WINDOWS PROGRAMMING USING VISUAL C++

1. WINDOWS SDK PROGRAMMMING 3
1.1. WHAT IS A WINDOWS WINDOW? 3
1.2. THE WINDOWS SOFTWARE DEVELOPMENT KIT (SDK) 3
1.3. THE WINDOWS APPLICATION PROGRAMMING INTERFACES 4
1.4. THE WIN 16 AND WIN32 APIs 4
1.5. EVENT-DRIVEN PROGRAMMING AND THE MESSAGE LOOP 5
1.6. THE WinMain( ) FUNCTION 6
1.7. THE WINDOW PROCEDURE 7
1.8. WINDOW CLASSES AND WINDOW STYLES 8
1.9. MODULE-DEFINITION FILES 8
1.10. A MINIMAL SDK WINDOWS PROGRAM 9

2. AN INTRODUCTION TO MFC 17
2.1. WINDOWS VERSUS MFC 17
2.2. THE MICROSOFT FOUNDATION CLASSES 17

3. CONSTRUCTING AN MFC PROGRAM 20
3.1. WRITING THE MINIMUM MFC PROGRAM 20
3.2. RESPONDING TO WINDOWS MESSAGES 24

4. UNDERSTANDING MENUS 26
4.1. CREATING A MENU RESOURCE 26
4.2. DEALING WITH RESOURCE FILES 27
4.3. INTRODUCING THE MENU APPLICATION 30
4.4. EXPLORING THE MENU APPLICATION 30

5. GRAPHICS AND TEXT DRAWING 36
5.1. UNDERSTANDING DEVICE CONTEXTS 36
5.2. INTRODUCING THE PAINT1 APPLICATION 36
5.3. EXPLORING THE APPLICATION 43

6. DIALOG BOXES 50
6.1. INTRODUCING THE DIALOG APPLICATION 50
6.2. CREATING A DIALOG BOX RESOURCE 51
6.3. CREATING A DIALOG CLASS. 53
6.4. EXPLORING THE DIALOG APPLICATION 56
6.5. COMMON DIALOG CLASSES 63
6.6. USING OTHER CONTROLS WITH DIALOG BOXES. 64
6.7. INTRODUCING THE CONTROL1 APPLICATION 65
6.8. EXPLORING THE CONTROL1 APPLICATION 66

7. USING BIMAPS 73
7.1. INTRODUCING DEVICE-DEPENDENT BITMAPS 73
7.2. CREATING THE BITMAP APPLICATION 73
7.3. EXPLORING THE BITMAP APPLICATION 77

8. USING APPWIZARD TO CREATE AN MFC PROGRAM 79
8.1. UNDERSTANDING WIZARDS 79
8.2. CREATING MFC PROGRAM WITH APPLICATION WIZARD 79
8.3. RUNNING YOUR FIRST MFC APPLICATION 80
8.4. EXPLORING APPWIZARD'S FILES AND CLASSES 80

9. DOCUMENTS AND VIEWS 82
9.1. UNDERSTANDING THE DOCUMENT CLASS 82
9.2. UNDERSTANDING THE VIEW CLASS 83

9.3. AN EXAMPLE OF DOCUMENT-VIEW APPLICATION 84
9.4. PRINTING THE VIEW 90
10. EXERCISE 109

1. WINDOWS SDK PROGRAMMMING

Before application frameworks existed for developing Windows application, software developers were forced to use the Windows Software Development Kit (SDK). This chapter describes the structure of an SDK application. It provides a relatively low level of understanding of the SDK way of doing things and makes clear how Windows operates with in an application context. Understanding traditional Windows programs written in C can better prepare you for understanding how MFC does things.

The following section discuss some of the most important internal structures and operations that Windows programs use and that you should understand to become proficient with MFC programming. We will look at the following:

- The Windows SDK and APIs
- Event- driven programming , messaging, and the traditional message loop
- Window classes and window style
- The WinMain () function
- Window procedures

These topics are vital to a solid understanding of MFC.

1.1. WHAT IS A WINDOWS WINDOW?

Before we get too deep in the details of Windows programming and how to create and display windows, let's talk about the visible window itself. Figure below shows a typical Windows 95- style window. There are some major differences between the look of this window and those that appear in 16- bit Windows 3.1x.

Fig. 1
A typical Windows 95- style window

All windows have a border that may or may not allow resizing the window, and may even be invisible. Most windows have a title bar, a System menu, and minimize and maximize buttons. These items were all present in Windows3.1x. The close button in the upper right corner of a window is brand new for Windows 95 and Windows NT4.0. Also new is the sizing box in the lower- right corner. For the user, the most important part of the window is the client area. This is where a Window 95 programs typically displays all the information and controls with which the user interact. Many windows also contain scroll bars to let users scroll this data horizontally or vertically.

1.2. THE WINDOWS SOFTWARE DEVELOPMENT KIT (SDK)

The Windows SDK- is an acronym for Software Development Kit (SDK). Windows SDK programming just take some getting used to. It takes a little time and study to grasp what's going on in there, behind that enchanting graphical user interface (GUI). The Windows SDK has been with us for many year and has set the stage for modern Windows programming with MFC.

The SDK is simply a set of tools designed to help C programmers create Windows application. The main problem with the SDK is that the tool kit (and the task of written procedural Windows programs in C) has become a bit dated. The Windows SDK consists of the following elements:

- A rather large set of books describing Windows functions, message, structure, macros and resources.
- Various tools, including a dialog editor and an image editor
- Online help files
- A set of Windows libraries and header files

- Sample Windows programs in C

Many of these have been consolidated or replaced with more modern counterparts. For example, Microsoft's Visual C++ combines the image editor, dialog editor, help files and other tools into one package. The same is true of other compiler vendors like Borland, Watcom, and Symantec. The basic elements still there, they've simply been repackaged and modernized. The latest incarnation of this tool kit is the Win 32 SDK. The main difference between the SDK and MFC is the difference between C and C++. Using MFC requires that you make the tradition to C++ and OOP.

The new paradigm of Windows programming is OOP, as are the new flavors of Windows itself (Windows 95 and Windows NT). This being the case, you might be surprised to learn that programs written in C with the SDK have a lot in common with those written with MFC. This is because the new flavors of Windows still work in basically the same way as their predecessors; MFC just hide a lot of the complexity from the programmer. This is a good thing because the underlying system keeps getting more and more complicated.

1.3. THE WINDOWS APPLICATION PROGRAMMING INTERFACES

The term API is an acronym for application programming interface. An application programming interface (API) is simply a set of function calls in one program (or set of related programs) a programmer uses to create other programs. The internals of the functions don't have to be known, just the function prototypes and return values. The thing that turns a set of functions into an API is that the functions have documented specifications that can be used by anyone.

The Windows APIs are of two basic varieties:

- APIs for 16- bit Windows such as Window 3.1 (Win16APIs)
- APIs for 32-bit Windows such as Windows 95 and Windows NT (Win32 APIs)

1.4. THE WIN 16 AND WIN32 APIs

The Win 16 and Win32 APIs are similar in most respect, but the Win 16 API can be considered subset of the Win32 API. The Win32 API contains almost everything that the Win 16 API has, and much more. At its core, each relies on three main components to provide most of the functionality of Windows. These core API components are described in the table below.

The major component of the Win 16 and Win32 APIs

Win 16 API	***Win32 API***	***Description***
USER.EXE	USER32 . DLL	The USER components is responsible for window management, including messages, menus, cursors, communications, timers, and most other functions not related to actually displaying windows, but to controlling them.
GDI.EXE	GDI32.DLL	The GDI components is the Graphics Device Interface: it takes care for the user interface and graphics drawing, including Windows metafiles, bitmaps, device contexts, and fonts.
KRNL386.EXE	KERNEL32.DLL	The KERNEL components handles the low - level functions of memory, task, and resource management that are the heart of Windows.

Although the Win 16 version of these components have . EXE extensions, they are actually all DLLs and cannot execute on their own.

Several other components help fill in the gaps left by these big three and are also considered to be a part of the Windows APIs. These components include DLLs for common dialog boxes, printing, file compression, version control, and multimedia support, among others.

1.5. EVENT-DRIVEN PROGRAMMING AND THE MESSAGE LOOP

Windows is a message-driven (or event-driven) operating system (OS). Event-driven simply means that every part of the OS communicates with every other part - and with applications- through Windows messages. Events occur in response to message being passed between windows and in response to user interactions with the OS and applications. One of your major job as a Windows programmer is to respond to those events.

If you are come from a DOS programming background, dealing with command-line switches and batch file, please realize that programming for Windows required new mindset. Although your application may have functions A, B and C, you can't be certain that they will execute in Windows in any certain order. Windows programs must deal with keyboard and mouse input that can call any one of numerous functions.

1.5.1. The Event-Driven Model

Windows operates on the principle of event-driven program execution. If you've come from a top-down, procedural programming backward, the concept of event-driven programming may seem a bit foreign to you. Using the traditional procedural programming model of top-down design and bottom-up execution, the application developer is in complete control. Flow of program execution follows predefined path set down by the programmer. Function A is always followed by function B, which is always followed by function C, and so forth. The application developer determines the specific order in which things happen within the flow of program execution.

That doesn't mean that users of procedural programs don't have any input, because of course they do. The point is that, in general, if the application is executing function C, for example, you can rest assured that functions A and B have already executed (because it is impossible to get to function C without first going through functions A and B). This single-flow-of- execution style of programming has its advantages and its drawbacks. Programming of this type can certainly reduce the need for error checking: it is also extremely unresponsive to the needs and expectations of today's typical end user. The operating system simply executes the program and then waits for it to finish. The main difference about programming for Windows 95 is that the operating system not only executes your program, it talks to it! In fact, your application talks back - they communicate.

Windows applications use the event-driven model, which is very different from procedural programming. The application must set up variables and structures and perform other initialization, just as a typical procedural program does. At some point, the initialization finishes and activity ceases. The Windows application just sits there, waiting for user input of one form or another. This input can be in the form of a mouse click or a keystroke. As soon as the user provides input: a cascade of events follows, and the application responds. The trick is to try and take into account the possible actions the end user may take and to be there waiting for them when they happen. That's not to say that every conceivable action must be accounted for, only those that make sense within the context of your application.

1.5.2. An Event-Driven Example

As an example, assume that the user is going to click an About menu item, which you considerately provided in the menu bar of your new set Windows application. Windows is watching the state of the system very closely and listening for any stirrings of user input. Then, it happens: The user clicks a menu item! Windows, being the watchful OS that it is, notices the click and sends a message to your application that say, in effect, "The user just clicked your About menu!". You had anticipated that the user would click the About menu item some day. Your application responds to the message by executing a function that fires up a dialog box, proudly displaying information about your application.

That's really all there is to it. Your job is to anticipate what user will do with your application's user interface objects and have a function lying in wait for them, ready to execute at the appropriate time. Just when that time is, no one but the user can really say. You can't possibly known which message will come at any given time.

What do you do if a situation arises for which you don't anticipate the user's action? The solution is simple: Ignore the user input and let Windows handle it. Later in this chapter , you'll see how all this come together to produce and event- driven, message- handling program.

Windows application can send and receive message to the OS and to and from other applications: there are hundreds of message to deal with (although a typical application responds to only a small portion of these message). Figure below shows the basic flow of a Windows application and Windows messaging.

Fig.2
Basic flow of a Windows application and Windows messaging.

Windows provides a holding area for your application's messages called a message queue. Each process currently executing in the OS gets its own message queue. All messages sent by Windows (and by other applications to yours) are stored in this queue until called for. An SDK program calls for Windows messages in a special loop called a message loop. A basic message loop looks like this:

```
while( GetMessage(&msg, 0,0,0) )
{
     TranslateMessage(&msg);
     DispatchMessage(&msg);
}
```

The message loop is simply a while() loop that runs until the program receives a message to terminate execution. Inside the while( ) loop, the Windows API functions GetMessage( ) is called with each iteration of the loop. The function prototype for GetMessage( ) is declared in WINDOWS.H as follows:

```
BOOL GetMessage(MSG FAR* lpmsg, HWND hwnd,
                UNIT uMsgFilterMin, UNIT  uMsgFilterMax)
```

The GetMessage( ) and DispatchMessage( ) API functions each take a pointer to a MSG structure as their only parameter. TranslateMessage ( ) performs some keyboard translations. DispatchMessage( ) send the message to the window's windows procedure.

The GetMessage() function retrieves the next message waiting in the program's message queue

1.6. THE WinMain( ) FUNCTION

In a conventional DOS program written in C, a functional called *main( )* serves as the initial entry point into the program . Windows programs need an entry point too, and a function called *WinMain( )* is what Windows searches for, by name, as the initial entry point for a windows application at run time. The function prototype for WinMain() is as follows :

```
int  FAR PASCAL WinMain (HINSTANCE  hInst, HINSTANCE  hPrevInst,
                         LPSTR 1pCmdLine, int nCmdShow);
```

Don't be confused if you see various forms of this prototype in Windows C programs you come across. An equivalent function prototype for the WinMain( ) function might be as shown here.

```
INT APIENTRY WinMain (HINSTANCE hInst, HINSTANCE hPrevInst,
                      LPSTR 1pCmdLine, INT nCmdShow)
```

The underlying structure of these two function prototypes is the same because of Windows' extensive use of data type aliasing. Some examples of data type aliasing show here:

- FAR PASCAL and APIENTRY (these are the same)

- MSG FAR* and LPMSG (these are the same)

These statements means the same thing to the compiler because the C++ preprocessor substitutes the alias for the underlying actual data type during the compilation process. The functionality you code within WinMain( ) can be as simple or as complex as you see fit but, at a minimum, the function must perform the following tasks:

- Initialize the application

- Initialize and create application windows

- Enter the application's message loop

1.7. THE WINDOW PROCEDURE

A *Window procedure* is the control center of a traditional Windows SDK program written in C. Each window created in a Windows program contain its own message loop and its own window procedure to handle message for that particular window. When an application's main message loop calls *DispatchMessage( ),* Windows routes the message to the application's window procedure. A window procedure is simply a function, typically named WndProc( ), that does the message processing for a window. It is here that the program decides what to do with the information contained within the MSG structure of the message currently being processed. The application is said to handle a message, or to respond to an event. A window procedure is defined as follows.

```
LRESULT CALLBACK WndProc(HWND hWnd, UNIT message,
                         WPARAM  wParam, LPARAM 1Param) ;
```

The parameters for a window procedure are the same as the first four members in the MSG structure. These parameters are parsed from the MSG structure and sent to the window procedure DispatchMessage(). All message for an application are passed form the loop to the main window procedure.

A window procedure first looks at the UINT message parameter to determine the message being received. Depending on the specific message, the procedure can gather more information by scrutinizing the WPARAM and LPARAM parameters, which contain additional information. The processing of Windows message occurs within a (usually very large) *switch - case* statement block. A window procedure's message handler can get very convoluted because a message can be processed at many levels and additional switch- case statements are often nested many levels deep. This is often the case with command message (WM_ COMMAND) that must be processed at more than one level. A simple message-handling block is given below:

```
switch (message)
{
   case WM_LBUTTONDOWN:   // left mouse button pressed

      MessageBeep(MB_ICONINFORMATION);
      MessageBox(hwndMain, szAboutLeft, "About",
                 MB_OK | MB_ICONEXCLAMATION);
      break;

   case WM_RBUTTONDOWN:   // right mouse button pressed

      MessageBeep(MB_ICONINFORMATION);
      MessageBox(hwndMain, szAboutRight, "About",
                 MB_OK | MB_ICONINFORMATION);
      break;

   case WM_DESTROY:       // the window has been destroyed

      PostQuitMessage(0);
      return 0;
```

```
      default:
         //Send unhandled messages off to Windows for processing
         return DefWindowProc (hwndMain, message, wParam, lParam);
   }
```

At the very least, an application's window procedure must check for the WM_DESTROY message and respond with the PostQuitMessage( ) API function, or the application won't shut down. Message that aren't explicitly handled in your code should be passed into Windows for default message processing. This is done by calling the DefWindowProc ( ) API function.

1.8. WINDOW CLASSES AND WINDOW STYLES

The visible window displayed on a computer monitor is just the tip of the iceberg. There's a lot of hidden data lurking beneath the GUI window you see. This data defines the size and position of a window as well as its colours, title bar and many other aspects of the window . The means of sorting all this information is a data structure called a window class (WNDCLASS) structure.

1.8.1. Window Class Structure (WNDCLASS)

The WNDCLASS structure is used to define all windows that you see on your monitor. The structure is fairly simple and it looks like this:

```
typedef struct _WNDCLASS
{
      UINT        style;          //window class style
      WNDPROC     lpfnWndProc;    //window procedure
      INT         cbClsExtra;     //extra class bytes
      INT         cbWndExtra;     //extra window bytes
      HANDLE      hInstance;      //instance handle
      HICON       hIcon;          //icon handle
      HCURSOR     hCursor;        //cursor handle
      HBRUSH      hbrBackground;  //background brush
      LPCSTR      lpszMenuName;   //menu resource
      LPCSTR      lpszClassName;  //window class name
}WNDCLASS;
```

1.8.2. Window Styles

In addition to the available window *class* style, there are also individual *window* styles. As you know, you can have more than one window in a Windows program. A window's style (or combination of style) determines not only its visual appearance but also its relationship to other windows in a program. All the button, list boxes, and other user interface gadgets you see on-screen are windows too.

1.9. MODULE-DEFINITION FILES

A module-definition file (.DEF) is a text file the linker uses when linking object modules into an executable program. In the past, every Windows program had to have a module-definition file for use by the linker. Although a module-definition file is no longer required for 32-bit Windows programs, you can include one if you want. A module-definition file defines the name of the application, its code and data segments, its memory requirements, and any functions exported by the application, its code and data segments, its memory requirements, and functions exported by the application.

```
------------------------------------------------------------
//SDK1.DEF - The module definition file for a minimal SDK

NAME           SDK1
DESCRIPTION    'Minimal SDK Program'
```

```
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'
CODE            MOVABLE DISCARDABLE
DATA            MOVABLE MULTIPLE
HEAPSIZE        1024
STACKSIZE       5120
EXPORTS         WndProc @1
```

A description of each keyword used in this module-definition file is shown in table below.

The description of module-definition file keywords

Statement	**Meaning**
NAME	This statement is required; it defines the name of the application as used by Windows
DESCRIPTION	This statement is optional; it places a text message into the application's executable file.
EXETYPE	This statement marks the executable file as a Windows executable file. A Windows application must specify: EXETYPE WINDOWS.
STUB	This statement specifies an optional DOS executable "stub" file linked into the beginning of a Windows executable file. The SDK supplies a standard stub (named WINSTUB.EXE) that displays the familiar *This program requires Microsoft Windows* warning message and terminates the program if a user tries to run the application without Windows. You can also create and use your own stubs for custom warning message or functionality.
CODE	This statement defines the memory attributes of an application's code segment, allowing Windows to deal with memory management in various ways.
DATA	This segment defines the memory requirements of an application's data segment.
HEAPSIZE	This statement defines the size (in byte) of an application's local heap.
STACKSIZE	This statement defines the size (in bytes) of an application's stack.
EXPORTS	This statement defines the names of an application's exported functions, along with their ordinal values. All Windows callback functions, except *WinMain* ( ), are exported.

