameter	Description
-----------	-------------

<i>pch</i>	A pointer to a buffer that contains the bytes to send
------------	---

<i>nSize</i>	The size (in bytes) of the block to send
--------------	--

<i>nSecs</i>	Time-out value in seconds
--------------	---------------------------

Return value	The actual number of bytes sent
--------------	---------------------------------

- **Receive**—This TCP function calls the Winsock *recv* function after calling *select* to activate the time-out. This function returns only the bytes that have been received. For more information, see the description of the *CHttpBlockingSocket* class in the next section.

Parameter	Description
-----------	-------------

<i>pch</i>	A pointer to an existing buffer that will receive the incoming bytes
------------	--

<i>nSize</i>	The maximum number of bytes to receive
--------------	--

<i>nSecs</i>	Time-out value in seconds
--------------	---------------------------

Return value	The actual number of bytes received
--------------	-------------------------------------

- **SendDatagram**—This UDP function calls the Winsock *sendto* function. The program on the other end needs to call *ReceiveDatagram*. There is no need to call *Listen*, *Accept*, or

Connect for datagrams. You must have previously called *Create* with the parameter set to *SOCK_DGRAM*.

Parameter	Description
<i>pch</i>	A pointer to a buffer that contains the bytes to send
<i>nSize</i>	The size (in bytes) of the block to send
<i>psa</i>	The datagram's destination address; a <i>CSockAddr</i> object or a pointer to a variable of type <i>sockaddr</i>
<i>nSecs</i>	Time-out value in seconds
Return value	The actual number of bytes sent

- ***ReceiveDatagram***—This UDP function calls the Winsock *recvfrom* function. The function returns when the program at the other end of the connection calls *SendDatagram*. You must have previously called *Create* with the parameter set to *SOCK_DGRAM*.

Parameter	Description
<i>pch</i>	A pointer to an existing buffer that will receive the incoming bytes
<i>nSize</i>	The size (in bytes) of the block to send
<i>psa</i>	The datagram's destination address; a <i>CSockAddr</i> object or a pointer to a variable of type <i>sockaddr</i>
<i>nSecs</i>	Time-out value in seconds
Return value	The actual number of bytes received

- ***GetPeerAddr***—This function calls the Winsock *getpeername* function. It returns the port and IP address of the socket on the other end of the connection. If you are connected to the Internet through a Web proxy server, the IP address is the proxy server's IP address.

Parameter	Description
<i>psa</i>	A <i>CSockAddr</i> object or a pointer to a variable of type <i>sockaddr</i>

- ***GetSockAddr***—This function calls the Winsock *getsockname* function. It returns the socket address that Winsock assigns to this end
- of the connection. If the other program is a server on a LAN, the IP address is the address assigned to this computer's network board. If the other program is a server on the Internet, your service provider assigns the IP address when you dial in. In both cases, Winsock assigns the port number, which is different for each connection.

Parameter	Description
-----------	-------------

<i>psa</i>	A <i>CSockAddr</i> object or a pointer to a variable of type <i>sockaddr</i>
------------	--

- **GetHostByName**—This static function calls the Winsock function *gethostbyname*. It queries a name server and then returns the socket address corresponding to the host name. The function times out by itself.

Parameter	Description
-----------	-------------

<i>pchName</i>	A pointer to a character array containing the host name to resolve
----------------	--

<i>ushPort</i>	The port number (default value 0) that will become part of the returned socket address
----------------	--

Return value	The socket address containing the IP address from the DNS plus the port number <i>ushPort</i>
--------------	---

- **GetHostByAddr**—This static function calls the Winsock *gethostbyaddr* function. It queries a name server and then returns the host name corresponding to the socket address. The function times out by itself.

Parameter	Description
-----------	-------------

<i>psa</i>	A <i>CSockAddr</i> object or a pointer to a variable of type <i>sockaddr</i>
------------	--

Return value	A pointer to a character array containing the host name; the caller should not delete this memory
--------------	---

- **Cleanup**—This function closes the socket if it is open. It doesn't throw an exception, so you can call it inside an exception *catch* block.
- **operator SOCKET**—This overloaded operator lets you use a *CBlockingSocket* object in place of a *SOCKET* parameter.

The *CHttpBlockingSocket* Class

If you call *CBlockingSocket::Receive*, you'll have a difficult time knowing when to stop receiving bytes. Each call returns the bytes that are stacked up at your end of the connection at that instant. If there are no bytes, the call blocks, but if the sender closed the socket, the call returns zero bytes.

In the HTTP section, you learned that the client sends a request terminated by a blank line. The server is supposed to send the response headers and data as soon as it detects the blank line, but the client needs to analyze the response headers before it reads the data. This means that as long as a TCP connection remains open, the receiving program must process the received data as it comes in. A simple but inefficient technique would be to call *Receive* for 1 byte at a time. A better way is to use a buffer.

The *CHttpBlockingSocket* class adds buffering to *CBlockingSocket*, and it provides two new member functions. Here is part of the `\vcpp32\ex34A\Blocksock.h` file:

```
class CHttpBlockingSocket : public CBlockingSocket
{
public:
    DECLARE_DYNAMIC(CHttpBlockingSocket)
    enum {nSizeRecv = 1000}; // max receive buffer size (> hdr
line                                     // length)

    CHttpBlockingSocket();
    ~CHttpBlockingSocket();
    int ReadHttpHeaderLine(char* pch, const int nSize, const int
nSecs);
    int ReadHttpResponse(char* pch, const int nSize, const int
nSecs);
private:
    char* m_pReadBuf; // read buffer
    int m_nReadBuf; // number of bytes in the read buffer
};
```

The constructor and destructor take care of allocating and freeing a 1000-character buffer. The two new member functions are as follows:

- ***ReadHttpHeaderLine***—This function returns a single header line, terminated with a `<cr><lf>` pair. *ReadHttpHeaderLine* inserts a terminating zero at the end of the line. If the line buffer is full, the terminating zero is stored in the last position.