1.10.

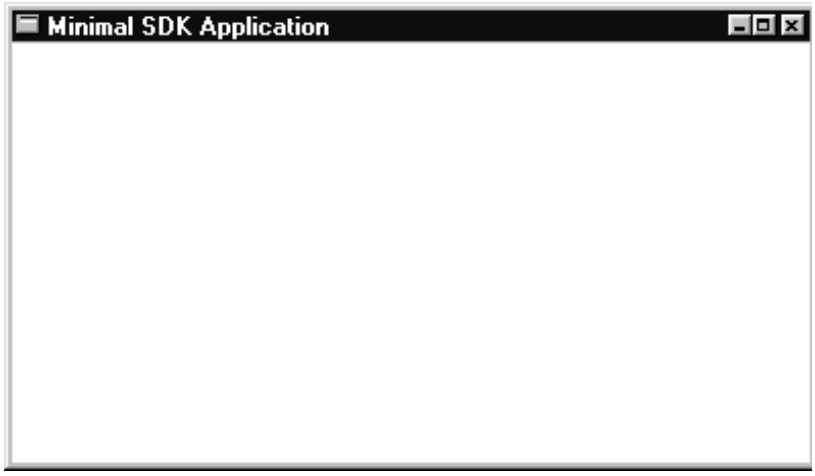
1.11. A MINIMAL SDK WINDOWS PROGRAM

Only two functions are *required* by a Windows program:

- WinMain ( )
- The application's window procedure

As you've seen, the WinMain ( ) function serves as the program entry point, handles initialization, and sets up the message loop. The window procedure handles and processes Windows message for the application.

Now that you've seen what all the main components of a traditional Windows application do, let's take a look at a minimal Windows SDK program written in C. This small program serves no real purpose other than to bring all the pieces together into a real, working Windows application. It has a lot of overhead code and handles only three Windows message (two mouse-click message and a quit message). The figure below shows application's only window.



Listing below reveals the code for the above program:

```
/*****************************************************************
  Module   : SDK1.C

  Purpose  : Implementation of a minimal SDK program
*****************************************************************/

#include <windows.h>

// Function prototypes
//
INT PASCAL WinMain(HINSTANCE, HINSTANCE, LPSTR, INT);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

//
// Global variables
//
HINSTANCE ghInst;       // current instance

char szAppName[]  = "WinSdk";                    // The app name
char szAppTitle[] = "Minimal SDK Application";  // caption text


/*****************************************************************
  Function : WinMain(HINSTANCE, HINSTANCE, LPSTR, INT)

  Purpose  : Program entry point. Calls initialization function,
             processes message loop.
*****************************************************************/

INT PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpszCmdParam, INT nCmdShow)
{
   HWND      hwndMain;     // main window handle
```

```
   MSG       message;      // window message
   WNDCLASS  wc;           // window class

   //
   // If no previous instance register the new window class
   //
   if (!hPrevInstance) // Are other instances of the app running?
   {
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc   = WndProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = hInstance;
      wc.hIcon         = LoadIcon(NULL,IDI_APPLICATION);
      wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
      wc.hbrBackground = GetStockObject(WHITE_BRUSH);
      wc.lpszMenuName  = NULL;
      wc.lpszClassName = szAppName;

      // Register the window class with Windows
      RegisterClass (&wc);
   }

   //
   // Create the app window
   //
   hwndMain = CreateWindow (
      szAppName,             // window class name
      szAppTitle,            // 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
   );

   //
   // Make the window visible and update its client area
   //
   ShowWindow (hwndMain, SW_SHOWMAXIMIZED);// Show the window
   UpdateWindow (hwndMain);// Sends WM_PAINT message

   //
   // Enter the program's message loop
   //
   while (GetMessage (&message, NULL, 0, 0))
   {
      DispatchMessage (&message);
   }
   return message.wParam;
}

/*****************************************************************
  Function : WndProc(HWND, UINT, WPARAM, LPARAM)

  Purpose  : Processes messages
*****************************************************************/

LRESULT CALLBACK WndProc(HWND hwndMain,  // window handle
                         UINT message,   // type of message
                         WPARAM wParam,  // additional information
                         LPARAM lParam)  // additional information
```

```
{
   //
   // Some local variables
   //
   char szAboutLeft[] = "This is a minimal Windows SDK program.\n"
                        "You've pressed the left mouse button!";

   char szAboutRight[] = "This is a minimal Windows SDK program.\n"
                         "You've pressed the right mouse button!";
   //
   // message handlers
   //
   switch (message)
   {
      case WM_LBUTTONDOWN:   // left mouse button pressed

         MessageBeep(MB_ICONINFORMATION);
         MessageBox(hwndMain, szAboutLeft, "About",
                    MB_OKCANCEL | MB_ICONEXCLAMATION);
         break;

      case WM_RBUTTONDOWN:   // right mouse button pressed

         MessageBeep(MB_ICONINFORMATION);
         MessageBox(hwndMain, szAboutRight, "About",
                    MB_OK | MB_ICONINFORMATION);
         break;

      case WM_DESTROY:       // the window has been destroyed

         PostQuitMessage(0);
         return 0;

      default:
         //Send unhandled messages off to Windows for processing
         return DefWindowProc (hwndMain, message, wParam, lParam);
   }
   return 0L;
}
```

1.11.1. Understanding the Program

The code for this program is straightforward, but it deserves a closer examination. We start off with *#include <windows.h>*. This statement includes the main Windows header file (WINDOWS.H) that, in turn, includes all the Windows headers that contain structures, macros, API prototypes and everything else a typical SDK program might need. Then the two mandatory SDK functions are declared: the program entry point, WinMain(), and the application's window procedure, in this case WndProc().

Three application global variables are declared:

Variable	Description
HINSTANCE ghInst	This is the instance handle of the current instance.
char szAppName[ ]	This is the name used by the Windows to identify the application
char szAppTitle [ ]	This is the text that appears in the application window's title bar.

1.11.2. The Program Entry Point: WinMain( )

The entry point for a Windows application is the WinMain( ) function. In this sample program, WinMain() contains all the application initialization and window initialization and creation code; it also contain the

application's message loop. WinMain( ) declares a variable for the main window handle (hwndMain), a message structure variable (message), and a window class variable (wc).

1.11.3. Initializing and Registering the Window Class

Windows allows multiple instance of a single application to run concurrently : each application window has a window class. A window class in an application must be initialized and registered with Windows only once: when the first instance begins execution. If there are no previous instances of the application running, the WNDCLASS structure variable *wc* is initialized with values and registered with Windows by calling the RegisterClass( ) API function with a pointer to *wc* as the only parameter.

1.11.4. Creating the Application Window

Next, the application window is created by calling , appropriately enough, the API function CreateWindow ( ). As you might guess, creating windows is one of the most important activities of a Window Program. Because it is so important, let's look at the CreateWindow() function in more detail. CreateWindow() takes 11 parameters (see the following table)

The Parameters of the CreateWindow( ) function.

Parameter	Description
LPCTSTR 1pClassName	A pointer to a Null- terminated string specifying the window class name. This name can be any name previously registered with the RegisterClass( ) function, or can be any of the predefined Windows control class names.
LPCTSTR 1pWindowName	A pointer to a NULL- terminated string specifying the name of the window being created.
DWORD dwStyle	A combination of window styles and control styles that specifies the style of the window being created.
int x, int y	These parameters specify the initial position of the window. This is the window's upper-left corner (in screen coordinates) for overlapped or pop-up windows: it is the upper-left corner (relative to the parent window's upper-left corner) for child windows. The value CW_USEDEFAULT lets Windows select the default position for the upper-left corner of the window.
int nWidth, int nHeight	These parameters specify the width and height of the window, respectively. The value CW_USEDEFAULT lets Windows select a default width and height for the window.
HWND hWndParent	Specifies the parent window, if any, of the window being created.
HMENU hMenu	Identifies a menu or specifies a child window identifier, depending on the window style. For overlapped or pop-up windows, this parameter identifies the menu resource handle for the window. For child windows, this parameter is the child window identifier. (Note: All child windows with the same parent window must have unique identifiers).
Handle hInstance	Identifies the instance handle of the module to be associated with the window.
LPVOID 1pParam	A pointer to the value of a CREATESTRUCT member referenced by the lParam parameter of the WM_CREATE message for single document interface windows; a pointer to CLIENTCREATESTRUCT structure for multiple document interface window.(Note: this parameter is often set to NULL.)

The penultimate step in WinMain( ) is to show the newly created window by calling the ShowWindow( ) and UpdateWindow ( ) API functions:

```
// Make the window visible and update its client area
//
ShowWindow (hwndMain, SW_SHOWMAXIMIZED);// Show the window
UpdateWindow (hwndMain);// Sends WM_PAINT message
```

The final step is t enter the application's message loop:

```
// Enter the program's message loop
//
```

```
while (GetMessage (&message, NULL, 0, 0))
{
   DispatchMessage (&message);
}
return message.wParam;
```

1.11.5. Handling Message in the Window Procedure

The application's window procedure, WndProc( ) in this example, first declares two character strings (szAboutLeft and szAboutRight) for the display of text message to the user. The Display of these string constitutes the only functionality of the program. These strings are displayed in response to the receipt of Windows messages indicating that the user has clicked either the left or the right mouse button in the application window's client area.

To trap and respond to a message, the program must first test the value of the *message* parameter (which contains the actual unknown WM_ Windows message value) in a switch statement:

```
switch (message) //test the message value
```

If a message handler exists for the value, the corresponding `case` statement responds accordingly. To trap and respond to the left mouse click message, the application does this:

```
case WM_LBUTTONDOWN:   // left mouse button pressed

   MessageBeep(MB_ICONINFORMATION);
   MessageBox(hwndMain, szAboutLeft, "About",
              MB_OKCANCEL | MB_ICONEXCLAMATION);
   break;
```

This code is executed when the message parameter has the value WM_LBUTTONDOWN. The right mouse click is handled in exactly the same way, trapping the WM_RBUTTONDOWN message:

```
case WM_RBUTTONDOWN:   // right mouse button pressed

   MessageBeep(MB_ICONINFORMATION);
   MessageBox(hwndMain, szAboutRight, "About",
              MB_OK | MB_ICONINFORMATION);
   break;
```

In response to the clicks, the application calls the API functions MessageBeep ( ) and MessageBox( ). If the user's PC is sound enabled, MessageBeep ( ) plays the wave file associated with the Windows-Asterisk sound in the Sound applet in the user's Control Panel. The wave file that sounds is determined by the MB_ICONASTERISK parameter. If the user's PC is *not* sound enabled, MessageBeep ( ) just beeps the PC speaker. The MessageBeep ( ) function can take any of the parameters listed in table below:

The possible parameters for the MessageBeep ( ) function

Parameter	**Meaning**
0xFFFFFFFF	Standard beep using the PC speaker
MB_ICONASTERISK	SystemAstrisk
MB_ICONEXCLAMATION	SystemExclamation
MB_ICONHAND	SystemHand
MB_ICONQUESTION	SystemQuestion
MB_OK	SystemDefault

MessageBox ( ) displays one of the text messages, szAboutLeft or szAboutRight, according to the message being handled. If the user clicks the close button or chooses Close from the System menu, Windows posts a WM_DESTROY message to the message queue to destroy the application window.

```
case WM_DESTROY:       // the window has been destroyed

   PostQuitMessage(0);
   return 0;
```

The PostQuitMessage() API function posts WM_QUIT message to the queue. This results in the GetMessage() function within the message loop finding the WM_QUIT message and terminating execution of the application.

During application execution, any unhandled messages are sent back to Windows through the DefWindowProc() API function to provide default processing:

```
return DefWindowProc (hwndMain, message, wParam, lParam);
```

2. AN INTRODUCTION TO MFC

Microsoft Foundation Class (MFC) Library is a collection of C++ classes for Microsoft Windows Operating System. MFC library simplifies the writing of Windows applications. Using Windows SDK programming, you'll need 80 or 90 lines of code just to get an empty window on-screen. Using MFC, you can get your first window on-screen with only a few lines of code.

2.1. WINDOWS VERSUS MFC

Windows applications usually have a main window with a menu bar, scroll bars, sizing buttons, and other controls all of which are handled to a great extent by Windows. For example, when you create a window with a menu bar, your program doesn't need to control the menu. Windows does this for you, sending your program a message whenever the user selects an item in the menu.

A Windows program can receive hundreds of different messages while it's running. Your application determines whether to respond to these messages or to ignore them. If the user clicks the right button of the mouse in your window, for example, your program gets a message (`WM_RBUTTONDOWN`). If the program determines that the user clicked on something important, the program can handle the message, performing the function the user requested. On the other hand, the program can simply ignore the message and let Windows take care of it.

Learning to program Windows, although not especially difficult, takes a lot of time and practice. Before you program your first application, you should be familiar with at least the most used functions in the API so that you know the tools you have at your disposal.

MFC simplifies the process of writing Windows applications by hiding many of the details inside custom window classes. By using MFC, you can create a fully operational window with very few lines of code.

2.2. THE MICROSOFT FOUNDATION CLASSES

MFC includes many classes you can use to write your applications more quickly and easily. These classes represent objects from which a Windows application is created, such as windows, dialog boxes, menu bars, window controls, and many more.

MFC provides the following main categories of classes:

- ⇒ Applications
- ⇒ Windows
- ⇒ Menus
- ⇒ Dialog boxes
- ⇒ Documents and views
- ⇒ Controls
- ⇒ Graphics
- ⇒ Archival and file
- ⇒ Various support classes

Fig.: MFC class hierarchy chart

2.2.1. The CObject Class

If you examine the MFC class hierarchy, you'll see that almost every class in the library is derived from the `CObject` class. `CObject` is the base for most MFC classes, so it must contain only the functionality common to all of the classes. With so many diverse classes in MFC, this commonality must be very general in nature.

In fact, `CObject` includes only eight member functions, including its constructor and destructor. These member functions enable the class to perform basic serialization (saving and reading data from and to storage), to provide run-time class information, and to output diagnostic information.

Although `CObject` acts as the base class for the majority of MFC classes, you can derive your own custom classes from `CObject` and, thus, automatically acquire built-in support for the functionality built into

CObject most notably the capability to serialize your custom object. Any custom class you want to develop that needs to save and load data should be derived, directly or indirectly, from CObject.

2.2.2. The Application Class

Every MFC program begins life as an application object. You derive this application object from MFC's CWinApp class, which provides initialization, runtime, and ending services for your Windows program. More specifically, CWinApp registers, creates, and shows an application's main window; sets the application's message loop running (so that the application can interact with the user and Windows); and deletes the application when the user is finished. CWinApp also provides some additional services, such as providing access to the application's command line, handling file activity, processing Windows messages, and detecting when Windows is idle. Every MFC program instantiates a CWinApp-derived object.

2.2.3. The Window Classes

When you create a basic MFC application, it's up to you to tell MFC the type of main window it should create for the application. This is where MFC's window classes come in. By forcing your MFC application to create your custom main window, you can add all the functionality you need to your application. To make this task easier for you, MFC features several different types of windows like Frame windows, View windows, MDI windows, Dialog boxes etc..

All of the window classes are derived from the CWnd base class, which supplies the basic functionality required by any window class. This functionality includes initialization, positioning and sizing, painting, coordinate mapping, scrolling, message handling, and much more. The CWnd class, however, has only one public data member, m_hWnd, which is the handle of the window with which the class is associated. Although you can derive your own window classes directly from CWnd, you're more likely to use one of MFC's specialized window classes.

2.2.4. Frame Windows

Frame windows, represented by the CFrameWnd class, supply all the functionality of the CWnd class and also handle document/view matters, the menu bar, control bars, accelerators, and more. The CFrameWnd class contains only two public data members, m_bAutoMenuEnable and rectDefault, which controls whether menu items are enabled or disabled automatically and controls the window's starting size and position, respectively.

In most cases, you'll derive the main window class for an SDI (single-document interface) application from the CFrameWnd class. Such a frame window usually has a complete set of window controls such as minimize, maximize, restore, and close buttons and holds the application's menu bar, toolbar, and status bar. However, MFC is flexible enough that you can create just about any type of frame window that you need from the CFrameWnd class.

2.2.5. View Windows

To make it easier to handle the data associated with an application's window, MFC offers the document/view architecture. The CDocument classis responsible for storing a window's data, whereas the CView displays the data, as well as enables the user to manipulate the data.

The CView class provides all the functionality needed by a generic view window. This includes displaying data in the frame window's client area and printing data. A document can be associated with more than one view. Although a document can have more than one view, an individual view must always be associated with only a single document.

MFC derives many specialized view classes from the general CView class. These include CCtrlView, CEditView, CListView, CRichEditView, CTreeView, CScrollView, CFormView, CDaoRecordView, and CRecordView. You can tell by the class names what many of these classes do.

2.2.6. MDI Windows

Most document-oriented Windows applications enable the user to open more than one document simultaneously. MFC provides the CMDIFrameWnd class to encapsulate the extra functionality needed by Multiple Document

Interface frame windows. This extra functionality includes handling the MDI client window, which provides an area for MDI child (document) windows; managing the main and child-window menu bars; forwarding messages to MDI child windows; and arranging MDI child windows within the client window.

In order to display the different documents associated with the MDI application, you need to create MDI child windows, which are represented in MFC by the `CMDIChildWnd` class. Just like regular frame windows, MDI child windows can contain view windows (instantiated from `CView`) that depict the data stored in a document object (instantiated from `CDocument`).

2.2.7. Dialog Boxes

Most Windows applications rely on dialog boxes to retrieve information from the user. Because dialog boxes are so important in Windows programming, MFC features a whole set of classes dedicated to these specialized windows. MFC features a basic dialog box class, `CDialog` which handles all the details of constructing, executing, and closing a dialog box, including built-in responses for the most often used buttons and a mechanism for transferring data to and from the dialog box. In addition, your MFC dialog boxes can be modal or modeless and can contain any number of controls.

MFC also includes classes for Windows' common dialog boxes. These dialog classes including `CColorDialog`, `CFileDialog`, `CFindReplaceDialog`, `CFontDialog`, and `CPrintDialog` enable users to select file names, find text strings, set up a printer, and choose colors and fonts.

You actually create your dialog box using a resource editor such as the dialog box editor included as part of the Microsoft Developer Studio, which comes with Visual C++. The dialog box you create with the editor comprises the visual elements of the dialog box, including the dialog box's main window and all of the controls that window contains. To add MFC functionality to the dialog box, you then derive a class from `CDialog`, attaching that class to the dialog box template you created with the dialog box editor.