Parameter	Description
<i>pch</i>	A pointer to an existing buffer that will receive the incoming line (zero-terminated)
<i>nSize</i>	The size of the <i>pch</i> buffer
<i>nSecs</i>	Time-out value in seconds
Return value	The actual number of bytes received, excluding the terminating zero

- ***ReadHttpResponse***—This function returns the remainder of the server's response received when the socket is closed or when the buffer is full. Don't assume that the buffer contains a terminating zero.

Parameter	Description
<i>pch</i>	A pointer to an existing buffer that will receive the incoming data
<i>nSize</i>	The maximum number of bytes to receive
<i>nSecs</i>	Time-out value in seconds

Return value The actual number of bytes received

A Simplified HTTP Server Program

Now it's time to use the blocking socket classes to write an HTTP server program. All the frills have been eliminated, but the code actually works with a browser. This server doesn't do much except return some hard-coded headers and HTML statements in response to any GET request. (See the EX34A program later in this chapter for a more complete HTTP server.)

Initializing Winsock

Before making any Winsock calls, the program must initialize the Winsock library. The following statements in the application's *InitInstance* member function do the job:

```
WSADATA wsd;  
WSAStartup(0x0101, &wsd);
```

Starting the Server

The server starts in response to some user action, such as a menu choice. Here's the command handler:

```
CBlockingSocket g_sListen; // one-and-only global socket for  
listening  
void CSocketView::OnInternetStartServer()  
{  
    try {  
        CSockAddr saServer(INADDR_ANY, 80);  
        g_sListen.Create();  
        g_sListen.Bind(saServer);  
        g_sListen.Listen();  
        AfxBeginThread(ServerThreadProc, GetSafeHwnd());  
    }  
    catch(CBlockingSocketException* e) {  
        g_sListen.Cleanup();  
        // Do something about the exception  
        e->Delete();  
    }  
}
```

Pretty simple, really. The handler creates a socket, starts listening on it, and then starts a worker thread that waits for some client to connect to port 80. If something goes wrong, an exception is thrown. The global *g_sListen* object lasts for the life of the program and is capable of accepting multiple simultaneous connections, each managed by a separate thread.

The Server Thread

Now let's look at the *ServerThreadProc* function:

```
UINT ServerThreadProc(LPVOID pParam)
{
    CSockAddr saClient;
    CHttpBlockingSocket sConnect;
    char request[100];
    char headers[] = "HTTP/1.0 200 OK\r\n"
        "Server: Inside Visual C++ SOCK01\r\n"
        "Date: Thu, 05 Sep 1996 17:33:12 GMT\r\n"
        "Content-Type: text/html\r\n"
        "Accept-Ranges: bytes\r\n"
        "Content-Length: 187\r\n"
        "\r\n"; // the important blank line
    char html[] =
        "<html><head><title>Inside Visual C++
Server</title></head>\r\n"
        "<body><body
background=\"../samples/images/usa1.jpg\">\r\n"
        "<h1><center>This is a custom home
page</center></h1><p>\r\n"
        "</body></html>\r\n\r\n";
    try {
        if(!g_sListen.Accept(sConnect, saClient)) {
            // Handler in view class closed the listening socket
            return 0;
        }
        AfxBeginThread(ServerThreadProc, pParam);
        // read request from client
        sConnect.ReadHttpHeaderLine(request, 100, 10);
        TRACE("SERVER: %s", request); // Print the first header
        if(strnicmp(request, "GET", 3) == 0) {
            do { // Process the remaining request headers
                sConnect.ReadHttpHeaderLine(request, 100, 10);
                TRACE("SERVER: %s", request); // Print the other
headers
                } while(strcmp(request, "\r\n"));
            sConnect.Write(headers, strlen(headers), 10); //
response hdrs
            sConnect.Write(html, strlen(html), 10); // HTML code
        }
        else {
            TRACE("SERVER: not a GET\n");
            // don't know what to do
        }
        sConnect.Close(); // Destructor doesn't close it
    }
    catch(CBlockingSocketException* e) {
        // Do something about the exception
        e->Delete();
    }
    return 0;
}
```

The most important function call is the *Accept* call. The thread blocks until a client connects to the server's port 80, and then *Accept* returns with a new socket, *sConnect*. The current thread immediately starts another thread.

In the meantime, the current thread must process the client's request that just came in on *sConnect*. It first reads all the request headers by calling *ReadHttpRequestLine* until it detects a blank line. Then it calls *Write* to send the response headers and the HTML statements. Finally, the current thread calls *Close* to close the connection socket. End of story for this connection. The next thread is sitting, blocked at the *Accept* call, waiting for the next connection.

Cleaning Up

To avoid a memory leak on exit, the program must ensure that all worker threads have been terminated. The simplest way to do this is to close the listening socket. This forces any thread's pending *Accept* to return *FALSE*, causing the thread to exit.

```
try {
    g_sListen.Close();
    Sleep(340); // Wait for thread to exit
    WSACleanup(); // Terminate Winsock
}
catch(CUserException* e) {
    e->Delete();
}
```

A problem might arise if a thread were in the process of fulfilling a client request. In that case, the main thread should positively ensure that all threads have terminated before exiting.

A Simplified HTTP Client Program

Now for the client side of the story—a simple working program that does a blind GET request. When a server receives a GET request with a slash, as shown below, it's supposed to deliver its default HTML file:

```
GET / HTTP/1.0
```

If you typed *http://www.slowsoft.com* in a browser, the browser sends the blind GET request.

This client program can use the same *CHttpBlockingSocket* class you've already seen, and it must initialize Winsock the same way the server did. A command handler simply starts a client thread with a call like this:

```
AfxBeginThread(ClientSocketThreadProc, GetSafeHwnd());
```

Here's the client thread code:

```
CString g_strServerName = "localhost"; // or some other host name
UINT ClientSocketThreadProc(LPVOID pParam)
{
    CHttpBlockingSocket sClient;
```

```
char* buffer = new char[MAXBUF];
int nBytesReceived = 0;
char request[] = "GET / HTTP/1.0\r\n";
char headers[] = // Request headers
    "User-Agent: Mozilla/1.22 (Windows; U; 32bit)\r\n"
    "Accept: */*\r\n"
    "Accept: image/gif\r\n"
    "Accept: image/x-xbitmap\r\n"
    "Accept: image/jpeg\r\n"
    "\r\n"; // need this
CSockAddr saServer, saClient;
try {
    sClient.Create();
    saServer =
CBlockingSocket::GetHostByName(g_strServerName, 80);
    sClient.Connect(saServer);
    sClient.Write(request, strlen(request), 10);
    sClient.Write(headers, strlen(headers), 10);
    do { // Read all the server's response headers
        nBytesReceived = sClient.ReadHttpHeaderLine(buffer,
100, 10);
    } while(strcmp(buffer, "\r\n")); // through the first
blank line
    nBytesReceived = sClient.ReadHttpResponse(buffer, 100,
10);
    if(nBytesReceived == 0) {
        AfxMessageBox("No response received");
    }
    else {
        buffer[nBytesReceived] = '\0';
        AfxMessageBox(buffer);
    }
}
catch(CBlockingSocketException* e) {
    // Log the exception
    e->Delete();
}
sClient.Close();
delete [] buffer;
return 0; // The thread exits
}
```

This thread first calls *CBlockingSocket::GetHostByName* to get the server computer's IP address. Then it creates a socket and calls *Connect* on that socket. Now there's a two-way communication channel to the server. The thread sends its GET request followed by some request headers, reads the server's response headers, and then reads the response file itself, which it assumes is a text file. After the thread displays the text in a message box, it exits.

3.WinInet

WinInet is a higher-level API than Winsock, but it works only for HTTP, FTP, and gopher client programs in both asynchronous and synchronous modes. You can't use it to build servers. The WININET DLL is independent of the WINSOCK32 DLL. Microsoft Internet Explorer 3.0 (IE3) uses WinInet, and so do ActiveX controls.

WinInet's Advantages over Winsock

WinInet far surpasses Winsock in the support it gives to a professional-level client program. Following are just some of the WinInet benefits:

- **Caching**—Just like IE3, your WinInet client program caches HTML files and other Internet files. You don't have to do a thing. The second time your client requests a particular file, it's loaded from a local disk instead of from the Internet.
- **Security**—WinInet supports basic authentication, Windows NT challenge/response authentication, and the Secure Sockets Layer (SSL). Authentication is described in [Chapter 35](#).
- **Web proxy access**—You enter proxy server information through the Control Panel (click on the Internet icon), and it's stored in the Registry. WinInet reads the Registry and uses the proxy server when required.
- **Buffered I/O**—WinInet's read function doesn't return until it can deliver the number of bytes you asked for. (It returns immediately, of course, if the server closes the socket.) Also, you can read individual text lines if you need to.
- **Easy API**—Status callback functions are available for UI update and cancellation. One function, *CInternetSession::OpenURL*, finds the server's IP address, opens a connection, and makes the file ready for reading, all in one call. Some functions even copy Internet files directly to and from disk.
- **User friendly**—WinInet parses and formats headers for you. If a server has moved a file to a new location, it sends back the new URL in an HTTP Location header. WinInet seamlessly accesses the new server for you. In addition, WinInet puts a file's modified date in the request header for you.

The MFC WinInet is a modern API available only for Win32. The MFC wrapping is quite good, which means we didn't have to write our own WinInet class library. Yes, MFC WinInet supports blocking calls in multithreaded programs, and by now you know that makes us happy.

The MFC classes closely mirror the underlying WinInet architecture, and they add exception processing. These classes are summarized in the sections on the following pages.

CInternetSession

You need only one *CInternetSession* object for each thread that accesses the Internet. After you have your *CInternetSession* object, you can establish HTTP, FTP, or gopher connections or you can open remote files directly by calling the *OpenURL* member function. You can use the *CInternetSession* class directly, or you can derive a class from it in order to support status callback functions.

The *CInternetSession* constructor calls the WinInet *InternetOpen* function, which returns an *HINTERNET session handle* that is stored inside the *CInternetSession* object. This function initializes your application's use of the Win- Inet library, and the session handle is used internally as a parameter for other WinInet calls.

CHttpConnection

An object of class *CHttpConnection* represents a "permanent" HTTP connection to a particular host. You know already that HTTP doesn't support permanent connections and that FTP doesn't either. (The connections last only for the duration of a file transfer.) WinInet gives the appearance of a permanent connection because it remembers the host name.

After you have your *CInternetSession* object, you call the *GetHttpConnection* member function, which returns a pointer to a *CHttpConnection* object. (Don't forget to delete this object when you are finished with it.)