To pass data to and from the dialog box, you create data members for the class and associate those data members with the controls for which they are to store data. You can then initialize the dialog box controls by setting these data members. Similarly, you can retrieve data from the dialog box's controls after it's closed by checking the contents of the associated data members.

3. CONSTRUCTING AN MFC PROGRAM

AppWizard is great for getting an application started quickly. However, because AppWizard does so much of the work for you, you never get to learn many of the details of MFC programming. Knowing how MFC really works, not only enables you to forgo AppWizard when you'd rather do the work yourself, but it also helps you understand AppWizard programs better so you can customize them more effectively.

Constructing a minimum MFC program involves the following steps:-

- **Creating an application -** MFC's CWinApp class represents a program's application object.
- **Creating a frame-window class -** The `CFrameWnd` class represents an application's main window.
- **Creating a project workspace for your MFC application -** Project workspaces enable you to save and restore the settings that are important to your work habits and program.
- **Associating Windows messages with message response functions -** When Windows sends your application a message, the application can respond by calling a message response function.

3.1. WRITING THE MINIMUM MFC PROGRAM

To create a minimal MFC program you need only a few lines of code. The following list outlines the steps you must take to create the smallest, functional MFC program:

1. Create an application class derived from `CWinApp`.

2. Create a window class derived from `CFrameWnd`.

3. Instantiate a global instance of your application object.

4. Instantiate your main window object in the application object's `InitInstance()` function.

5. Create the window element (the Windows window) associated with the window object by calling `Create()` from the window object's constructor.

3.1.1. Creating the Application Class

The first step in writing an MFC program is to create your application class, in which you must override the `InitInstance()` function. Listing below shows the header file for your first handwritten application class, called `CFirstApp`.

```
////////////////////////////////////////////////////////
// FIRSTAPP.H: Header file for the CFirstApp class, which  //
//             represents the application object.          //
////////////////////////////////////////////////////////
class CFirstApp : public CWinApp
{
public:
    CFirstApp();
    BOOL InitInstance();
};
```

In the preceding code, you can see that the `CFirstApp` class is derived from MFC's `CWinApp` class. The `CFirstApp` class has a constructor and overrides the `InitInstance()` function, which is the function MFC calls to create the application's window object. Because the base class's `InitInstance()` does nothing but return a value of `TRUE`, if you want your application to have a window, you must override `InitInstance()` in your application class.

The following listing shows how the application class actually creates its main window object. This is the `CFirstApp` class's implementation file.

```
////////////////////////////////////////////////////////////
// FIRSTAPP.CPP: Implementation file for the CFirstApp class,   //
```

```
//               which represents the application object.       //
////////////////////////////////////////////////////////////////////
#include <afxwin.h>
#include "firstapp.h"
#include "mainfrm.h"
CFirstApp FirstApp;
CFirstApp::CFirstApp()
{
}
BOOL CFirstApp::InitInstance()
{
    m_pMainWnd = new CMainFrame();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    return TRUE;
}
```

The first thing to notice here is the line

```
#include <afxwin.h>
```

near the top of the file. AFXWIN.H is the header file for the MFC classes, so it must be included in any file that references those classes. Failure to include AFXWIN.H in an MFC program causes the compiler to display a long list of error messages, because it won't recognize the MFC classes you're using.

After the `include` statements, you can see where the program instantiates its global application object with the following line:

```
CFirstApp FirstApp;
```

The application object must be global to ensure that it is the first object instantiated in the program.

In the `CFirstApp` class, the constructor does nothing, although you can add initialization code to the constructor as needed. In this first program, it's the application class's `InitInstance()` function that's most important. MFC calls `InitInstance()` once, at the beginning of the application's run. `InitInstance()`'s main task is to instantiate the application's main window object, which the `CFirstApp` class does with the following line:

```
m_pMainWnd = new CMainFrame();
```

This line creates a new `CMainFrame()` object on the heap, calling the `CMainFrame()` class's constructor, where, the class creates the main window element. A pointer to the new window object gets stored in `m_pMainWnd`, which is a data member of the application class. You must be sure to initialize `m_pMainWnd` properly, as it's through this pointer that the application class maintains contact with its main window.

After instantiating the window object, the window element must usually be displayed. This is done by calling the window object's `ShowWindow()` function, like this:

```
m_pMainWnd->ShowWindow(m_nCmdShow);
```

`ShowWindow()`'s single argument indicates how the window should be displayed initially. You need only pass along the application object's `m_nCmdShow` data member, which contains the value of the `nCmdShow` parameter that Windows passed to `WinMain()`.

When an application starts up, there are several ways that its window can be initially displayed. These different display styles are represented by a set of constants defined by Windows, including `SW_HIDE`, `SW_MINIMIZE`, `SW_SHOWMAXIMIZED`, `SW_SHOWMINIMIZED`, and `SW_SHOWNORMAL`, among others. With MFC, you don't need to worry about these different styles unless you want to do something special with a window. Just passing `m_nCmdShow` to `ShowWindow()` as the display style shows the window in the default manner.

3.1.2. Creating the Frame-Window Class

Now that you know how to create your application class, which is responsible for instantiating your main window. Listing below is the header file for the `CMainFrame` class, which represents the `CFirstApp` class's main window.

```
////////////////////////////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which  //
//            represents the application's main window.    //
////////////////////////////////////////////////////////////
class CMainFrame : public CFrameWnd
```

```
{
public:
    CMainFrame();
    ~CMainFrame();
};
```

`CMainFrame` is derived from `CFrameWnd` class. Although the `CMainFrame` class's header file doesn't provide much information how the application's main window is created. That information lies in the class's implementation file, which is shown below.

```
//////////////////////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame     //
//              class, which represents the application's  //
//              main window.                               //
//////////////////////////////////////////////////////
#include <afxwin.h>
#include "mainfrm.h"
CMainFrame::CMainFrame()
{
    Create(NULL, "First MFC Application");
}
CMainFrame::~CMainFrame()
{
}
```

In the class's constructor, you can see that the application's window element is created by a simple call to the window class's `Create()` member function, like this:

```
Create(NULL, "First MFC Application");
```

`Create()` really requires eight arguments, although all but the first two have default values. `Create()`'s prototype looks like this

```
BOOL Create(LPCTSTR lpszClassName,
            LPCTSTR lpszWindowName,
            DWORD dwStyle = WS_OVERLAPPEDWINDOW,
            const RECT& rect = rectDefault,
            CWnd* pParentWnd = NULL,
            LPCTSTR lpszMenuName = NULL,
            DWORD dwExStyle = 0,
            CCreateContext* pContext = NULL);
```

`Create()`'s arguments are:

- Class's name
- Pointer to the window's title
- The window's style flags
- The window's size and position (stored in a `RECT` structure)
- Pointer to the parent window
- Pointer to the window's menu name
- Any extended style attributes
- Pointer to a `CCreateContext` structure (which helps MFC manage the document and view objects)

If you use `NULL` for the class name, MFC uses its default window class. As for the title string, this is the title that appears in the window's title bar. If you want to create different types of windows, you can use different style flags for `dwStyle`. For example, the line

```
Create(NULL, "First MFC Application", WS_OVERLAPPED | WS_BORDER | WS_SYSMENU);
```

creates a window with a system menu, a Close button, and a thin border. Because such a window is missing the Minimize and Maximize buttons, as well as the usual thick border, the user cannot change the size of the window but can only move or close it. The styles you can use include `WS_BORDER`, `WS_CAPTION`, `WS_CHILD`, `WS_MAXIMIZEBOX`, `WS_MINIMIZEBOX`, `WS_THICKFRAME`, and more.

By changing the default values of the fourth argument, you can position and size a window. For example, the line

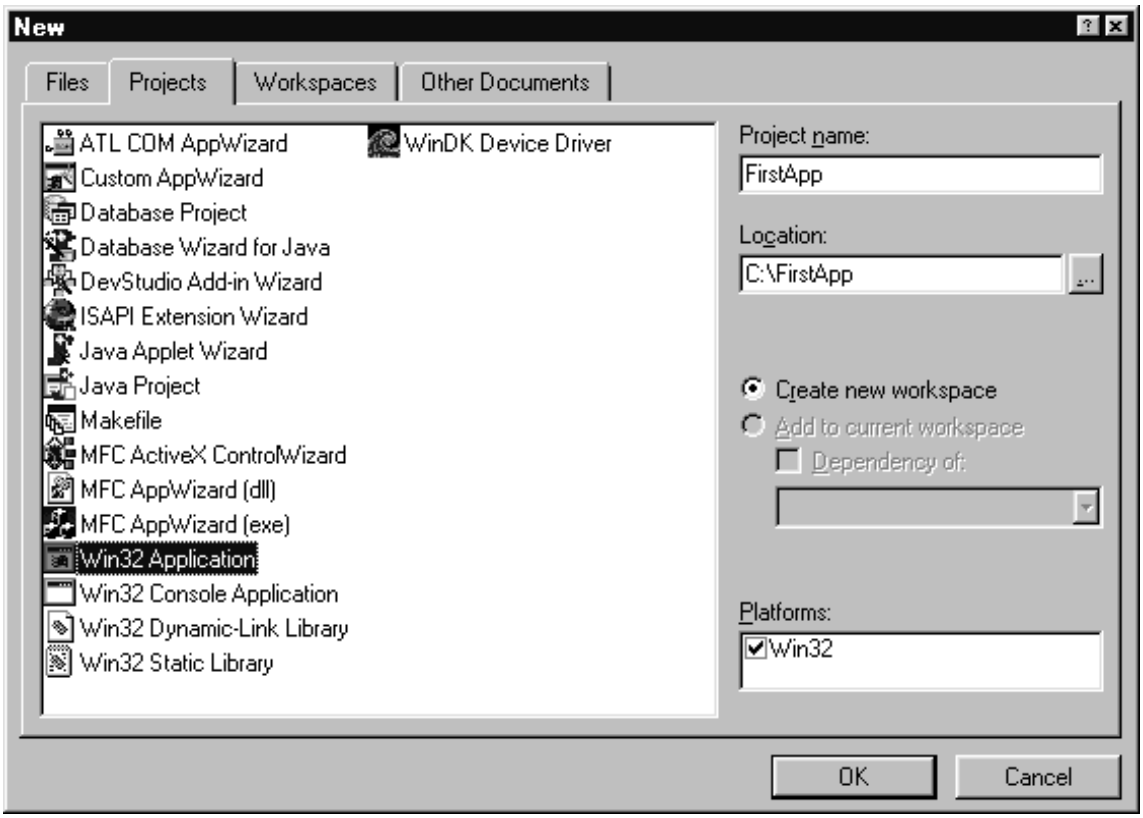
```
Create(NULL,"First MFC
Application",WS_OVERLAPPEDWINDOW,CRect(0,0,100,100));
```

creates a window of size 100x100 located in the desktop's upper-left corner.

3.1.3. Compiling an MFC Program

After creating your application and window classes you have to manually create a project for your files so that Developer Studio can compile and link the files into your final executable application. In this section, you create a new project for the FirstApp application that you just examined. To create a project for the application, perform the easy steps that follow.

- Select File, New from Developer Studio's menu bar. The New property sheet appears.
- If necessary, select the Projects tab. The Projects page appears.
- Select Win32 Application in the left hand pane, type **FirstApp** in the Project Name box, set the Location box to the folder in which you want to create the project, and click the OK button. Developer Studio creates the new project workspace.



- If the files, FIRSTAPP.H, FIRSTAPP.CPP, MAINFRM.H, and MAINFRM.CPP are already created in another directory copy the files into the project folder that you created in Step 3. Otherwise, create the application's source code files inside the project folder.
- Select Project, Add to Project, Files from Developer Studio's menu bar. The Insert Files into Project dialog box appears.
- Click on FIRSTAPP.CPP and MAINFRM.CPP to highlight them (hold down the Ctrl key), and then click the OK button. Developer Studio adds these two source code files to the project.
- Select the Project, Settings command from the menu bar. The Project Settings property sheet appears.
- Change the Microsoft Foundation Classes box to Use MFC In A Shared DLL, and then click the OK button. Developer Studio adds MFC support to the project. (You can also select Use MFC In A Static Library if you want the MFC code to be linked into your application's executable file. This makes your executable file larger but eliminates the need to distribute the MFC DLLs with your application. In most cases, the shared DLL is preferable.)

At this point, you're ready to compile and link your new project. To do this, just click the Build button on the toolbar or select Build, Build from the menu bar. When the project has been compiled and linked, select Build, Execute to run the application. When you do, you see the main window screen.

3.2. RESPONDING TO WINDOWS MESSAGES

To do something useful, your new MFC application must be able to respond to messages sent to its window. This is how a Windows application knows what the user and the system are doing. To respond to Windows messages, an MFC application must have a message map that tells MFC what functions handle what messages. In this section, you'll add message mapping to the MFC application.

3.2.1. Declaring a Message Map

Any class that has `CCmdTarget` as an ancestor can respond to Windows messages. These include application, frame-window, view-window, document, and control classes. Where you decide to intercept a message depends a great deal upon your program's design.

In your new MFC application, you'll add message mapping to the frame window, represented by the `CMainFrame` class. The first step in adding message mapping is to declare a message map in the class's header file. Placing the line

```
DECLARE_MESSAGE_MAP()
```

at the end of the class's declaration, just before the closing brace, takes care of this easy task. `DECLARE_MESSAGE_MAP()` is a macro defined by MFC. This macro takes care of all the details required to declare a message map for a class. You can look for `DECLARE_MESSAGE_MAP()`'s definition in MFC's source code if you want a better look, but you really don't need to know how it works.

3.2.2. Defining a Message Map

After you have declared your message map in your class's header, it's time to define the message map in your class's implementation file. Usually, programmers place the message map definition near the top of the file, right after the `include` lines. However, you can place it just about anywhere in the file as long as it's not nested inside a function or other structure.

What does a message map definition look like? The one that follows maps `WM_LBUTTONDOWN` messages to a message map function called `OnLButtonDown()`:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

The first line in the preceding code is a macro defined by MFC that takes care of initializing the message map definition. The macro's two arguments are the class for which you're defining the message map and that class's immediate base class. You need to tell MFC about the base class so that it can pass any unhandled messages on up the class hierarchy, where another message map might contain an entry for the message.

After the starting macro, you place the message map entries, which match messages with their message map functions. How you do this depends upon the type of message you're trying to map. For example, MFC has previously defined message map functions for all of the Windows system messages, such as `WM_LBUTTONDOWN`. To write a message map for such a message, you simply add an `ON_` prefix to the message name and tack on the parentheses. So `WM_LBUTTONDOWN` becomes `ON_WM_LBUTTONDOWN()`, `WM_MOUSEMOVE` becomes `ON_WM_MOUSEMOVE()`, `WM_KEYDOWN` becomes `ON_WM_KEYDOWN()`, and so on.

To determine the matching message map function for the map entry, you throw away the `_WM_` in the macro name and then spell the function name in uppercase and lowercase. For example, `ON_WM_LBUTTONDOWN()` becomes `OnLButtonDown()`; `ON_WM_MOUSEMOVE()` ends up as `OnMouseMove()`; and `ON_WM_KEYDOWN()` matches up with `OnKeyDown()`.

There are several other types of macros that you can use to define message map entries. For example, the `ON_COMMAND()` macro maps menu commands to their response functions, whereas the `ON_UPDATE_COMMAND_UI()` macro enables you to attach menu items to the functions that keep the menu items' appearance updated (checked, enabled, and so on).

3.2.3. Writing Message Map Functions

In the example for message map, there's only a single message map entry, which is `ON_WM_LBUTTONDOWN()`. This is the macro for the `WM_LBUTTONDOWN` Windows message. In order to complete the message mapping, you must now write the matching `OnLButtonDown()` function.

First, you need to declare the message map functions in your class's header file. The `OnLButtonDown()` prototype looks like this:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

The `afx_msg` part of the prototype does nothing more than mark the function as a message map function. You could leave the `afx_msg` off, and the function would still compile and execute just fine. However, Microsoft programming conventions suggest that you use `afx_msg` to distinguish message map functions from other types of member functions in your class. After declaring the message function prototype in the header file, add the function definition in the implementation file.

The following listings shows the necessary modifications done in the MAINFRM.H and MAINFRAME.CPP files for `WM_LBUTTONDOWN message mapping.`

```
//////////////////////////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which  //
//            represents the application's main window.    //
//////////////////////////////////////////////////////////
class CMainFrame : public CFrameWnd
{
public:
    CMainFrame();
    ~CMainFrame();
    // Message map functions.
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);

    DECLARE_MESSAGE_MAP()
};


//////////////////////////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame     //
//              class, which represents the application's  //
//              main window.                               //
//////////////////////////////////////////////////////////
#include <afxwin.h>
#include "mainfrm.h"
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
//////////////////////////////////////////////////////////
// CMainFrame: Construction and destruction.             //
//////////////////////////////////////////////////////////
CMainFrame::CMainFrame()
{
    Create(NULL, "MFC Application");
}
CMainFrame::~CMainFrame()
{
}
//////////////////////////////////////////////////////////
// Message map functions.                                //
//////////////////////////////////////////////////////////
void CMainFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
    MessageBox("You clicked the left mouse button!", "MFC Application");
}
```

When you compile and run this program, you see the application's main window. Click the left mouse button inside the window, a message box pops up to let you know that your message map is working.

4. UNDERSTANDING MENUS

Although creating and manipulating MFC menus takes quite a bit of program code, the process is fairly simple. You need only create the menu with the Visual C++ built-in menu editor, add message response functions to your window class, and add the appropriate entries to your message map. Then, after you load the menu in your program, the menu takes care of itself, and the appropriate message response functions are called whenever the user selects a menu item. The steps for adding menu support to your MFC application are as follows:

1.Create your menu resource using the Visual C++ menu editor. This process defines the menu titles that will appear in the menu bar as well as the menu commands that'll be in each pop-up menu.

2.Add menu state variables and message map functions to your window class's declaration. The message map functions can be conventional message response functions or special functions that control the appearance of menu commands.

3.Add the appropriate entries (using the predefined macros) to your window class's message map table. There are four commonly used macros: ON_COMMAND, ON_COMMAND_RANGE, ON_UPDATE_COMMAND_UI, and ON_UPDATE_COMMAND_UI_RANGE.

4.Write the message respond and update-command-UI functions that you have listed in your message map. The message response functions respond to the user's commands, whereas the update-command-UI functions determine whether commands are enabled, checked, bulleted, and so on.

In the rest of this chapter, you'll see, not only how to perform the preceding steps, but also how to assign command IDs to your menus and how to ensure that those menu IDs are accessible to the rest of your program. In the next section, you learn to use the Visual C++ menu editor to create your menu resource.

4.1. CREATING A MENU RESOURCE

As you now know, the first step toward adding a menu to your MFC application is creating the menu resource, which acts as a sort of template for Windows. When Windows sees the menu resource in your program, it uses the commands in the resource to construct the menu bar for you. To create a menu resource, just perform the following steps:
1.Select the Insert, Resource command from Developer Studio's menu bar. The Insert Resource dialog box appears, as shown in Fig 4.1.