The *GetHttpConnection* member function calls the WinInet *InternetConnect* function, which returns an *HINTERNET* connection handle that is stored inside the *CHttpConnection* object and used for subsequent WinInet calls.

CFTPConnection*, *CGopherConnection

These classes are similar to *CHttpConnection*, but they use the FTP and gopher protocols. The *CFTPConnection* member functions *GetFile* and *PutFile* allow you to transfer files directly to and from your disk.

CInternetFile

With HTTP, FTP, or gopher, your client program reads and writes byte streams. The MFC WinInet classes make these byte streams look like ordinary files. If you look at the class hierarchy, you'll see that *CInternetFile* is derived from *CStdioFile*, which is derived from *CFile*. Therefore, *CInternetFile* and its derived classes override familiar *CFile* functions such as *Read* and *Write*. For FTP files, you use *CInternetFile* objects directly, but for HTTP and gopher files, you use objects of the derived classes *CHttpFile* and *CGopherFile*. You don't construct a *CInternetFile* object directly, but you call *CFTPConnection::OpenFile* to get a *CInternetFile* pointer.

If you have an ordinary *CFile* object, it has a 32-bit *HANDLE* data member that represents the underlying disk file. A *CInternetFile* object uses the same *m_hFile* data member, but that data member holds a 32-bit Internet file handle of type *HINTERNET*, which is not interchangeable with a *HANDLE*. The *CInternetFile* overridden member functions use this handle to call WinInet functions such as *InternetReadFile* and *InternetWriteFile*.

CHttpFile

This Internet file class has member functions that are unique to HTTP files, such as *AddRequestHeaders*, *SendRequest*, and *GetFileURL*. You don't construct a *CHttpFile* object directly, but you call the *CHttpConnection::OpenRequest* function, which calls the WinInet function *HttpOpenRequest* and returns a *CHttpFile* pointer. You can specify a GET or POST request for this call.

Once you have your *CHttpFile* pointer, you call the *CHttpFile::SendRequest* member function, which actually sends the request to the server. Then you call *Read*.

CFtpFileFind, CGopherFileFind

These classes let your client program explore FTP and gopher directories.

CInternetException

The MFC WinInet classes throw *CInternetException* objects that your program can process with try/catch logic.

Internet Session Status Callbacks

WinInet and MFC provide callback notifications as a WinInet operation progresses, and these status callbacks are available in both synchronous (blocking) and asynchronous modes. In synchronous mode (which we're using exclusively here), your WinInet calls block even though you have status callbacks enabled.

Callbacks are easy in C++. You simply derive a class and override selected virtual functions. The base class for WinInet is *CInternetSession*. Now let's derive a class named *CCallbackInternetSession*:

```
class CCallbackInternetSession : public CInternetSession
{
public:
    CCallbackInternetSession( LPCTSTR pstrAgent = NULL, DWORD
dwContext = 1,
        DWORD dwAccessType = PRE_CONFIG_INTERNET_ACCESS,
        LPCTSTR pstrProxyName = NULL, LPCTSTR pstrProxyBypass =
NULL,
        DWORD dwFlags = 0 ) { EnableStatusCallback() }
protected:
    virtual void OnStatusCallback(DWORD dwContext, DWORD
dwInternalStatus,
        LPVOID lpvStatusInformation, DWORD
dwStatusInformationLength);
};
```

The only coding that's necessary is a constructor and a single overridden function, *OnStatusCallback*. The constructor calls *CInternetSession::EnableStatusCallback* to enable the status callback feature. Your WinInet client program makes its various Internet blocking calls, and when the status changes, *OnStatusCallback* is called. Your overridden function quickly updates the UI and returns, and then the Internet operation continues. For HTTP, most of the callbacks originate in the *CHttpFile::SendRequest* function.

What kind of events trigger callbacks? A list of the codes passed in the *dwInternalStatus* parameter is shown here.

Code Passed

INTERNET_STATUS_RESOLVING_NAME

Action Taken

Looking up the IP address of the

	supplied name. The name is now in <i>lpvStatusInformation</i> .
<i>INTERNET_STATUS_NAME_RESOLVED</i>	Successfully found the IP address. The IP address is now in <i>lpvStatusInformation</i> .
<i>INTERNET_STATUS_CONNECTING_TO_SERVER</i>	Connecting to the socket.
<i>INTERNET_STATUS_CONNECTED_TO_SERVER</i>	Successfully connected to the socket.
<i>INTERNET_STATUS_SENDING_REQUEST</i>	Send the information request to the server.
<i>INTERNET_STATUS_REQUEST_SENT</i>	Successfully sent the information request to the server.
<i>INTERNET_STATUS_RECEIVING_RESPONSE</i>	Waiting for the server to respond to a request.
<i>INTERNET_STATUS_RESPONSE_RECEIVED</i>	Successfully received a response from the server.
<i>INTERNET_STATUS_CLOSING_CONNECTION</i>	Closing the connection to the server.
<i>INTERNET_STATUS_CONNECTION_CLOSED</i>	Successfully closed the connection to the server.
<i>INTERNET_STATUS_HANDLE_CREATED</i>	Program can now close the handle.
<i>INTERNET_STATUS_HANDLE_CLOSING</i>	Successfully terminated this handle value.
<i>INTERNET_STATUS_REQUEST_COMPLETE</i>	Successfully completed the asynchronous operation.

You can use your status callback function to interrupt a WinInet operation. You could, for example, test for an event set by the main thread when the user cancels the operation.

A Simplified WinInet Client Program

And now for the WinInet equivalent of our Winsock client program that implements a blind GET request. Because you're using WinInet in blocking mode, you must put the code in a worker thread. That thread is started from a command handler in the main thread:

```
AfxBeginThread(ClientWinInetThreadProc, GetSafeHwnd());
```

Here's the client thread code:

```
CString g_strServerName = "localhost"; // or some other host name
UINT ClientWinInetThreadProc(LPVOID pParam)
{
    CInternetSession session;
```

```
CHttpConnection* pConnection = NULL;
CHttpFile* pFile1 = NULL;
char* buffer = new char[MAXBUF];
UINT nBytesRead = 0;
try {
    pConnection = session.GetHttpConnection(g_strServerName,
80);

    pFile1 = pConnection->OpenRequest(1, "/"); // blind GET
    pFile1->SendRequest();
    nBytesRead = pFile1->Read(buffer, MAXBUF - 1);
    buffer[nBytesRead] = '\\0'; // necessary for message box
    char temp[10];
    if(pFile1->Read(temp, 10) != 0) {
        // makes caching work if read complete
        AfxMessageBox("File overran buffer - not cached");
    }
    AfxMessageBox(buffer);
}
catch(CInternetException* e) {
    // Log the exception
    e->Delete();
}
if(pFile1) delete pFile1;
if(pConnection) delete pConnection;
delete [] buffer;
return 0;
}
```

The second *Read* call needs some explanation. It has two purposes. If the first *Read* doesn't read the whole file, that means that it was longer than *MAXBUF - 1*. The second *Read* will get some bytes, and that lets you detect the overflow problem. If the first *Read* reads the whole file, you still need the second *Read* to force WinInet to cache the file on your hard disk. Remember that WinInet tries to read all the bytes you ask it to—through the end of the file. Even so, you need to read 0 bytes after that.

Programming the Microsoft Internet Information Server

Microsoft Internet Information Server (IIS) 4.0, which is bundled with Microsoft Windows NT Server 4.0. IIS is actually three separate servers—one for HTTP (for the World Wide Web), one for FTP, and one for gopher. This chapter tells you how to write HTTP server extensions using the Microsoft IIS application programming interface (ISAPI) that is part of Microsoft ActiveX technology. an ISAPI server extension and an ISAPI filter, both of which are DLLs. An ISAPI server extension can perform Internet business transactions such as order entry. An ISAPI filter intercepts data traveling to and from the server and thus can perform specialized logging and other tasks.

Microsoft IIS

Microsoft IIS is a high-performance Internet/intranet server that takes advantage of Windows NT features such as I/O completion ports, the Win32 function *TransmitFile*, file-handle caching, and CPU scaling for threads.

Installing and Controlling IIS

When you install Windows NT Server 4.0, you are given the option of installing IIS. If you selected IIS at setup, the server will be running whenever Windows NT is running. IIS is a special kind of Win32 program called a service (actually three services—WWW, HTTP, and gopher—in one program called inetinfo.exe), which won't appear in the taskbar. You can control IIS from the Services icon in the Control Panel, but you'll probably want to use the Internet Service Manager program instead.

Running Internet Service Manager

You can run Internet Service Manager from the Microsoft Internet Server menu that's accessible on the Start menu.

You can also run an HTML-based version of Internet Service Manager remotely from a browser. That version allows you to change service parameters, but it won't let you turn services on and off. Figure 35-1 shows the Microsoft Internet Service Manager screen with the World Wide Web (WWW) running and FTP services stopped.

You can select a service by clicking on its icon at the left. The triangle and square buttons on the toolbar of the screen allow you to turn the selected service on or off.

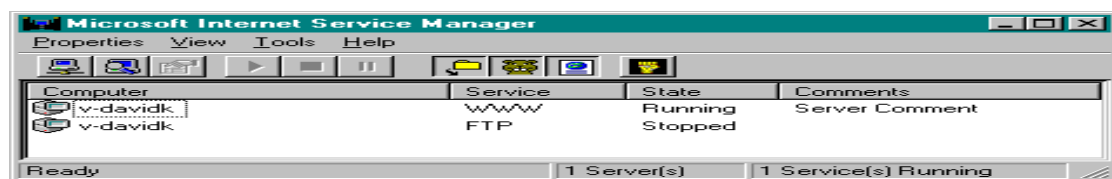


Figure 35-1. *The Microsoft Internet Service Manager screen.*

IIS Security

After you double-click on the WWW service icon of the Microsoft Internet Service Manager screen, you'll see a property sheet. The Service page lets you configure IIS security. When a client browser requests a file, the server impersonates a local user for the duration of the request

and that user name determines which files the client can access. Which local user does the server impersonate? Most often, it's the one you see in the Username field, shown in Figure 35-2.