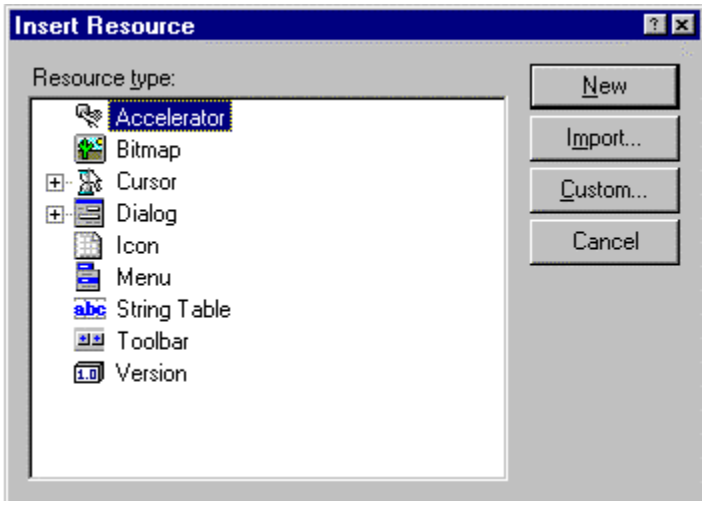


FIG. 4.1
The Insert Resource dialog box enables you to select the type of resource you want to add to your project.

2.Double-click Menu in the Resource Type box. The menu editor appears in one of Developer Studio's panes . You actually build your menu resource in the menu editor.

3.Double-click the blank menu to define your first menu. The Menu Item Properties property sheet appears (see Figure 4.2).

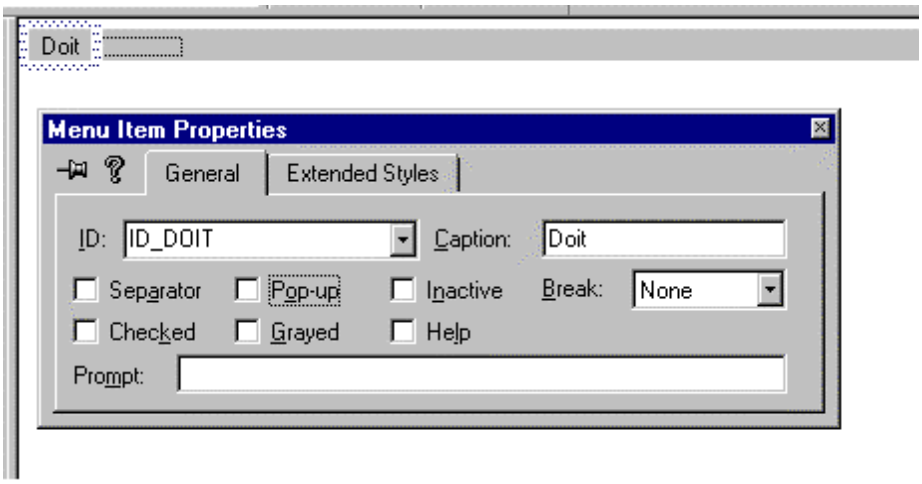


FIG. 4.2
You define each menu title and command using the Menu Item Properties property sheet.

4.Type the menu's name into the Caption box, and press Enter. The menu editor then adds the new menu title to the menu bar.

5.Double-click the blank menu item to start defining the first menu command in the new menu you created in Step 4. Type the command's ID into the ID box, type the command's caption into the Caption box (see Figure 4.2), and then press Enter.
Unlike menu titles (PopupMenu), which have only a caption, a menu command has both an ID and a caption. For a menu command the check box ‘Pop-up’ should be unchecked.

6.Repeat the appropriate steps above to define all your menus and menu commands.

When you create menu captions, you can specify a hotkey by placing an ampersand (&) immediately before the letter in the caption that the user can press to select that command. Windows automatically underlines the hotkey when it displays the menu caption.

4.1.1. Defining Menu IDs

So that Windows can tell the application which menu command the user has selected, you must define IDs for all commands in your menus. The ID is a numerical value represented by a constant. However, you don't have to worry about an ID's actual value because when you add a new command-ID constant to your menu resource, the menu editor automatically gives it the next consecutive ID value.

4.2. DEALING WITH RESOURCE FILES

After creating your menu resource (or any other type of resource for that matter), Developer Studio creates at least two files that you need to add to your application. The first file is called RESOURCE.H. This file contains all of the resource IDs (in this case, menu IDs) that you've defined. You must include RESOURCE.H in any file that refers to the constants you've defined. Otherwise, the compiler will have no idea what the constants represent and will complain with a string of error messages. Listing 4.1 shows the RESOURCE.H file that was created by Developer Studio for the Menu application you'll examine later in this chapter.

Listing 4.1 RESOURCE.H—The RESOURCE.H File Holds All the Constants You Defined in Your Resource File

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by rc00.rc
//
#define IDR_MENU1                   101
#define IDR_ACCELERATOR1            102
#define ID_MIN                      40001
#define ID_SEC                      40002
#define ID_DEGREE                   40003
#define ID_TIME_DIGITAL             40004
#define ID_TIME_ANALOG              40005

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE    103
#define _APS_NEXT_COMMAND_VALUE     40006
#define _APS_NEXT_CONTROL_VALUE     1000
#define _APS_NEXT_SYMED_VALUE       101
#endif
#endif
```

The first block of constants in Listing 4.1 are the constants that were defined when creating the application's menus. The second block of constants are used internally by Developer Studio to help manage the other symbols. For example, you can see that the last constant defined in the first block has a value of 40005, which means the next available value is 40006. The constant APS_NEXT_COMMAND_VALUE is not so coincidentally given the value 40006.

The second file that Developer Studio creates will have an .RC extension. This is the resource script that defines all your applications resources, including not just menus, but also dialog boxes, string tables, icons, cursors, and more. The resource script is a lot like a source code file, except it's written in a language that the resource compiler understands. The resource compiler takes the .RC file and compiles it into a .RES file, which is the binary representation of your application's resources. Listing 4.2 shows the rc00.RC file, which is the resource script for the menus you'll experiment with when you examine the Menu application later in this chapter, in the section titled "Exploring the Menu Application."

Listing 4.2 rc00.RC—The Resource Script of the Menu Application

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
```

```
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

#endif    // APSTUDIO_INVOKED
//
// Menu
//

IDR_MENU1 MENU DISCARDABLE
BEGIN
   MENUITEM "&MIN",                    ID_MIN
   MENUITEM "&SEC",                    ID_SEC
   POPUP "&Time"
   BEGIN
      MENUITEM "&Digital",               ID_TIME_DIGITAL
      MENUITEM "&Analog",                 ID_TIME_ANALOG
   END
END

//
// Accelerator
//

IDR_ACCELERATOR1 ACCELERATORS DISCARDABLE
BEGIN
   "M",          ID_MIN,              VIRTKEY, CONTROL, NOINVERT
   "S",          ID_SEC,              VIRTKEY, CONTROL, NOINVERT
   VK_F8,        ID_DEGREE,           VIRTKEY, NOINVERT
   "n",          ID_MIN,              ASCII,  NOINVERT
END

// String Table
STRINGTABLE DISCARDABLE
BEGIN
   ID_MIN               "Minute"
   ID_SEC               "Second"
END
```

```
#endif    // English (U.S.) resources

#ifndef APSTUDIO_INVOKED
//
// Generated from the TEXTINCLUDE 3 resource.
#endif    // not APSTUDIO_INVOKED
```

The specific area of interest in Listing 4.2 is the section of resource script following the Menu comment block. There you can see the actual commands that the resource compiler uses to create the binary version of the application's menus. Many programmers still write their resource scripts by hand, and most like to tinker with the script produced by programs, such as Developer Studio.

When you've had a chance to explore the Menu application that's presented later in this chapter (in the section titled "Exploring the Menu Application"), you might want to compare the resource script to the application's actual menus. You'll immediately see how the script's structure is converted into the menu hierarchy that you found in the application's menu bar.

4.3. INTRODUCING THE MENU APPLICATION

This is an application, to introduce how menus are created and how they respond to click events. On running the application you will see a window as shown below.

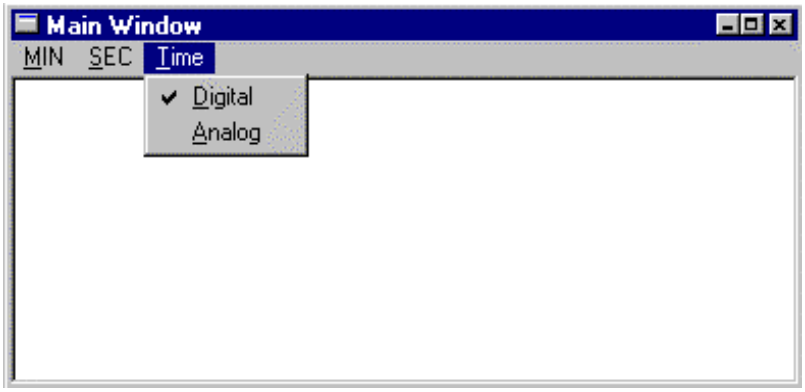


FIG. 4.3 The Main window of the Menu Application

First, turn your attention to the Time MenuItem. Click the menu's title, and you'll see that the menu has 2 commands named Digital and Analog. (see Figure 4.3). When the application begins, Digital is checked.
If you click on the MenuItem or command Analog, a message box appears confirming that the command was received by the application.
When you click the Time menu next time the check mark is next to Analog indicating that Analog was clicked previously. Only one of the 2 options (Analog or Digital) can be selected at a time. Whichever option you click inherits the CheckMark.

4.4. EXPLORING THE MENU APPLICATION

The source code is given below.
Listing 4.3 St00FrmWnd.H—The Header File of the Frame Window Class

```
////////////////////////////////////////////////////////
// St00FrmWnd.H: Header file for the CSt00FrmWnd class, which
//          represents the application's main window.
////////////////////////////////////////////////////////

#include <afxwin.h>
```

```
class CSt00FrmWnd : public CFrameWnd
{
public:
	CSt00FrmWnd();

protected:
	void OnUpdateDigital(CCmdUI* pCmdUI);
	void OnUpdateAnalog(CCmdUI* pCmdUI);
	void OnMin();
	void OnSec();
	void OnDigital();
	void OnAnalog();
	DECLARE_MESSAGE_MAP()
private:
	BOOL m_bDigital;
};
```

Listing 4.4 St00FrmWnd.CPP—The Implementation File of the Frame Window Class

```
//////////////////////////////////////////////////////////
// St00FrmWnd.CPP: Implementation file for the CMainFrame
//             class, which represents the application's
//             main window.
//////////////////////////////////////////////////////////
#include "St00FrmWnd.h"
#include "resource.h"


BEGIN_MESSAGE_MAP(CSt00FrmWnd,CFrameWnd)
	//{{AFX_MSG_MAP(CSt00FrmWnd)
	//}}AFX_MSG_MAP
	ON_COMMAND(ID_MIN,OnMin)
	ON_COMMAND(ID_SEC,OnSec)
	ON_COMMAND(ID_TIME_DIGITAL,OnDigital)
	ON_COMMAND(ID_TIME_ANALOG,OnAnalog)
	ON_UPDATE_COMMAND_UI(ID_TIME_DIGITAL,OnUpdateDigital)
	ON_UPDATE_COMMAND_UI(ID_TIME_ANALOG,OnUpdateAnalog)
END_MESSAGE_MAP()


CSt00FrmWnd::CSt00FrmWnd()
{
	Create(NULL,"Main Window",WS_OVERLAPPEDWINDOW,
		rectDefault,NULL,MAKEINTRESOURCE(IDR_MENU1));
	m_bDigital=TRUE;
	LoadAccelTable(MAKEINTRESOURCE(IDR_ACCELERATOR1));
};

void CSt00FrmWnd::OnMin()
{
	MessageBox("Min","Title",MB_OK);
};

void CSt00FrmWnd::OnSec()
{
		MessageBox("Sec","Title",MB_OK);
};
```

```
void CSt00FrmWnd::OnDigital()
{
	MessageBox("Digital","Title",MB_OK);
	m_bDigital=TRUE;
};

void CSt00FrmWnd::OnAnalog()
{
		MessageBox("Analog","Title",MB_OK);
		m_bDigital=FALSE;
};

void CSt00FrmWnd::OnUpdateDigital(CCmdUI* pCmdUI)
{
	if (m_bDigital)
		pCmdUI->SetCheck(1);
	else
		pCmdUI->SetCheck(0);
}
void CSt00FrmWnd::OnUpdateAnalog(CCmdUI* pCmdUI)
{
	if (m_bDigital)
		pCmdUI->SetCheck(0);
	else
		pCmdUI->SetCheck(1);

}
```

4.4.1. Declaring Menu State Variables

If you have very simple menus in your applications—menus that require no special handling, such as check marking or disabling—you probably don't need menu state variables in your program. However, most full-fledged Windows applications have at least a few menu commands that require special handling. The most common example is a menu command that must be enabled or disabled depending upon the application's current status. For example, the Edit, Paste command should not be enabled unless there's something in the Clipboard to paste. Another example might be a menu of options that can be turned on or off. If an option is on, it should be check marked or bulleted.

If you look at the declaration of the CSt00FrmWnd class, you'll see the data member declaration shown below.

```
BOOL m_bDigital;
```

Here, m_bDigital tracks the state of Time. That is, if it is Digital or Analog. If m_bDigital =TRUE, then the selected item is Digital. If m_bDigital = FASLE , selected item is Analog.

4.4.2. Declaring Menu Message Handlers

As you know, MFC uses message maps to associate message response functions with messages Windows sends to the application. Some of the messages are system messages, such as WM_PAINT, which are sent automatically by Windows. Other messages that an application must respond to are the messages triggered when the user selects menu commands. You'll soon see how to add these messages to your message map. However, you also need to declare the message response functions with which these messages will be associated. Listing 4.5 shows how the Menu application's CSt00FrmWnd class declares its menu message response member functions.

Listing 4.5 Declaring Menu Message Handlers

```
	void OnMin();
	void OnSec();
	void OnDigital();
```

```
void OnAnalog();
```

In most MFC applications, you'll need to create update-command-UI functions, which are responsible for the appearance of menu commands. For example, update-command-UI functions can add check marks or bullets to menu items. They can also enable or disable menu items. Just as with the regular menu message handlers, the update-command-UI handlers must be declared in your class. Listing4.6 shows how they're declared in the Menu application's CMainFrame class.

Listing4.6 Declaring Update-Command-UI Member Functions

```
void OnUpdateDigital(CCmdUI* pCmdUI);
void OnUpdateAnalog(CCmdUI* pCmdUI);
```

All of the update-command-UI functions have a single parameter, which is a pointer to a CCmdUI object. You'll soon learn how the update-command-UI functions control the appearance of menu items.

4.4.3. Defining the Message Map

Now that you have your menu message map functions declared, it's time to define the message map itself. To define the various types of message map functions used with menus, you use four new message map macros: ON_COMMAND, ON_COMMAND_RANGE, ON_UPDATE_COMMAND_UI, and ON_UPDATE_COMMAND_UI_RANGE. Listing 4.7shows the message map entries for the Menu application's message handlers.

Listing 4.7 Message Handler Message Map Entries

```
ON_COMMAND(ID_MIN,OnMin)
ON_COMMAND(ID_SEC,OnSec)
ON_COMMAND(ID_TIME_DIGITAL,OnDigital)
ON_COMMAND(ID_TIME_ANALOG,OnAnalog)
```

Use the ON_COMMAND macro to map a single menu command to a message response function. The macro's first argument is the message ID, and the second argument is the name of the function that will handle that message. For example, in the Menu application, Digital command has a command ID of ID_TIME_DIGITAL. This message ID, which is sent to the application whenever the user selects the Time,Digital command, is mapped to the message response function OnDigital().

If you want to map a range of menu commands to a single message response function, use the ON_COMMAND_RANGE macro, like this:

```
ON_COMMAND_RANGE(ID_TIME_DIGITAL,
   ID_TIME_ANALOG, OnDigital)
```

The ON_COMMAND_RANGE macro's three arguments are the range's starting ID, the range's ending ID, and the name of the message response function. In order for this type of message mapping to work, the command IDs in the range must be consecutive.

Once you have your message-response functions added to your message map, it's time to think about any update-command-UI functions you might need. If you have any menu items that are subject to enabling/disabling, check marking, or other similar changes, you'll need to add ON_UPDATE_COMMAND_UI macros to your message map. In the Menu application, the update-command-UI entries look like Listing 4.8.

Listing 4.8 Defining Update-Command-UI Message Map Entries

```
ON_UPDATE_COMMAND_UI(ID_TIME_DIGITAL,OnUpdateDigital)
ON_UPDATE_COMMAND_UI(ID_TIME_ANALOG,OnUpdateAnalog)
```

As you can see, update-command-UI message map entries are similar to the regular ON_COMMAND entries, requiring two arguments: the ID for which you're defining the entry and the name of the function to which the ID

should be mapped. However, although the definition of the table entries is similar, the way you write the actual response update-command-UI functions is very different, as you'll soon see.

4.4.4. Writing Menu Message Handlers

When it comes to responding to menu commands, the process is fairly straightforward once you've created the message map. You just write a function that performs whatever tasks the menu command is supposed to handle.

Listing 4.9 Responding to a Menu Command

```
void CSt00FrmWnd::OnDigital()
{
        MessageBox("Digital","Title",MB_OK);
        m_bDigital=TRUE;
};
```

The OnDigital() message response function simply displays a message box and sets the value of m_bDigital flag to TRUE. The m_bDigital flag is used in the OnUpdateDigital(), OnUpdateAnalog functions to determine the visual state of the associated menu items.

4.4.5. Writing Update-Command-UI Functions

The last topic you need to cover in this chapter is the update-command-UI functions, which control how your menu commands look to the user when a pop-up menu is displayed. As with the more conventional message response functions, how you write your update-command-UI functions depends upon each menu's purpose. In the case of the Menu application's Time menu, the function only needs to check or uncheck the menu commands, Analog and Digital, depending upon the value of the flag associated with the command. For example, look at the OnUpdateDigital() function shown in Listing 4.10.

Listing 4.10 Updating the Appearance of Menu Commands

```
void CSt00FrmWnd::OnUpdateDigital(CCmdUI* pCmdUI)
{
        if (m_bDigital)
                pCmdUI->SetCheck(1);
        else
                pCmdUI->SetCheck(0);
}
```

Here you can see that the function receives a single parameter, which is a pointer to a CCmdUI object. You use the CCmdUI object's member functions to manipulate the menu command. These member functions are Enable() (which enables or disables a menu item), SetCheck() (which adds or removes a check mark), SetRadio() (which adds or removes bullets), and SetText() (which changes a menu command's caption). All of these functions, except SetText(), require a single parameter. A non-zero value turns the attribute on, and a 0 turns the attribute off. For example, calling SetCheck(TRUE) turns a check mark on, whereas calling SetCheck(FALSE) turns it off. The argument for the SetText() function is the new text string for the menu item.

In Listing 4.10, the m_bDigital flag is either TRUE or FALSE, so it can be used to directly control whether the menu command displays a check mark.

4.4.6. Displaying the menu on the screen

The menu should be displayed on the screen when the window is created itself. The Create memeber function of CFrameWnd which is called in the Frame Windows constructor takes several arguments and they are explained in the Chapter “Constructing an MFC program”. The 6th parameter to the Create function takes the Menu Name. Since Our menu is identified by an int ID (IDR_MENU1), it has to be converted to a resource string using the MAKEINTRESOURCE macro.

```
CSt00FrmWnd::CSt00FrmWnd()
{
        Create(NULL,"Main Window",WS_OVERLAPPEDWINDOW,
                rectDefault,NULL,MAKEINTRESOURCE(IDR_MENU1));
```

```
        m_bDigital=TRUE;
        LoadAccelTable(MAKEINTRESOURCE(IDR_ACCELERATOR1));
};
```

If the menu name is not given in the Create function, a window with no menu is created.

MFC gives you all of the tools you need to create professional-looking menus for your applications. By taking advantage of message maps, you can associate a specific function with a menu message as well as keep your menu commands visually updated. Menus are an important element of most Windows applications—an element that takes some work to implement properly. However, menu handling in an MFC program is much simpler than writing your menu-handling code from scratch.

5. GRAPHICS AND TEXT DRAWING

5.1. UNDERSTANDING DEVICE CONTEXTS

As you know, every Windows application (in fact, every computer application) must manipulate data in some way. Most applications must also display data. Unfortunately though, because of Windows' device independence, this task is not as straightforward in Windows as it is in DOS.

Although device independence forces you, the programmer, to deal with data displays indirectly, it helps you by ensuring that your programs run on all popular devices. In most cases, Windows handles devices for you through the device drivers that the user has installed on the system. These device drivers intercept the data the application needs to display and translates the data appropriately for the device on which it will appear, whether that is a screen, a printer, or some other output device.

To understand how all of this device independence works, imagine an art teacher trying to design a course of study appropriate for all types of artists. The teacher creates a course outline that stipulates the subject of a project, the suggested colors to be used, the dimensions of the finished project, and so on. What the teacher doesn't stipulate is the surface on which the project will be painted or the materials needed to paint on that surface. In other words, the teacher stipulates only general characteristics. The details of how these characteristics are applied to the finished project are left up to each artist.

The instructor in the preceding scenario is much like a Windows programmer. The programmer has no idea who might eventually use the program and what kind of system that user might have. The programmer can recommend the colors in which data should be displayed and the coordinates at which the data should appear, for example, but it is the device driver—the Windows artist—that ultimately decides how the data appears.

A system with a VGA monitor might display data with fewer colors than a system with a Super VGA monitor. Likewise, a system with a monochrome monitor displays the data in only a single color. Monitors with high resolutions can display more data than lower-resolution monitors. The device drivers, much like the artists in the imaginary art school, must take the display requirements and fine-tune them to the device on which the data will actually appear. And it is a data structure called a device context that links the application to the device's driver.

A device context (DC) is little more than a data structure that keeps track of the attributes of a window's drawing surface. These attributes include the currently selected pen, brush, and font that will be used to draw on the screen. Unlike an artist, who can have many brushes and pens with which to work, a DC can use only a single pen, brush, or font at a time. If you want to use a pen that draws wider lines, for example, you need to create the new pen and then replace the DC's old pen with the new one. Similarly, if you want to fill shapes with a red brush, you must create the brush and "select it into the DC," which is how Windows programmers describe replacing a tool in a DC.

A window's client area is a versatile surface that can display anything a Windows program can draw. The client area can display any type of data because everything displayed in a window, whether it be text, spreadsheet data, a bitmap, or any other type of data, is displayed graphically. MFC helps you display data by encapsulating Windows' GDI functions and objects into its DC classes.

5.2. INTRODUCING THE PAINT1 APPLICATION

The sample program in this chapter shows you how to use MFC to display many types of data in an application's window. When you run the program, the main window appears, showing a Rectangle, an Ellipse a Line and Text drawn drawn on screen (see Figure 5.1).

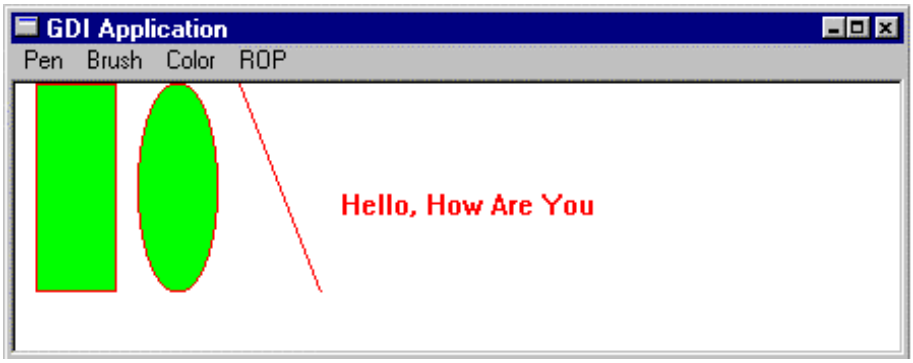


FIG. 5.1 The main Window

You can change the style and color of the pens and brushes through the Menu. When the Brush color changes, the fill color of the Ellipse and rectangle also changes. The Pen width also can be changed through the Menu by clicking the menu, Pen , Width and then the menu command 1,3 or 5 (see Figure 5.2). The style of the pen can be changed to Dot, Dash or Solid.

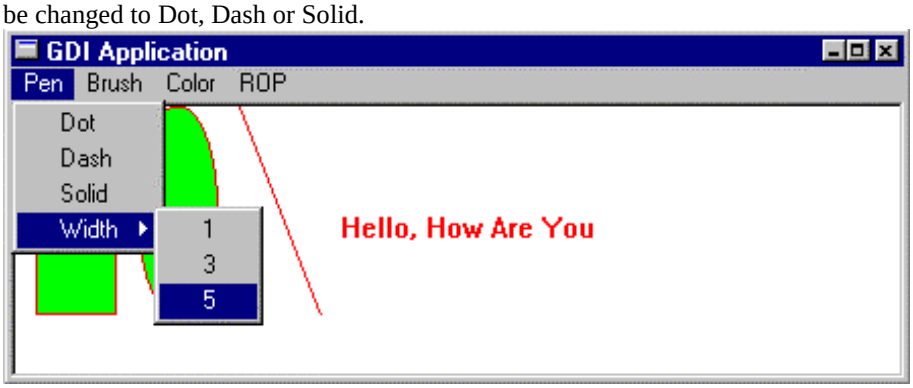


FIG. 5.2

The screen in fig 5.3 is obtained by selecting Dash Pen Style.

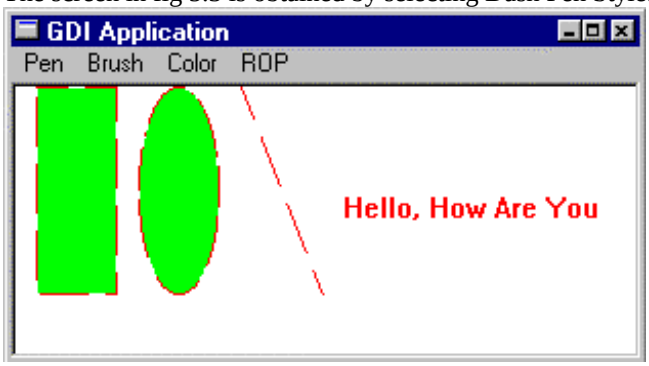


FIG. 5.3 Pen Style - Dash

The brush Style can be changed by selecting the MenuItem Brush and then the command Solid, Diagonal, Cross or Horizontal. The Popup Menu is shown in FIG 5.4. By changing the Brush style, the Fill pattern inside the Ellipse and Rectangle can be changed.

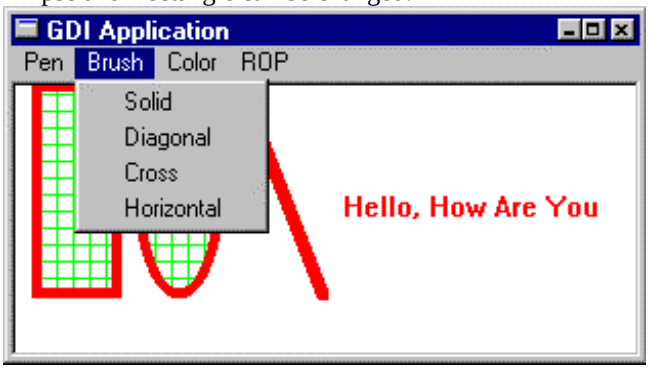


FIG 5.4
Fig. 5.5 shows the screen when pen width is selected as 5 and Brush Style is selected as Cross.

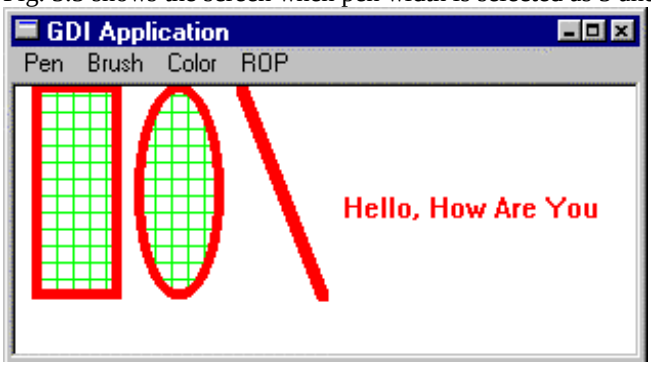


FIG 5.5 Pen Width - 5 and Brush Style-Cross.
This screen is produced by creating new pens and Brushes and drawing the shapes with these objects. Different brushes enable your window to display colors and patterns. The color of the Brush and pen can be changed by selecting the Menuitem Color and then selecting Red, Green or Blue.

Listing 5.1 through Listing 5.4 are the source codes for the Paint1 application. Take some time now to run the application, as well as to look over the source code in the listings. In the sections that follow, you'll learn how the program produces its displays and how MFC simplifies the drawing of a window's displays.

Listing 5.1 MyApp.H—The Header File of the Application Class

```
////////////////////////////////////////////////////////
// MyApp.h: interface for the MyApp class.
//
////////////////////////////////////////////////////////////////////

#if !defined(AFX_MYAPP_H__AFD0BB02_0C43_11D2_9210_00A0C95C441A__INCLUDED_)
#define AFX_MYAPP_H__AFD0BB02_0C43_11D2_9210_00A0C95C441A__INCLUDED_

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

#include <afxwin.h>

class MyApp : public CWinApp
{
public:
        BOOL InitInstance();
        MyApp();
        virtual ~MyApp();

};

#endif // !defined(AFX_MYAPP_H__AFD0BB02_0C43_11D2_9210_00A0C95C441A__INCLUDED_)

MyApp theApp;
```

Listing 5.2 MyApp.CPP—The Implementation File of the Application Class

```
////////////////////////////////////////////////////////
// MyApp.cpp: implementation of the MyApp class.
//
////////////////////////////////////////////////////////////////////
#include "MyApp.h"
#include "MainFrm.h"
```

```
/////////////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////////////

MyApp::MyApp()
{
}

MyApp::~MyApp()
{
}

BOOL MyApp::InitInstance()
{
        m_pMainWnd = new CMainFrm;
        m_pMainWnd->ShowWindow(m_nCmdShow);
        return TRUE;
}
```

Listing 5.3 MAINFRM.H—The Header File of the Frame Window

```
/////////////////////////////////////////////////////////////
// CMainFrm frame

class CMainFrm : public CFrameWnd
{
        DECLARE_DYNCREATE(CMainFrm)



// Attributes
public:
        CMainFrm();
// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMainFrm)
        //}}AFX_VIRTUAL

// Implementation
protected:
        virtual ~CMainFrm();

        // Generated message map functions
        //{{AFX_MSG(CMainFrm)
        afx_msg void OnBrushCross();
        afx_msg void OnBrushDiagonal();
        afx_msg void OnBrushSolid();
        afx_msg void OnBrushHorizontal();
        afx_msg void OnColorBlue();
        afx_msg void OnColorGreen();
        afx_msg void OnColorRed();
        afx_msg void OnPenDash();
        afx_msg void OnPenDot();
        afx_msg void OnPenSolid();
        afx_msg void OnRopCopy();
```

```
	afx_msg void OnRopXor();
	afx_msg void OnPaint();
	afx_msg void OnPenWidth1();
	afx_msg void OnPenWidth3();
	afx_msg void OnPenWidth5();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
private:
	int m_nROP;
	LOGBRUSH m_lbBrush;
	LOGPEN m_lpPen;

};
```

Listing 5.4 MAINFRM.CPP—The Implementation File of the Frame Window Class

```
/////////////////////////////////////////////////////////////
// MainFrm.cpp : implementation file
//

#include <afxwin.h>
#include "resource.h"
#include "MainFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMainFrm

IMPLEMENT_DYNCREATE(CMainFrm, CFrameWnd)

CMainFrm::CMainFrm()
{
	Create(NULL,"GDI Application",WS_OVERLAPPEDWINDOW,
		rectDefault,NULL,MAKEINTRESOURCE(IDR_MENU2));

	m_nROP=R2_COPYPEN;
	m_lpPen.lopnColor = RGB(255,0,0);
	m_lpPen.lopnStyle = PS_SOLID;
	CPoint point(1,2);
	m_lpPen.lopnWidth=point;
	m_lbBrush.lbColor = RGB(0,255,0);

}

CMainFrm::~CMainFrm()
{
}


BEGIN_MESSAGE_MAP(CMainFrm, CFrameWnd)
	//{{AFX_MSG_MAP(CMainFrm)
	ON_COMMAND(ID_BRUSH_CROSS, OnBrushCross)
	ON_COMMAND(ID_BRUSH_DIAGONAL, OnBrushDiagonal)
	ON_COMMAND(ID_BRUSH_SOLID, OnBrushSolid)
```

```
	ON_COMMAND(ID_BRUSH_HORIZONTAL, OnBrushHorizontal)
	ON_COMMAND(ID_COLOR_BLUE, OnColorBlue)
	ON_COMMAND(ID_COLOR_GREEN, OnColorGreen)
	ON_COMMAND(ID_COLOR_RED, OnColorRed)
	ON_COMMAND(ID_PEN_DASH, OnPenDash)
	ON_COMMAND(ID_PEN_DOT, OnPenDot)
	ON_COMMAND(ID_PEN_SOLID, OnPenSolid)
	ON_COMMAND(ID_ROP_COPY, OnRopCopy)
	ON_COMMAND(ID_ROP_XOR, OnRopXor)
	ON_WM_PAINT()
	ON_COMMAND(ID_PEN_WIDTH_1, OnPenWidth1)
	ON_COMMAND(ID_PEN_WIDTH_3, OnPenWidth3)
	ON_COMMAND(ID_PEN_WIDTH_5, OnPenWidth5)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMainFrm message handlers

void CMainFrm::OnBrushCross()
{
	// TODO: Add your command handler code here
		m_lbBrush.lbStyle = BS_HATCHED;
		m_lbBrush.lbHatch = HS_CROSS;
		Invalidate();
}

void CMainFrm::OnBrushDiagonal()
{
	// TODO: Add your command handler code here
	m_lbBrush.lbStyle = BS_HATCHED;
	m_lbBrush.lbHatch = HS_FDIAGONAL;
	Invalidate();
}

void CMainFrm::OnBrushSolid()
{
	// TODO: Add your command handler code here
		m_lbBrush.lbStyle = BS_SOLID;
		Invalidate();
}

void CMainFrm::OnBrushHorizontal()
{
	// TODO: Add your command handler code here
	m_lbBrush.lbStyle = BS_HATCHED;
	m_lbBrush.lbHatch = HS_HORIZONTAL;
	Invalidate();
}

void CMainFrm::OnColorBlue()
{
	// TODO: Add your command handler code here
	m_lpPen.lopnColor = RGB(0,0,255);
	m_lbBrush.lbColor =RGB(0,0,255);
	Invalidate();
}

void CMainFrm::OnColorGreen()
```

```
{
	// TODO: Add your command handler code here
	m_lpPen.lopnColor = RGB(0,255,0);
	m_lbBrush.lbColor =RGB(0,255,0);
	Invalidate();
}

void CMainFrm::OnColorRed()
{
	// TODO: Add your command handler code here
	m_lpPen.lopnColor = RGB(255,0,0);
	m_lbBrush.lbColor =RGB(255,0,0);
	Invalidate();
}

void CMainFrm::OnPenDash()
{
	// TODO: Add your command handler code here
	m_lpPen.lopnStyle = PS_DASH;
	Invalidate();
}

void CMainFrm::OnPenDot()
{
	// TODO: Add your command handler code here
	m_lpPen.lopnStyle = PS_DOT;
	Invalidate();
}

void CMainFrm::OnPenSolid()
{
	// TODO: Add your command handler code here
	m_lpPen.lopnStyle = PS_SOLID;
	Invalidate();
}

void CMainFrm::OnRopCopy()
{
	// TODO: Add your command handler code here
	m_nROP=R2_COPYPEN;
	Invalidate();
}



void CMainFrm::OnRopXor()
{
	// TODO: Add your command handler code here
	m_nROP=R2_XORPEN;
	Invalidate();
}

void CMainFrm::OnPaint()
{
	CPaintDC dc(this); // device context for painting

	dc.SetROP2(m_nROP);
	CPen pen;
	CBrush brush;
```

```
	pen.CreatePenIndirect(&m_lpPen);
	brush.CreateBrushIndirect(&m_lbBrush);
	CPen* oldPen = dc.SelectObject(&pen);
	CBrush* oldBrush = dc.SelectObject(&brush);

	dc.Rectangle(10,0,50,100);// draws a rectangle

	dc.Ellipse(60,0,100,100);//Draws an Ellipse

	//Draws a line
	dc.MoveTo(110,0);
	dc.LineTo(150,100);

	dc.SetTextColor(m_lpPen.lopnColor);//Changes Color of text

	dc.TextOut(160,50,"Hello, How Are You"); //Displays Text

	dc.SelectObject(oldPen);
	dc.SelectObject(oldBrush);
	// TODO: Add your message handler code here

	// Do not call CFrameWnd::OnPaint() for painting messages
}

void CMainFrm::OnPenWidth1()
{
	// TODO: Add your command handler code here
		CPoint point(1,2);
		m_lpPen.lopnWidth=point;
		Invalidate();
}

void CMainFrm::OnPenWidth3()
{
	// TODO: Add your command handler code here
		CPoint point(3,2);
		m_lpPen.lopnWidth=point;
		Invalidate();
}

void CMainFrm::OnPenWidth5()
{
	// TODO: Add your command handler code here
		CPoint point(5,2);
		m_lpPen.lopnWidth=point;
		Invalidate();
}
```

5.3. EXPLORING THE APPLICATION

5.3.1. Painting in an MFC Program

You have already learnt about message maps and how you can tell MFC which functions to call when it receives messages from Windows. One important message that every Windows program with a window must handle is WM_PAINT. Windows sends the WM_PAINT message to an application's window when the window needs to be redrawn. There are several events that cause Windows to send a WM_PAINT message. The first event is simply the running of the program by the user. In a properly written Windows application, the application's

window gets a WM_PAINT message almost immediately after being run to ensure that the appropriate data is displayed from the very start.