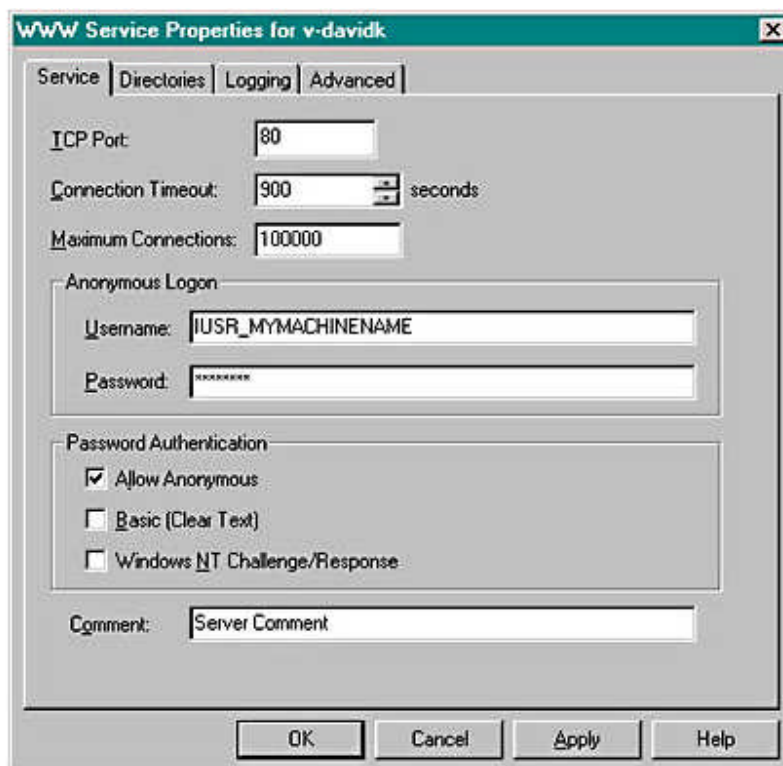


Figure 35-2. *The WWW Service Properties screen.*

Most Web page visitors don't supply a user name and password, so they are considered anonymous users. Those users have the same rights they would have if they had logged on to your server locally as IUSR_MYMACHINENAME. That means that IUSR_MYMACHINENAME must appear in the list of users that is displayed when you run User Manager or User Manager For Domains (from the Administrative Tools menu), and the passwords must match. The IIS Setup program normally defines this anonymous user for you. You can define your own WWW anonymous user name, but you must be sure that the entry on the Service page matches the entry in the computer's (or Windows NT domain's) user list.

Note also the Password Authentication options. For the time being, stick to the Allow Anonymous option only, which means that all Web users are logged on as IUSR_MYMACHINENAME. Later in this chapter, we'll explain Windows NT Challenge/Response.

IIS Directories

Remember SlowSoft's Web site from [Chapter 34](#)? If you requested the URL <http://slowsoft.com/newproducts.html>, the newproducts.html file would be displayed from the slowsoft.com home directory. Each server needs a home directory, even if that directory contains only subdirectories. The home directory does not need to be the server computer's root directory,

however. As shown in Figure 35-3, the WWW home directory is really \WebHome, so clients read the disk file \WebHome\newproducts.html.

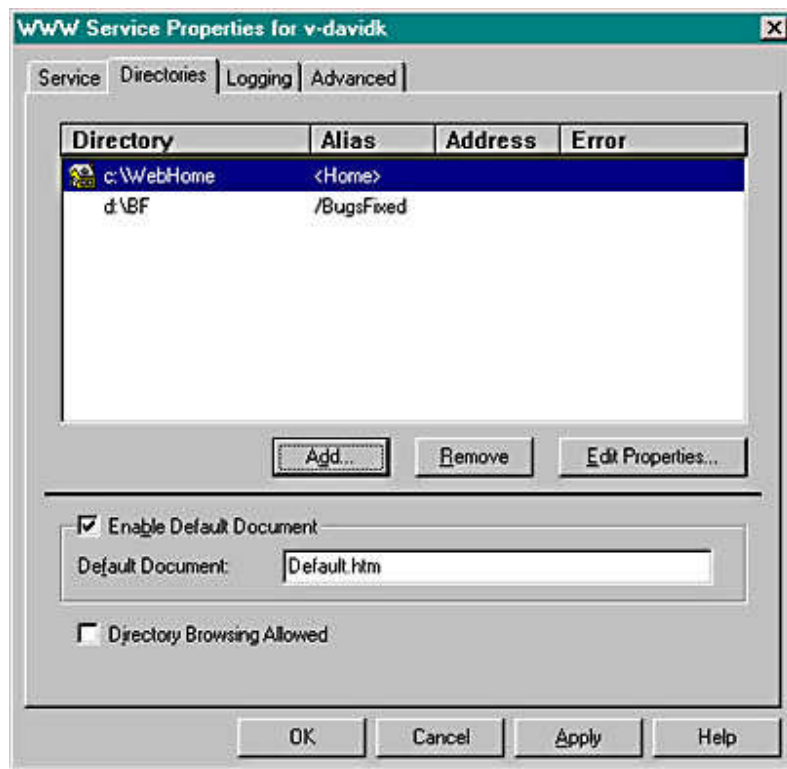


Figure 35-3. The \WebHome WWW home directory screen.

Your server could get by with a home directory only, but the IIS virtual directory feature might be useful. Suppose SlowSoft wanted to allow Web access to the directory \BF on the D drive. The screen above shows a virtual /BugsFixed directory that maps to D:\BF. Clients would access files with a URL similar to this: <http://slowsoft.com/BugsFixed/file1.html>.

If your computer was configured for multiple IP addresses (see the Control Panel Network icon), IIS would allow you to define virtual Web servers. Each virtual server would have its own home directory (and virtual directories) attached to a specified IP address, making it appear as though you had several server computers. Unfortunately, the IIS Web server listens on all the computer's IP addresses, so you can't run IIS simultaneously with the EX34A server with both listening on port 80.

the browsers can issue a blind request. As Figure 35-3 shows, Internet Service Manager lets you specify the file that a blind request selects, usually Default.htm. If you select the Directory Browsing Allowed option of the Directories page on the service property screen, browser clients can see a hypertext list of files in the server's directory instead.