Another time that a window might receive the WM_PAINT message is when the window has been resized or has recently been uncovered—either fully or partially—by another window. In either case, part of the window that wasn't visible before is now on the screen and must be updated.

Finally, a program can indirectly send itself a WM_PAINT message by invalidating its client area. Having this capability ensures that an application can change its window's contents almost any time it wants. For example, a word processor might invalidate its window after the user pastes some text from the Clipboard.

The message map macro for a WM_PAINT message is ON_WM_PAINT(). The matching message map function, OnPaint() should be called. This is another case when MFC has already done most of the work of matching a Windows message with its message response function.

So, in order to paint your window's display, you need to add an ON_WM_PAINT() entry to your message map and then write an OnPaint() function. In the OnPaint() function, you write the code that will produce your window's display. Then, whenever Windows sends your application the WM_PAINT message, MFC will automatically call OnPaint(), which will draw the window's display just as you want it.

If you look near the top of Listing 5.4, which is the implementation file of the frame window class, you'll see the application's main message map, as shown in Listing 5.5.

Listing 5.5 The Message Map of the Paint1 Application

```
BEGIN_MESSAGE_MAP(CMainFrm, CFrameWnd)
        //{{AFX_MSG_MAP(CMainFrm)
        ON_COMMAND(ID_BRUSH_CROSS, OnBrushCross)
        ON_COMMAND(ID_BRUSH_DIAGONAL, OnBrushDiagonal)
        ON_COMMAND(ID_BRUSH_SOLID, OnBrushSolid)
        ON_COMMAND(ID_BRUSH_HORIZONTAL, OnBrushHorizontal)
        ON_COMMAND(ID_COLOR_BLUE, OnColorBlue)
        ON_COMMAND(ID_COLOR_GREEN, OnColorGreen)
        ON_COMMAND(ID_COLOR_RED, OnColorRed)
        ON_COMMAND(ID_PEN_DASH, OnPenDash)
        ON_COMMAND(ID_PEN_DOT, OnPenDot)
        ON_COMMAND(ID_PEN_SOLID, OnPenSolid)
        ON_COMMAND(ID_ROP_COPY, OnRopCopy)
        ON_COMMAND(ID_ROP_XOR, OnRopXor)
        ON_WM_PAINT()
        ON_COMMAND(ID_PEN_WIDTH_1, OnPenWidth1)
        ON_COMMAND(ID_PEN_WIDTH_3, OnPenWidth3)
        ON_COMMAND(ID_PEN_WIDTH_5, OnPenWidth5)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

As you can tell by the message map's entries, the application can respond to WM_PAINT and WM_COMMAND messages. The ON_WM_PAINT() entry maps to the OnPaint() message map function. In the first line of that function, the program creates a DC for the client area of the frame window:

```
CPaintDC dc(this);
```

CPaintDC is a special class for managing paint DCs, which are device contexts that are used only when responding to WM_PAINT messages. In fact, if you're going to use MFC to create your OnPaint() function's paint DC, you must use the CPaintDC class. This is because an object of the CPaintDC class does more than just create a DC; it also calls the BeginPaint() Windows API function in the class's constructor and calls EndPaint() in its destructor. When responding to WM_PAINT messages, calls to BeginPaint() and EndPaint() are required. The CPaintDC class handles this requirement without your having to get involved in all the messy details.

As you can see, the CPaintDC constructor takes a single argument, which is a pointer to the window for which you're creating the DC.
After creating the paint DC, the OnPaint() function in Listing 5.4 uses its m_lbBrush, m_lpPen and m_nROP data to create a brush, pen and to change the ***Raster Operation (ROP)*** modes respectively. After the brush and Pen is created, they are selected into the Device context. A rectangle, Ellipse, Line and then Text (Hello, How Are You) is drawn on the screen.

5.3.2. Creating a Pen.

Pen Drawing objects are used for drawing the outlines of the shapes drawn by functions Rectangle(),Ellipse(), Line To(), Polygon(),RoundRect() etc.
To create a custom pen, you need only supply the pen's line style, thickness in pixels, and color.
The parameter Pen Style can be one of the styles listed in Table 5.2. Note that only solid lines can be drawn with different thicknesses. Patterned lines always have a thickness of 1. The second argument above is the line thickness.

Finally, the third argument is the line's color. The RGB macro takes three values for the red, green, and blue color components and converts them into a valid Windows color reference. The values for the red, green, and blue color components can be anything from zero to 255—the higher the value, the brighter the color component. The preceding line creates a bright blue pen. If all the color values were zero, the pen would be black, whereas if the color values were all 255, the pen would be white.

Table 5.1 Pen Styles

Style	**Meaning**
PS_DASH	Specifies a pen that draws dashed lines
PS_DASHDOT	Specifies a pen that draws dash dot patterned lines
PS_DASHDOTDOT	Specifies a pen that draws dash-dot-dot patterned lines
PS_DOT	Specifies a pen that draws dotted lines
PS_INSIDEFRAME	Specifies a pen that's used with shapes, where the line's thickness must not extend outside of the shape's frame.
PS_NULL	Specifies a pen that draws invisible lines
PS_SOLID	Specifies a pen that draws solid lines

You have a choice between two techniques for creating graphic objects, such as pens and brushes:

One-stage construction: Construct and initialize the object in one stage, all with the constructor.

```
CPen myPen1( PS_DOT, 5, RGB(0,0,0) );
```

Two-stage construction: Construct and initialize the object in two separate stages. The constructor creates the object and the CreatePen() member function of CPen initializes it.

```
// Two-stage: first construct the pen
CPen myPen2;
// Then initialize it
myPen2.CreatePen( PS_DOT, 5, RGB(0,0,0) ) ;
```

A Logpen can also be constructed using a structure, LOGPEN, which is a structure that defines the style, width, and color of a Pen. The LOGPEN structure has the following form:

```
typedef struct tagLOGPEN {  /* lgpn */
   UINT     lopnStyle;
   POINT    lopnWidth;
   COLORREF lopnColor;
} LOGPEN;
```

Initialise the members of the LOGPEN structure and then use CreatePenIndirect() member function of Cpen. This method is used in listing 5.4 . The variable m_lpPen of type LOGPEN is declared in CMainFrm class.. Its members are initialized through the Menu. The member functions of CMainFrm class responsible for setting the LOGPEN structure are given below.
To change Color

```
            OnColorBlue();
        OnColorGreen();
        OnColorRed();
To change Pen Style
        OnPenDash();
        OnPenDot();
        OnPenSolid();
To Change Width
        OnPenWidth1();
        OnPenWidth3();
        OnPenWidth5();
```

After creating the new pen, the program selects it into the DC, saving the pointer to the old pen, like this:

```
CPen* oldPen = dc.SelectObject(&pen);
```

After the pen is selected into the DC, the program can draw an Ellipse, Rectangle or line with the pen. See the listing from OnPaint() funtion below.

```
dc.Rectangle(10,0,50,100);// draws a rectangle

dc.Ellipse(60,0,100,100);//Draws an Ellipse
```

The Rectangle and Ellipse funtion takes a pointer to a CRect object or the Left, Top, Right and Bottom coordinates respectively. The next call is to the paint DC's MoveTo() and LineTo() member functions, like this:

```
//Draws a line
        dc.MoveTo(110,0);
        dc.LineTo(150,100);
```

The MoveTo() function positions the starting point of the line, whereas the LineTo() function draws a line—using the pen currently selected into the DC—from the point set with MoveTo() to the coordinates given as the function's two arguments.

Finally, the last step is to restore the DC by reselecting the old pen into the DC:

```
dc.SelectObject(oldPen);
```

5.3.3. Using Brushes

Creating and using brushes in an MFC program is not unlike using pens. In fact, just as with pens, you can create solid, hatched and patterned brushes. You can even create brushes from bitmaps that contain your own custom fill patterns.
Brush drawing objects specify the way that the filled areas of filled figures are drawn when funcions like Rectangle(), Ellipse(), Polygon(), RoundRect(),FillRect, FillRegn etc are called.
Windows defines several constants for the brush patterns (or hatch styles, as they're often called). Those constants are HS_BDIAGONAL, HS_CROSS, HS_DIAGCROSS, HS_FDIAGONAL, HS_HORIZONTAL, and HS_VERTICAL.

As with pens, brushes can also be created in 2 different ways.

One-stage construction: Construct and initialize the object in one stage, all with the constructor.

```
Creating a Solid Brush
CBrush RBrush( RGB(255,0,0) );

Creating a hatched brush
CBrush RBrush( HS_CROSS,RGB(255,0,0) );
```

Two-stage construction: Construct and initialize the object in two separate stages. The constructor creates the object and the CreateSolidBrush() or CreateHatchBrush()member function of CBrush initializes it.

```
// Two-stage: first construct the Brush
CBrush MyBrush;
// Then initialize it
MyBrush.CreateSolidBrush( RGB(255,0,0) ) ;

or for a hatched brush
MyBrush.CreateHatchBrush(HS_CROSS,RGB(255,0,0));
```

A Logbrush can also be constructed using a structure, LOGBRUSH, which is a structure that defines the style, width, and color of a Pen.
The LOGBRUSH structure has the following form:

```
typedef struct tag LOGBRUSH { /* lb */
  UINT     lbStyle;
  COLORREF lbColor;
  LONG     lbHatch;
} LOGBRUSH;
```

Initialise the members of the LOGBRUSH structure and then use CreateBrushIndirect() member function of CBrush. This method is used in listing 5.4 . The variable m_lbBrush of type LOGBRUSH is declared in CMainFrm class. Its members are initialized through the Menu. The member functions of CMainFrm class responsible for setting the LOGBRUSH structure are given below.
To change Color

```
OnColorBlue();
OnColorGreen();
OnColorRed();
```

To change Hatch Pattern

```
OnBrushCross();
OnBrushDiagonal();
OnBrushSolid();
OnBrushHorizontal();
```

After the program has created the new brush, a call to the paint DC's SelectObject() member function selects the brush into the DC and returns a pointer to the old brush:

```
CBrush* oldBrush = dc.SelectObject(&brush);
```

The rectangle and Ellipse is then drawn with this brush.

Rectangle() is just one of the shape-drawing functions you can call. Rectangle() takes as arguments the coordinates of the rectangle's upper-left and lower-right corners. When you run the application and look at the rectangle, you'll see that it is bordered by a thin red line. If you select a different pen into the DC, by selecting a different pen thickness through the menu, Windows will use that pen to draw the rectangle's border.

Shape-drawing functions frequently draw borders with the currently selected pen.

After drawing a rectangle, the program deselects the new brush from the DC.

```
dc.SelectObject(oldBrush);
```

5.3.4. Displaying Text.

Text can be diplayed on screen using the Paint DC’s TextOut member function. This is shown in the OnPaint() function in listing 5.4.

```
dc.SetTextColor(m_lpPen.lopnColor);//Changes Color of text

dc.TextOut(160,50,"Hello, How Are You"); //Displays Text
```

SetTextColor takes a COLORREF object as argument. All text drawn in the DC subsequently will be drawn with this color.
TextOut() function takes 3 arguments. They are the x and y coordinates of the starting point where the string has to appear and the string which and the third, a Cstring object that has to be drawn.

5.3.5. Raster Operation Modes.

Raster operation Modes are changed using the DC's SetROP2() member function. This function sets the current drawing mode. The drawing mode specifies how the colors of the pen and the interior of filled objects are combined with the color already on the display surface.

The drawing mode is for raster devices only; it does not apply to vector devices. Drawing modes are binary raster-operation codes representing all possible Boolean combinations of two variables, using the binary operators AND, OR, and XOR (exclusive OR), and the unary operation NOT.

SetROP2(int nDrawMode) takes one argument which is an integer representing the draw modes.
It can be any of the following values:

R2_BLACK Pixel is always black.

R2_WHITE Pixel is always white.

R2_NOP Pixel remains unchanged.

R2_NOT Pixel is the inverse of the screen color.

R2_COPYPEN Pixel is the pen color.

R2_NOTCOPYPEN Pixel is the inverse of the pen color.

R2_MERGEPENNOT Pixel is a combination of the pen color and the inverse of the screen color (final pixel = (NOT screen pixel) OR pen).

R2_MASKPENNOT Pixel is a combination of the colors common to both the pen and the inverse of the screen (final pixel = (NOT screen pixel) AND pen).

R2_MERGENOTPEN Pixel is a combination of the screen color and the inverse of the pen color (final pixel = (NOT pen) OR screen pixel).

R2_MASKNOTPEN Pixel is a combination of the colors common to both the screen and the inverse of the pen (final pixel = (NOT pen) AND screen pixel).

R2_MERGEPEN Pixel is a combination of the pen color and the screen color (final pixel = pen OR screen pixel).

R2_NOTMERGEPEN Pixel is the inverse of the R2_MERGEPEN color (final pixel = NOT(pen OR screen pixel)).

R2_MASKPEN Pixel is a combination of the colors common to both the pen and the screen (final pixel = pen AND screen pixel).

R2_NOTMASKPEN Pixel is the inverse of the R2_MASKPEN color (final pixel = NOT(pen AND screen pixel)).

R2_XORPEN Pixel is a combination of the colors that are in the pen or in the screen, but not in both (final pixel = pen XOR screen pixel).

R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color (final pixel = NOT(pen XOR screen pixel)).

6. DIALOG BOXES

As mentioned in the introduction to this chapter, dialog boxes are one way a Windows application can get information from the user. Chances are that your Windows application will have several dialog boxes, each designed to retrieve a different type of information from your user. However, before you can add these dialog boxes to your program, you must create them. To make this job simpler, Developer Studio includes an excellent resource editor. You must have already used the resource editor to create menus; you can use this handy visual tool to create all types of resources, including bitmaps, icons, string tables, and, of course, dialog boxes.

Because dialog boxes are used so extensively in Windows applications, MFC provides several classes that you can use to make dialog box manipulation easier and more convenient. Although you can use MFC's CDialog class directly, you'll most often derive your own dialog box class from CDialog in order to have more control over the way that your dialog box operates. However, MFC also provides classes for more specific types of dialog boxes, including CColorDialog, CFontDialog, and CFileDialog.

The minimum steps for adding a dialog box to your MFC application are as follows:

1.Create your dialog box resource using the Visual C++ dialog box editor. This process defines the appearance of the dialog box, including the types of controls it will contain.
2.Create an instance of CDialog, passing to the constructor your dialog box's resource ID and a pointer to the parent window.
3.Call the dialog box object's DoModal() member function to display the dialog box.

Although the preceding steps are all you need to display a simple dialog box, you'll usually need much more control over your dialog box than these steps provide. For example, this method of displaying a dialog box enables no way to retrieve data from the dialog box, which means the method is really only useful for dialog boxes that display information to the user—sort of like a glorified message box. To create a dialog box that you can control, perform the following steps:

1.Create your dialog box resource using the Visual C++ dialog box editor. This process defines the appearance of the dialog box as well as the controls that appear in the dialog box and the types of data those controls will return to your program.

2.Write a dialog box class derived from CDialog for your dialog box, including member variables for storing the dialog box's data. Initialize the member variables in the class's constructor.

3.Overload the DoDataExchange() function in your dialog box class. In the function, call the appropriate DDX and DDV functions to perform data transfer and validation.

4.In the window class that'll display the dialog box, create member variables for the dialog box controls whose data you want to store.

5.Create an instance of your dialog box class.

6.Call the dialog box object's DoModal() member function to display the dialog box.

7.When DoModal() returns, copy the dialog box data that you need to store from the dialog box object's member variables to the window class's matching variables.

In the rest of this chapter, you'll see how to perform all of the steps listed previously, including how to write your dialog box class and how to provide this class with automatic data transfer and validation. In the next section, you learn to use Developer Studio's dialog box editor.

6.1. INTRODUCING THE DIALOG APPLICATION

The Dialog application's main window displays the data collected from one of its dialog boxes.

The data displayed in the window are the default values for information that can be collected by one of the program's dialog boxes. On selecting the dialog menu command, a dialog box containing a number of controls appears. You can use these controls to enter information (see Figure 6.1). When you finish entering information, close the dialog box by clicking OK. The Dialog application's main window then displays the new information you entered into the dialog box's controls (see Figure 6.2).

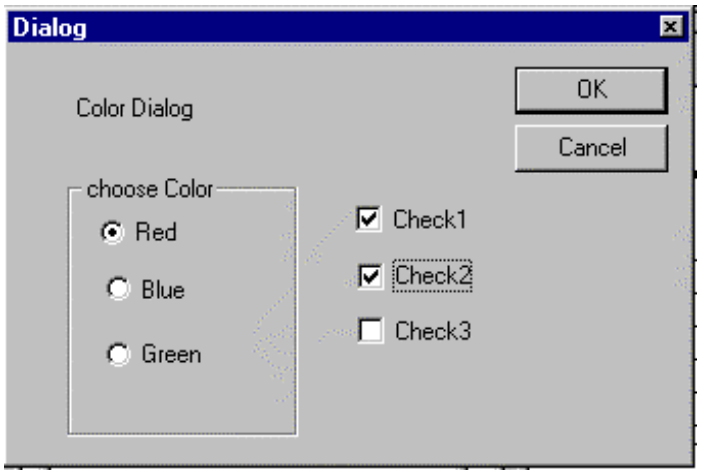


FIG. 6.1 This is the dialog box. You can make your selection in this

The dialog menu command reveals a dialog box containing several types of controls.

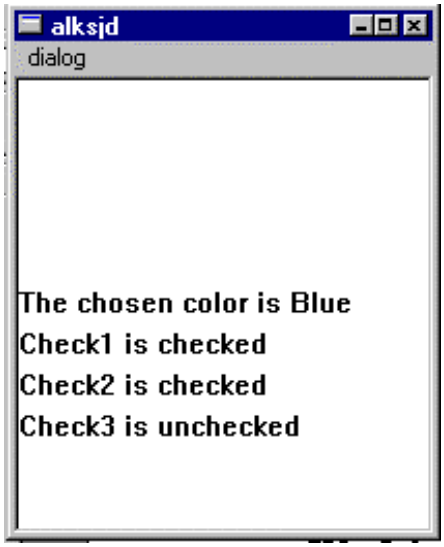


FIG. 6.2

After transferring data from the dialog box, you can display the user's choices in the window.

6.2. CREATING A DIALOG BOX RESOURCE

As you now know, the first step toward adding a dialog box to your MFC application is creating the dialog box resource, which acts as a sort of template for Windows. When Windows sees the dialog box resource in your program, it uses the commands in the resource to construct the dialog box for you. To learn how to create a dialog box resource, just perform the following steps:

1.Choose the Insert, Resource from the Developer Studio's menu bar. The Insert Resource dialog box appears. The Insert Resource dialog box enables you to select the type of resource you want to add to your project.

2.Double-click Dialog in the Resource Type box. The dialog box editor appears in one of the Developer Studio's panes

3.Use the dialog box editor to create your dialog box template by placing and editing the controls provided in the editor's toolbox.

4.Select controls on the toolbox and position them on the window of your dialog box, as shown in Figure 6.3.

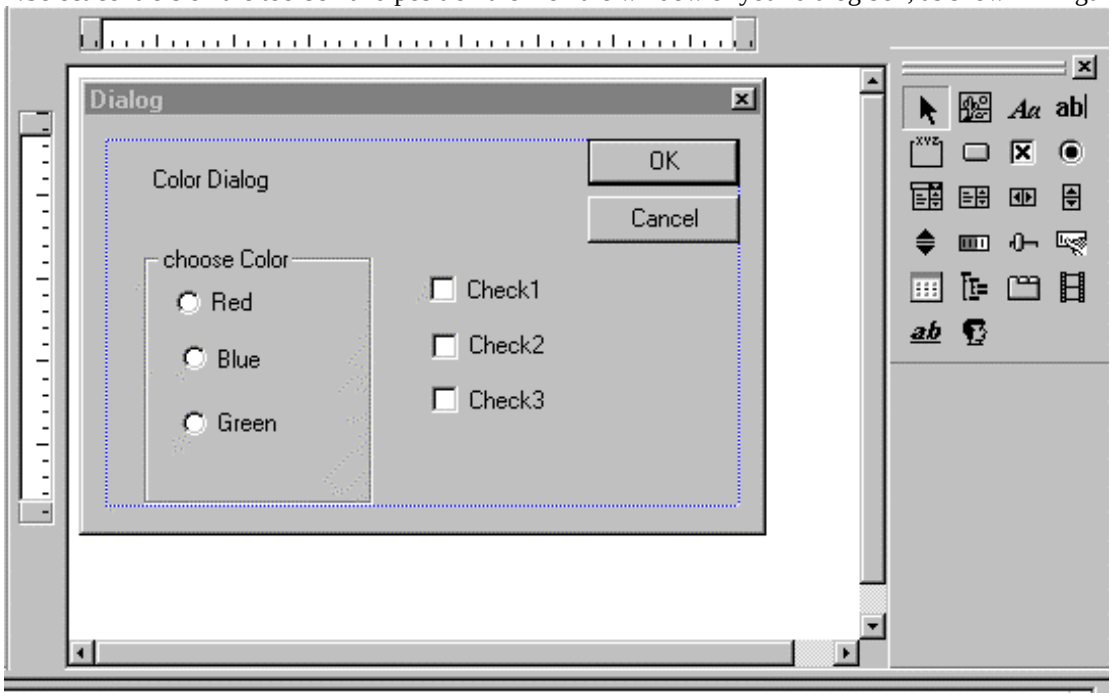


FIG. 6.3 The Dialog Editor

5.The dialog box editor's toolbox provides access to the controls you can place on your dialog box.

6.Double-click the controls you want to edit. The control's property sheet appears in which you can supply your own ID, caption, and other control information
Each control has its own property sheet in which you can enter the information needed to customize the control for use in your dialog box.

7.Select the dialog box, and then press Enter. The dialog box's property sheet, which enables you to customize the dialog box attributes, appears.
Just like controls, a dialog box has a property sheet that you can customize.

6.2.1. Defining Dialog Box and Control IDs

You might remember that, when you created your menu resource, you were able to choose menu IDs from predefined system IDs or create your own custom IDs. Because dialog boxes are often unique to an application (with the exception of the common dialog boxes), you will almost always create your own IDs for both the dialog box and the controls it contains. You can, if you like, accept the default IDs that the dialog box editor creates for you. However, these IDs are generic (i.e. IDD_DIALOG1, IDC_EDIT1, IDC_RADIO1, and so on.), and so you'll probably want to change them to something more specific.

In any case, as you can tell from the default IDs, a dialog box ID usually begins with the prefix IDD_, and control IDs usually begin with the prefix IDC_. You can, of course, use your own prefixes if you like, although sticking with conventions often makes your programs easier to read.

6.2.2. Grouping controls.

A Radio button (the Red, Green and Blue buttons in Fig6.3) are used in such cases where the user will select only one of the many options. In our example, the user has to select only one color out of Red, Green or Blue. On the contrary a checkbox is used when the user has to select multiple options out of a list. In the above case, a

user can select any one check box alone, any two or even all the three. In order to indicate a group of exclusive options as in the case of color selection, a group box is generally used.
In order to group the radiobuttons labeled Red, Green and Blue, follow the procedure below.

1. Mark the group and tabstop properties for the first radio button which is labeled Red. This will identify this radio button as the beginning of a group. Do not mark the group property for any of the radio buttons.
2.
3. Set the Tab order as shown in figure 6.4 so that the radio button group is in order and is followed by the check boxes. To mark the end of the radio button group, assign the group and tabstop properties to the next control (in tab order) after the last of the radio buttons, which will be the check box labeled check1.You can activate the radio buttons in the group with the mouse or with keyboard inputs.

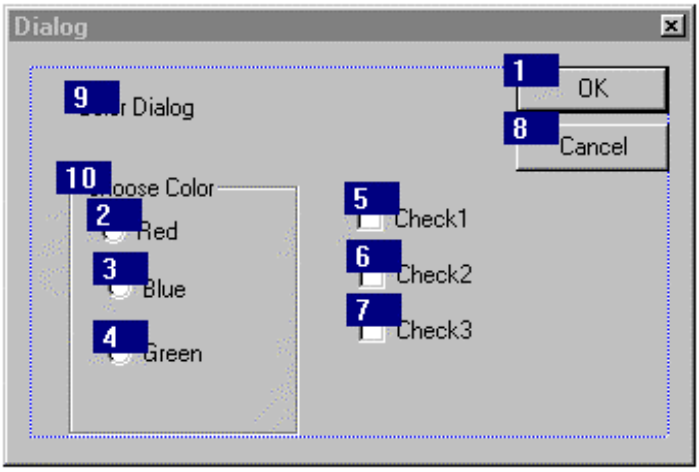


Fig.6.4 Tab Order for the controls

Note:
WS_GROUP Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined with the WS_GROUP style FALSE after the first control belong to the same group. The next control with the WS_GROUP style starts the next group (that is, one group ends where the next begins).
WS_TABSTOP Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style.

6.3. CREATING A DIALOG CLASS.

With our dialog box remaining open, open class wizard by selecting “view|Class Wizard” from the menu. The Add Class dialog appears. Click the ‘Create new class radio button and then OK. A ‘Create new Class ‘ dialog box appears which shows Cdialog as the base class and the dialog ID also. Type a name for the new class. Say CDlg1.
On clicking OK, the class wizard dialog box appears as shown in Fig6.5. The DoDataExchange has been automatically included.

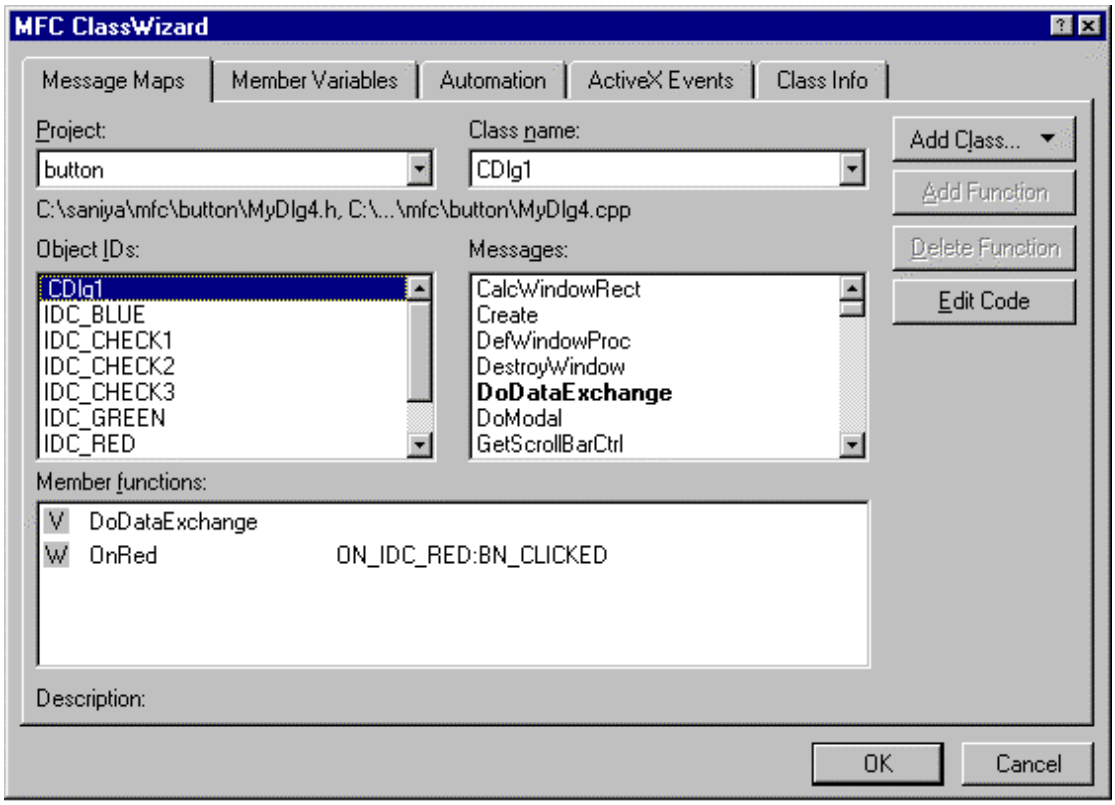


Fig.6.5 Class wizard with CDlg1 class and DoDataExchange

6.3.1. Creating member objects for the dialog.

Click on the member variable tab on the class wizard box to add the member object. See Fig6.6

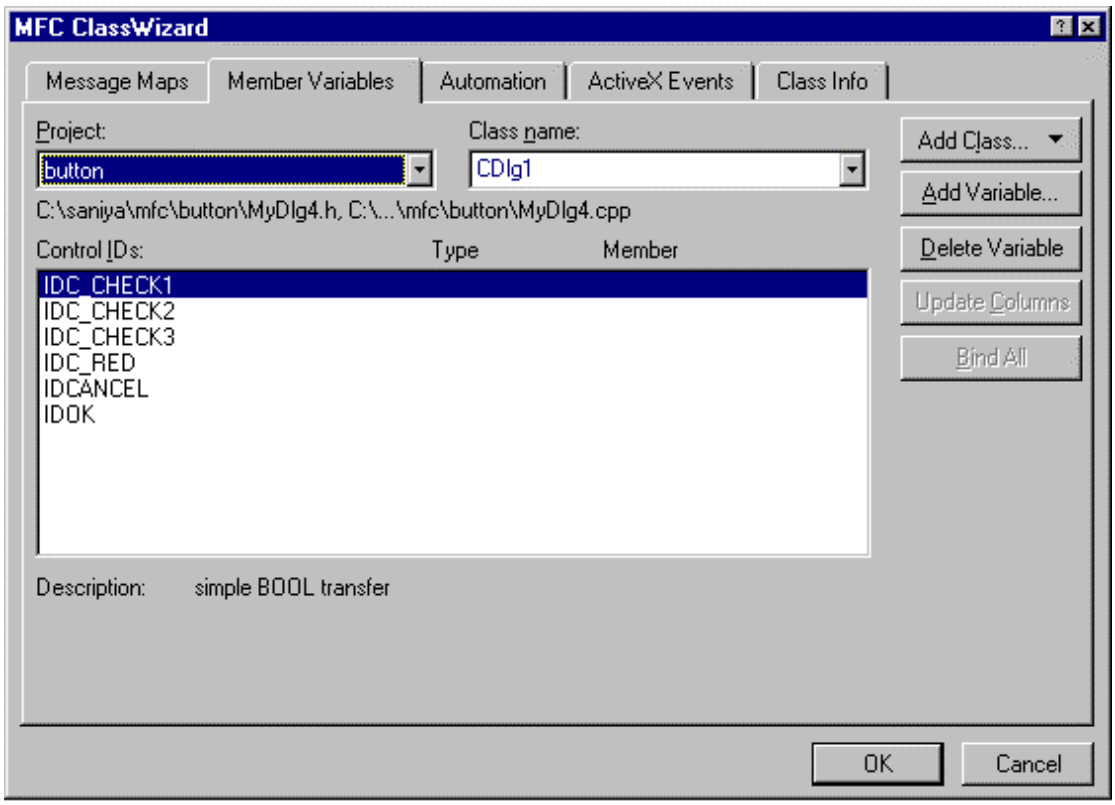


Fig6.6

6.3.2. Adding Member variables to the Dialog Class.

It is important to note that only the ID of the first Radio button in the group, IDC_RED appears. Only one member variable is required for a group of radiobuttons. This is due to the fact that DDX assigns a value to the integer member variable depending on which radiobutton is checked.
To add member variable for Check1 Checkbox, choose IDC_CHECK1. Then click on Add Variable button. Add member variable query box appears as in FIG6.7

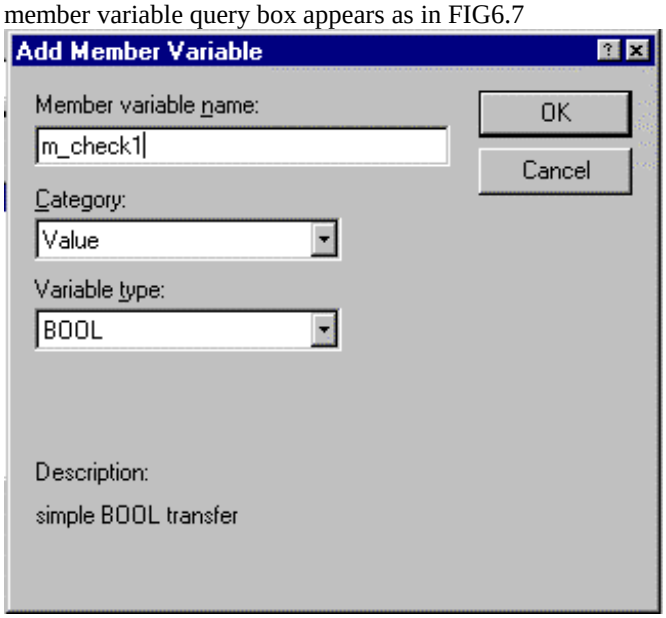


FIG6.7 Choosing member variables

Enter a member variable name. You must dropdown the catagory combobox and choose value as the category. When you do the variable type BOOL will be automatically filled in for you. Click OK to add the member object. Repeat to add member objects for the 3 check boxes and an integer variable for Radio buttons. When you are finished, the MFC class wizard should look as in figure 6.8

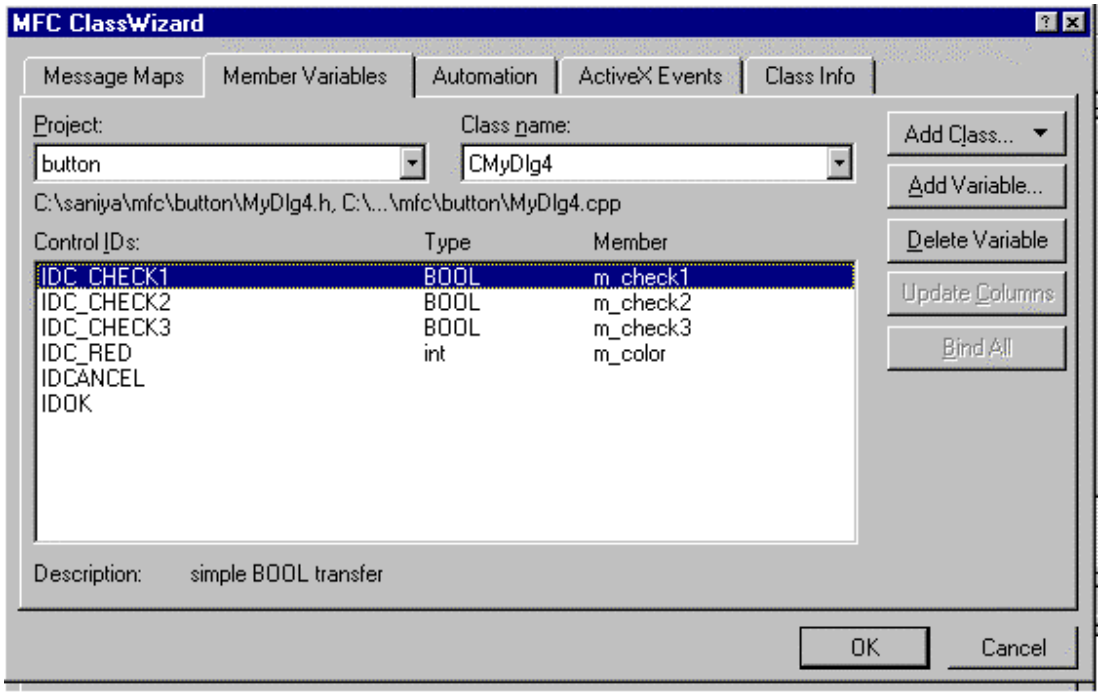


FIG 6.8 The finished Member variable box.

6.4. EXPLORING THE DIALOG APPLICATION

Listing 6.1 through Listing 6.4 are the most pertinent source code files for the Dialog application. Listing 6.1 and Listing6.2 constitute the application's CMainFrame class, which, of course, represents the application's main window. Listing 6.3 and Listing 6.4 are the source code files for the CDlg1 class, which represents the dialog box that appears when you select the application's dialog menu command.

Listing 6.1 MAINFRM.H—The Header File of the Main Window Class

```
////////////////////////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
//            represents the application's main window.
////////////////////////////////////////////////////////
class CMainFrame : public CFrameWnd
{
// Protected data members.
protected:
   BOOL m_check1;
        BOOL    m_check2;
        BOOL    m_check3;
        int m_radio;
// Constructor and destructor.
public:
   CMainFrame();
   ~CMainFrame();
// Overrides.
// Message map functions.
protected:
   // System message handlers.
   afx_msg void OnPaint();
   // Menu command message handlers.
   afx_msg void OnDialog();
   // Update command UI handlers.
   // None.