IIS Logging

IIS is capable of making log entries for all connections. You control logging from the Internet Service Manager's Logging property page. You can specify text log files, or you can specify logging to an SQL/ODBC database. Log entries consist of date, time, client IP address, file requested, query string, and so forth.

Testing IIS

It's easy to test IIS with a browser or with any of the EX35A clients. Just make sure that IIS is running and that the EX35A server is not running. The default IIS home directory is \Winnt\System32\inetrv\wwwroot, and some HTML files are installed there. If you're running a single machine, you can use the localhost host name. For a network, use a name from the Hosts file. If you can't access the server from a remote machine, run ping to make sure the network is configured correctly. Don't try to build and run ISAPI DLLs until you have successfully tested IIS on your computer.

4.ISAPI Server Extensions

An ISAPI server extension is a program (implemented as a DLL loaded by IIS) that runs in response to a GET or POST request from a client program (browser). The browser can pass parameters to the program, which are often values that the browser user types into edit controls, selects from list boxes, and so forth. The ISAPI server extension typically sends back HTML code based on those parameter values. You'll better understand this process when you see an example.

Common Gateway Interface and ISAPI

Internet server programs were first developed for UNIX computers, so the standards were in place long before Microsoft introduced IIS. The Common Gateway Interface (CGI) standard, actually part of HTTP, evolved as a way for browser programs to interact with scripts or separate executable programs running on the server. Without altering the HTTP/CGI specifications, Microsoft designed IIS to allow any browser to load and run a server DLL. DLLs are part of the IIS process and thus are faster than scripts that might need to load separate executable programs. Because of your experience, you'll probably find it easier to write an ISAPI DLL in C++ than to write a script in PERL, the standard Web scripting language for servers.

CGI shifts the programming burden to the server. Using CGI parameters, the browser sends small amounts of information to the server computer, and the server can do absolutely anything with this information, including access a database, generate images, and control peripheral devices. The server sends a file (HTML or otherwise) back to the browser. The file can be read from the server's disk, or it can be generated by the program. No ActiveX controls or Java applets are necessary, and the browser can be running on any type of computer.

A Simple ISAPI Server Extension GET Request

Suppose an HTML file contains the following tag:

```
<a href="scripts/maps.dll?State=Idaho">Idaho Weather Map</a><p>
```

When the user clicks on *Idaho Weather Map*, the browser sends the server a CGI GET request like this:

```
GET scripts/maps.dll?State=Idaho HTTP/1.0
```

IIS then loads maps.dll from its scripts (virtual) directory, calls a default function (often named *Default*), and passes it the *State* parameter *Idaho*. The DLL then goes to work generating a JPG file containing the up-to-the-minute satellite weather map for Idaho and sends it to the client.

If maps.dll had more than one function, the tag could specify the function name like this:

```
<a href="scripts/maps.dll?GetMap?State=Idah  
o&Res=5">Idaho Weather Map</a><p>
```

In this case, the function *GetMap* is called with two parameters, *State* and *Res*.

You'll soon learn how to write an ISAPI server similar to maps.dll, but first you'll need to understand HTML forms, because you don't often see CGI GET requests by themselves.

HTML Forms—GET vs. POST

In the HTML code for the simple CGI GET request above, the state name was hard-coded in the tag. Why not let the user select the state from a drop-down list? For that, you need a form, and here's a simple one that can do the job.

```
<html>  
<head><title>Weathermap HTML Form</title>  
</head>  
<body>  
<h1><center>Welcome to the Satellite Weathermap  
Service</center></h1>  
<form action="scripts/maps.dll?GetMap" method=GET>  
  <p>Select your state:  
  <select name="State">  
    <option> Alabama  
    <option> Alaska  
    <option> Idaho  
    <option> Washington  
  </select>  
<p><input type="submit"><input type="reset">  
</form>  
</body></html>
```

If you looked at this HTML file with a browser, you would see the form shown in Figure 35-4.



Figure 35-4. *The Weathermap HTML Form window.*

The select tag provides the state name from a list of four states, and the all-important "submit" input tag displays the pushbutton that sends the form data to the server in the form of a CGI GET request that looks like this:

```
GET scripts/maps.dll?GetMap?State=Idaho HTTP/1.0
(various request headers)
(blank line)
```

Unfortunately, some early versions of the Netscape browser omit the function name in form-originated GET requests, giving you two choices: provide only a default function in your ISAPI DLL, and use the POST method inside a form instead of the GET method.

If you want to use the POST option, change one HTML line in the form above to the following:

```
<form action="scripts/maps.dll?GetMap" method=POST>
```

Now here's what the browser sends to the server:

```
POST scripts/maps.dll?GetMap
(various request headers)
(blank line)
```

```
State=Idaho
```

Note that the parameter value is in the last line instead of in the request line.

ISAPI DLLs are usually stored in a separate virtual directory on the server because these DLLs must have execute permission but do not need read permission. Clicking the Edit Properties button shown in Figure 35-3 will allow you to access these permissions from the Internet Service Manager, or you can double-click on a directory to change its properties.

Writing an ISAPI Server Extension DLL

Visual C++ gives you a quick start for writing ISAPI server extensions. Just select ISAPI Extension Wizard from the Projects list. After you click the OK button, your first screen looks like Figure 35-5.

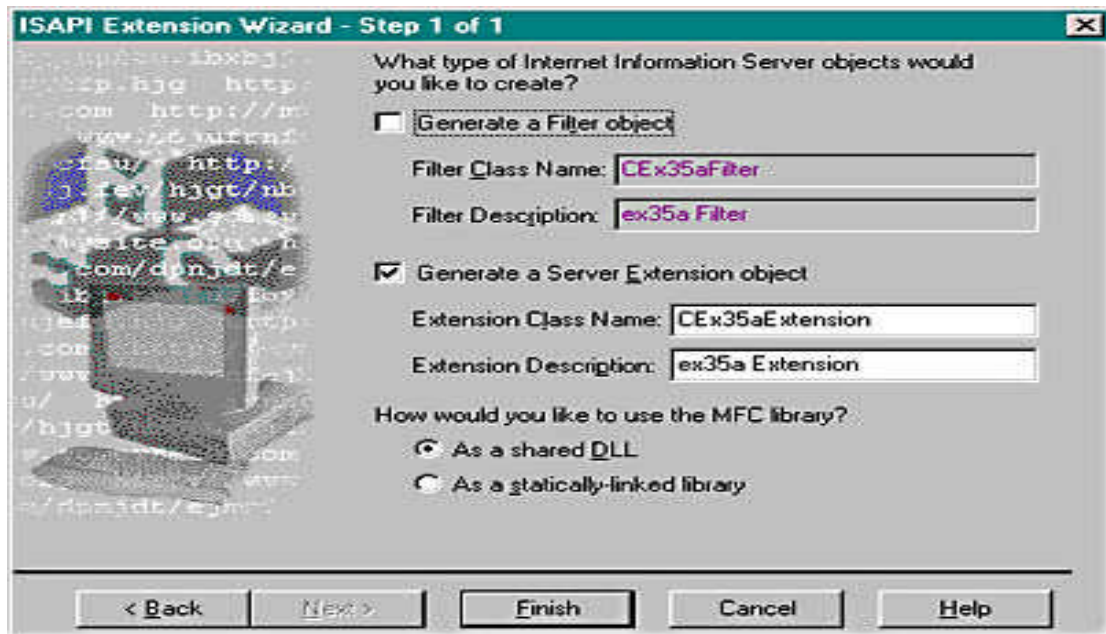


Figure 35-5. The Step 1 page of the ISAPI Extension Wizard.

Check the Generate A Server Extension Object box, and you've got a do-nothing DLL project with a class derived from the MFC *CHttpServer* class and a *Default* member function. Now it's time for a little programming.

You must write your ISAPI functions as members of the derived *CHttpServer* class, and you must write parse map macros to link them to IIS. There's no "parse map wizard," so you have to do some coding. It's okay to use the *Default* function, but you need a *GetMap* function too. First insert these lines into the wizard-generated parse map:

```
ON_PARSE_COMMAND(GetMap, CWeatherExtension, ITS_PSTR)
ON_PARSE_COMMAND_PARAMS("State")
```

Then write the *GetMap* function:

```
void CWeatherExtension::GetMap(CHttpServerContext* pCtxt, LPCTSTR
pstrState)
{
    StartContent(pCtxt);
    WriteTitle(pCtxt);
    *pCtxt << "Visualize a weather map for the state of ";
    *pCtxt << pstrState;
    EndContent(pCtxt);
}
```

```
}
```

This function doesn't actually generate the weather map (what did you expect?), but it does display the selected state name in a custom HTML file. The *CHttpServer::StartContent* and *CHttpServer::EndContent* functions write the HTML boilerplate, and *CHttpServer::WriteTitle* calls the virtual *CHttpServer::GetTitle* function, which you need to override:

```
LPCTSTR CWeatherExtension::GetTitle() const
{
    return "Your custom weather map"; // for browser's title
window
}
```

The MFC *CHttpServerContext* class has an overloaded << operator, which you use to put text in the HTML file you're building. Behind the scenes, an attached object of the MFC class *CHtmlStream* represents the output to the server's socket.

The MFC ISAPI Server Extension Classes

Now is a good time to review the three MFC classes that are used to create an MFC ISAPI server extension. Remember that these classes are for ISAPI server extensions only. Don't even think of using them in ordinary Winsock or WinInet applications.

CHttpServer

With the help of the ISAPI Extension Wizard, you derive a class from *CHttpServer* for each ISAPI server extension DLL that you create. You need one member function for each extension function (including the default function), and you need an overridden *GetTitle* function. The framework calls your extension functions in response to client requests, using the connections established in the parse map. The ISAPI Extension Wizard provides an overridden *GetExtensionVersion* function, which you seldom edit unless you need initialization code to be executed when the DLL is loaded.

One of the *CHttpServer* member functions that you're likely to call is *AddHeader*, which adds special response headers, such as Set-Cookie, before the response is sent to the server. (More on cookies later.)

CHttpServerContext

There's one *CHttpServer* object per DLL, but there is one *CHttpServerContext* object for each server transaction request. Your extension functions each provide a pointer to one of these objects. You don't derive from *CHttpServerContext*, so you can't easily have variables for individual transactions. Because different IIS threads can manage transactions, you have to be careful to perform synchronization for any data members of your *CHttpServer* class or global variables.

You've already seen the use of the *StartContent*, *EndContent*, and *WriteTitle* functions of the *CHttpServer* class plus the overloaded >> operator. You might also need to call the *CHttpServerContext::GetServerVariable* function to read information sent by the client in the request headers.

CHtmlStream

Most of the time, you don't use the *CHtmlStream* class directly. The *CHttpServerContext* class has a *CHtmlStream* data member, *m_pStream*, that's hooked up to the `>>` operator and serves as the output for HTML data. You could access the *CHtmlStream* object and call its *Write* member function if you needed to send binary data to the client. Because objects of the *CHtmlStream* class accumulate bytes in memory before sending them to the client, you need an alternative approach if your DLL relays large files directly from disk.