// Protected member functions.
protected:
   DECLARE_MESSAGE_MAP()
};
```

Listing 6.2 MAINFRM.CPP—The Implementation File of the Main Window Class

```
////////////////////////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//            class, which represents the application's
//            main window.
////////////////////////////////////////////////////////
#include <afxwin.h>
#include "mainfrm.h"
#include "resource.h"
#include "dlg1.h"
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   // Message map entries for system messages.
   ON_WM_PAINT()
   // Message map entries for menu commands.
   ON_COMMAND(IDM_DIALOG, OnDialog)
```

```
   // Message map entries for update command UI handlers.
   // None.
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////
// CMainFrame: Construction and destruction.
////////////////////////////////////////////////////////////
CMainFrame::CMainFrame()
{
   // Create the main frame window.
   Create(NULL, "Dialog App", WS_OVERLAPPEDWINDOW, rectDefault,
      NULL, MAKEINTRESOURCE(IDR_MENU1));
   // Initialize data members.
         m_radio=1;
         m_check1=TRUE;
         m_check2=FALSE;
         m_check3=FALSE;
}
CMainFrame::~CMainFrame()
{
}
////////////////////////////////////////////////////////////
// Overrides.
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
// Message map functions.
////////////////////////////////////////////////////////////
void CMainFrame::OnPaint()
{
   // Create a device context.

   CPaintDC dc(this); // device context for painting

         if (Dlg4.m_color==0)
               dc.TextOut(0,100,"The chosen color is Red");
         else if(Dlg4.m_color==1)
               dc.TextOut(0,100,"The chosen color is Green");
         else
               dc.TextOut(0,100,"The chosen color is Blue");

         if (m_check1==TRUE )
               dc.TextOut(0,120,"Check1 is checked");
         else
               dc.TextOut(0,120,"Check1 is unchecked");

         if (m_check2==TRUE )
               dc.TextOut(0,140,"Check2 is checked");
         else
               dc.TextOut(0,140,"Check2 is unchecked");

         if (m_check3==TRUE )
               dc.TextOut(0,160,"Check3 is checked");
         else
               dc.TextOut(0,160,"Check3 is unchecked");

}
void CMainFrame::OnDialog()
{
   // Create a new dialog-box object.
   CDlg1 dlg(this);
```

```
   // Copy the most current data into the dialog box.

            dlg.m_check1=m_check1;
            dlg.m_check2=m_check2;
            dlg.m_check3=m_check3;
            dlg.m_color=m_radio;
   // Display the dialog box.
   int result = dlg.DoModal();
   // If the user exited the dialog box with the OK button...
   if (result == IDOK)
   {
      // Copy the dialog's data into this class's data members.
            m_check1= dlg.m_check1;
            m_check2= dlg.m_check2;
            m_check3= dlg.m_check3;
            m_radio= dlg.m_color;
      // Force the window to show the new data.
      Invalidate();
   }
}
```

Listing 6.3 DLG1.H—The Header File of the CDlg1 Class

```
//////////////////////////////////////////////////////////
// DLG1.H: Header file for the CDlg class.
//////////////////////////////////////////////////////////
class CDlg1 : public CDialog
{
// Constructor.
public:
   CDlg1(CWnd* pParent);
// Data transfer variables.
public:
            int                 m_color;
            BOOL    m_check1;
            BOOL    m_check2;
            BOOL    m_check3;
// Overrides.
protected:
   virtual void DoDataExchange(CDataExchange* pDX);
};
```

Listing 6.4 DLG1.CPP—The Implementation File of the CDlg1 Class

```
//////////////////////////////////////////////////////////
// DLG1.CPP: Implementation file for the CDLG1 class.
//////////////////////////////////////////////////////////
#include <afxwin.h>
#include "dialog.h"
#include "resource.h"
#include "dlg1.h"
//////////////////////////////////////////////////////////
// CONSTRUCTOR
//////////////////////////////////////////////////////////
CDlg1::CDlg1(CWnd* pParent) : CDialog(IDD_TESTDIALOG, pParent)
```

```
{
   // Initialize data transfer variables.
         m_color = 1;
         m_check1 = FALSE;
         m_check2 = FALSE;
         m_check3 = FALSE;
}
/////////////////////////////////////////////////////////
// Overrides.
/////////////////////////////////////////////////////////
void CDlg1::DoDataExchange(CDataExchange* pDX)
{
   // Call the base class's version.
   CDialog::DoDataExchange(pDX);
   // Associate the data transfer variables with
   // the ID's of the controls.
         DDX_Radio(pDX, IDC_RED, m_color);
         DDX_Check(pDX, IDC_CHECK1, m_check1);
         DDX_Check(pDX, IDC_CHECK2, m_check2);
         DDX_Check(pDX, IDC_CHECK3, m_check3);
}
```

The CDlg1 class, which is derived from CDialog, provides the needed data members, as shown in Listing 6.5.

Listing 6.5 Declaring Data Members for the Dialog Box Controls

```
public:
         int            m_color;
         BOOL   m_check1;
         BOOL   m_check2;
         BOOL   m_check3;
```

Notice that these variables are declared as public data members of the class. This is important because if they were declared as private or protected, your program would not be able to access them outside of the class. (Yes, I know, this is another case where MFC breaks the rules of strict object-oriented design, which dictates that a class's data member should never be accessed from outside of the class.)

The variables you declare for your dialog box's controls must, of course, hold the appropriate type of data for the controls with which they're associated. For example, text boxes that return strings must be associated with string variables, whereas controls that return integers must be associated with integer variables. To determine the correct data type for a particular control, please refer to your Visual C++ online documentation or Windows programming manual.

Besides the data members, your dialog box class must supply a constructor and must override the virtual DoDataExchange() function, which is where the data transfer occurs. The CDlg1 class's constructor is declared like this:

```
public:
   CDlg1(CWnd* pParent);
```

The constructor's single argument is a pointer to the dialog box's parent window. As you'll soon see, the constructor's implementation will pass this pointer, as well as the dialog box template's resource ID, on to the CDialog class's constructor.

The CDlg1 class overrides the DoDataExchange() function like this:

```
protected:
```

```
   virtual void DoDataExchange(CDataExchange* pDX);
```

As you can see, DoDataExchange()'s only argument is a pointer to a CDataExchange object, which is responsible for handling the actual transfer of the data between the dialog box's controls and the class's data members.

With the class's header file ready to go, you can start writing the class's implementation. As always, the first step is to include the header files required to compile the code, as shown in Listing 6.6.

Listing 6.6 Including the Appropriate Header Files

```
#include <afxwin.h>
#include "dialog.h"
#include "resource.h"
#include "dlg1.h"
```

As you might remember from other programs, the first line above includes the general header file for MFC programs. The second line brings in the application class's declaration; the third is your resource IDs; and the fourth is the CDlg1 class's declaration.

The CDlg1 class's constructor is responsible for initializing the dialog box object, as shown in Listing 6.7.

Listing 6.7 Constructing a CDlg1 Object

```
CDlg1::CDlg1(CWnd* pParent) : CDialog(IDD_TESTDIALOG, pParent)
{
   // Initialize data transfer variables.
 m_color = 1;
            m_check1 = FALSE;
            m_check2 = FALSE;
            m_check3 = FALSE;

}
```

The CDlg1 constructor receives a single parameter, which is a pointer to the dialog box's parent window. The constructor passes this pointer, along with the dialog box's resource ID, to the base class's constructor. It is the resource ID that links the correct dialog box template to the dialog box class. Because you've built the ID into the class's constructor, you don't need to worry about it when you create the dialog box object. You need only supply the pointer to the parent window.

Inside the constructor, the program initializes the data members that represent the contents of the dialog box's controls. The values stored in these variables will automatically be copied to the dialog box's controls when the dialog box is displayed. When the user dismisses the dialog box, MFC copies the data from the controls into these variables, where they can be accessed by other parts of your program.

The only other function in the dialog box class is the overridden DoDataExchange() function, which is shown in Listing 7.9.

Listing 6.8 Overriding the DoDataExchange() Function

```
void CDlg1::DoDataExchange(CDataExchange* pDX)
{
 CDialog::DoDataExchange(pDX);
   // Associate the data transfer variables with
   // the ID's of the controls.
            DDX_Radio(pDX, IDC_RED, m_color);
            DDX_Check(pDX, IDC_CHECK1, m_check1);
```

```
    DDX_Check(pDX, IDC_CHECK2, m_check2);
    DDX_Check(pDX, IDC_CHECK3, m_check3);

}
```

It is the DoDataExchange() function that sets up MFC's capability to transfer data between the data members of the dialog box class and the controls of the dialog box. The function receives a single parameter, which is a pointer to a CDataExchange object. This pointer is supplied by MFC when it calls your overridden version of the function. You don't have to do anything with this pointer except pass it on to the base class's DoDataExchange() function, as well as to various DDX and DDV functions you'll call to set up the dialog box's data transfer. The first line of your DoDataExchange() function, as shown in Listing 6.8, should call the base class's version, passing along the pointer to the CDataExchange object.

What are DDX and DDV functions? An excellent question! MFC's DDX functions set up the link between a data member and a control. For example, in Listing 6.8, the call to DDX_Radio() links the RadioButton control group whose ID begins with IDC_RED to the data member m_color. This tells MFC to map the contents of m_color to the radio button control group when the dialog box is displayed and to map the data in the control group back to m_color when the dialog box is dismissed. Notice that a DDX function's first argument is the CDataExchange pointer passed into the DoDataExchange() function. There is a DDX function for each type of control you might want to include in a data transfer. These functions are listed in Table 6.1. To learn about each function's parameters, look them up in your Visual C++ online documentation.

Table 6.1 DDX Functions

Function	**Description**
DDX_CBIndex()	Links a combo box's index to an integer variable
DDX_CBString()	Links a combo box's string to a string variable
DDX_CBStringExact()	Links a combo box's selected string to a string variable
DDX_Check()	Links a check box with an integer variable
DDX_LBIndex()	Links a list box's index with an integer variable
DDX_LBString()	Links a list box's string to a string variable
DDX_LBStringExact()	Links a list box's selected string to a string variable
DDX_Radio()	Links a radio button with an integer variable
DDX_Scroll()	Links a scroll bar to an integer variable
DDX_Text()	Links a text box to a string variable

MFC's DDV functions perform data validation for controls.

Table 6.2 DDV Functions

Function	**Description**
DDV_MaxChars()	Limits the length of a string
DDV_MinMaxByte()	Limits a byte value to a specific range
DDV_MinMaxDouble()	Limits a double value to a specific range
DDV_MinMaxDWord()	Limits a DWORD value to a specific range
DDV_MinMaxFloat()	Limits a floating-point value to a specific range
DDV_MinMaxInt()	Limits an integer value to a specific range
DDV_MinMaxLong()	Limits a long integer value to a specific range
DDV_MinMaxUInt()	Limits an unsigned integer value to a specific range

It's important that you call DDV functions immediately after the DDX function that sets up the data exchange for a control. When you do this, MFC can set the input focus to the control that contains invalid data—not only showing the user exactly where the problem is, but also enabling him to enter a new value as conveniently as possible.

6.4.1. Using the Dialog Box Class

Now that you have your dialog box class written, you can create objects of that class within your program and display the associated dialog box element. The first step in using your new class is to include the header file of the dialog box class in any class that'll access the class. Failure to do this will cause the compiler to complain that it doesn't recognize the dialog box class. Also, when you develop the window class that'll display the dialog box, you'll usually want to create data members that mirror the data members of the dialog box class. This gives you a place to store information that you transfer from the dialog box. The CMainFrame class declares this set of member variables, as shown in Listing 6.9.

Listing 6.9 Declaring Storage for Dialog Box Data

```
protected:
  BOOL m_check1;
       BOOL   m_check2;
       BOOL   m_check3;
       int m_radio;
```

Notice that the data members declared in Listing 6.9 have the same names as the equivalent data members in the CDlg1 class. This convention makes it easier to keep track of what the variables do. Notice also that, unlike the equivalent variables in the CDlg1 class, the variables of the CMainFrame class are declared as protected, rather than as public. This is because no other class requires access to these variables in the same way that other classes require access to the variables of the dialog box class.

Just as the dialog box class initializes its data members in its constructor, so too does the CMainFrame class, as shown in Listing 6.10.

Listing 6.10 Initializing the Frame Window's Data Members

```
m_radio=1;
m_check1=TRUE;
m_check2=FALSE;
m_check3=FALSE;
```

In the CMainFrame class, the program displays the dialog box in the OnDialog() function, which MFC calls when the user chooses the Dialog command from the application's menu bar. The OnDialog() function first creates a dialog object of your new class, like this:

```
CDlg1 dlg(this);
```

With the dialog box object created, the program can now access the data members of the dialog box and initialize them to the current contents of the values stored in the CMainFrame class, as shown in Listing 6.11.

Listing 6.11 Transferring Data to the Dialog Box Object

```
dlg.m_check1=m_check1;
dlg.m_check2=m_check2;
dlg.m_check3=m_check3;
dlg.m_color=m_radio;
```

You might wonder why the main frame window class bothers to initialize the dialog box's data members before displaying the dialog box. In this program, the dialog box always appears with the last entered data in the dialog box because the dialog box's data disappears along with the dialog box when the dialog box is deleted (either by the object going out of scope or by being explicitly deleted with the delete operator). The only place the current dialog box data is saved is in the CMainFrame class. If you wanted the dialog box to always appear with its original default values, you wouldn't bother to copy the stored values into the data members of the dialog box, but instead would stick with the values supplied by the dialog box's own constructor.

After the program initializes the dialog box's contents, the program calls the DoModal() function of the dialog box object to display the dialog box to the user, like this:

```
int result = dlg.DoModal();
```

The DoModal() function handles all the user's interactions with the dialog box, which, in this case, amounts to little more than waiting for the user to click the OK button. When the user exits the dialog box, the DoModal() function returns, and your program can continue. Because the dialog box object is created locally on the stack, it is automatically deleted when it goes out of scope, which is when the OnDialog() function exits.

At this point, the user has control until he dismisses the dialog box. If the user exits by pressing the OK button, DoModal() returns the value IDOK. (The Cancel button causes DoModal() to return IDCANCEL.) In the case of IDOK, the program must copy the new data from the dialog box to the CMainFrame class's variables, as shown in Listing 6.12.

Listing 6.12 Retrieving Data from the Dialog Box

```
if (result == IDOK)
   {
      // Copy the dialog's data into this class's data members.
           m_check1= dlg.m_check1;
           m_check2= dlg.m_check2;
           m_check3= dlg.m_check3;
           m_radio= dlg.m_color;
           Invalidate();
}
```

The call to Invalidate() in Listing 6.12 forces the window's contents to be redrawn, this time using the new values retrieved from the dialog box. When the OnDialog() function ends, the dialog box object goes out of scope and disappears along with the data that the user entered into it. It's a good thing you saved that data in the CMainFrame class!

When a dialog box is displayed or closed, MFC's data-transfer mechanism handles moving data to and from the dialog box's controls. However, there might be times in your dialog box class when you want to force the data transfer to occur. You can do this by calling the UpdateData(BOOL bSaveAndValidate) function. This function's single argument is a Boolean value indicating whether the dialog box should have data transferred to (bSaveAndValidate = FALSE) or from (bSaveAndValidate = TRUE) its controls.
As you've learned, handling dialog boxes in an MFC program is much easier than handling them in a conventional Windows program because MFC provides a powerful class, CDialog, from which you can derive custom dialog box classes. These custom classes, not only perform automatic data exchange, but also validate the contents of a dialog box's controls before the user is allowed to dismiss the dialog box.

6.5. COMMON DIALOG CLASSES

In addition to class CDialog, MFC supplies several classes derived from CDialog that encapsulate commonly used dialog boxes, as shown in the following table. The dialog boxes encapsulated are called the "common dialog boxes" and are part of the Windows common dialog library (COMMDLG.DLL). The dialog-template resources and code for these classes are provided in the Windows common dialog boxes that are part of Windows versions 3.1 and later.

Common Dialog Classes

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

To construct a common dialog object, use the provided constructor or derive a new class and use your own custom constructor.

After initializing the dialog box's controls, call the DoModal member function to display the dialog box and allow the user to make a selection. DoModal returns the user's selection of either the dialog box's OK (IDOK) or Cancel (IDCANCEL) button.

If DoModal returns IDOK, you can use Common Dialog Classe's member functions to retrieve the information input by the user.

For eg GetColor() member function of CColorDialog will retrieve the user selection. In the case of CFileDialog, GetFileName() member function returns the filename.

Eg.

```
        CColorDialog BrColorDlg(SelectedColor,0,this);
    if(BrColorDlg.DoModal() ==IDOK)
        SelectedColor=BrColorDlg.GetColor();
```

SelectedColor is a COLORREF variable which holds the selected color.

6.6. USING OTHER CONTROLS WITH DIALOG BOXES.

There are several types of controls you can place in a dialog box, including check boxes, radio buttons, list boxes, combo boxes, and scroll bars. You should already be familiar with how these controls work, both from a user's and a programmer's point of view. If you've never programmed with Microsoft Foundation Classes, you may not be familiar with the classes with which MFC encapsulates each of the window controls. These classes include CButton, CEdit, CStatic, CListBox, and CComboBox.

Although MFC supplies classes that encapsulate the many window controls, Windows provides most of the services needed to handle these controls. A description of each window control follows:

- Static text Static text is a string of characters usually used to label other controls in a dialog box or window. Although it is considered to be a window control, static text cannot be manipulated by the user. You use the CStatic class to create and manipulate static text.
- Edit box An edit control accepts text input from the user. The user can edit the text in various ways before completing the input. You use the CEdit class to create and manipulate an edit box.
- Pushbutton A pushbutton (also simply called a "button")is a graphical object that triggers a command when the user clicks it. When clicked, a button's graphical image is usually animated to appear as if the button is pressed and released. You use the CButton class to create and manipulate pushbuttons.
- Check box A check box is a special type of button that toggles a check mark when clicked. Check boxes usually represent program options that the user can select. You use the CButton class to create and manipulate check boxes.
- Radio button Radio buttons are similar to check boxes, except only one radio button in a group can be selected at a given time. Radio buttons usually represent program options that are mutually exclusive. You use the CButton class to create and manipulate radio buttons.
- Group box Often, check boxes and radio buttons are placed into group boxes, which organize the buttons into logical groups. The user cannot interact with group boxes. You use the CStatic class to create and manipulate group boxes.
- List box A list box is a rectangle containing a set of selections. These selections are usually text items but can also be bitmaps or other objects. Depending on the list box's style flags, the user may select one or several objects in the list box. You use the CListBox class to create and manipulate a list box control.
- Combo box A combo box is similar to a list box, except it also includes an edit control in which the user can type a selection. You use the CComboBox class to create and manipulate combo boxes.
- Scroll bar A scroll bar is a graphical object containing a track that encloses a sliding box called the scroll box. By positioning the scroll box, the user can select a value from a given range. In addition to the scroll box, a scroll bar contains arrow boxes that, when clicked, move the scroll box a unit in the direction of the arrow. Although scroll bars are rarely used in dialog boxes, they can be created and manipulated by the CScrollBar class.

As mentioned previously, you can add any of these controls to a dialog box simply by using Developer Studio's dialog box editor. Figure 6.9 shows the dialog box under construction in this chapter's first program, Control1.

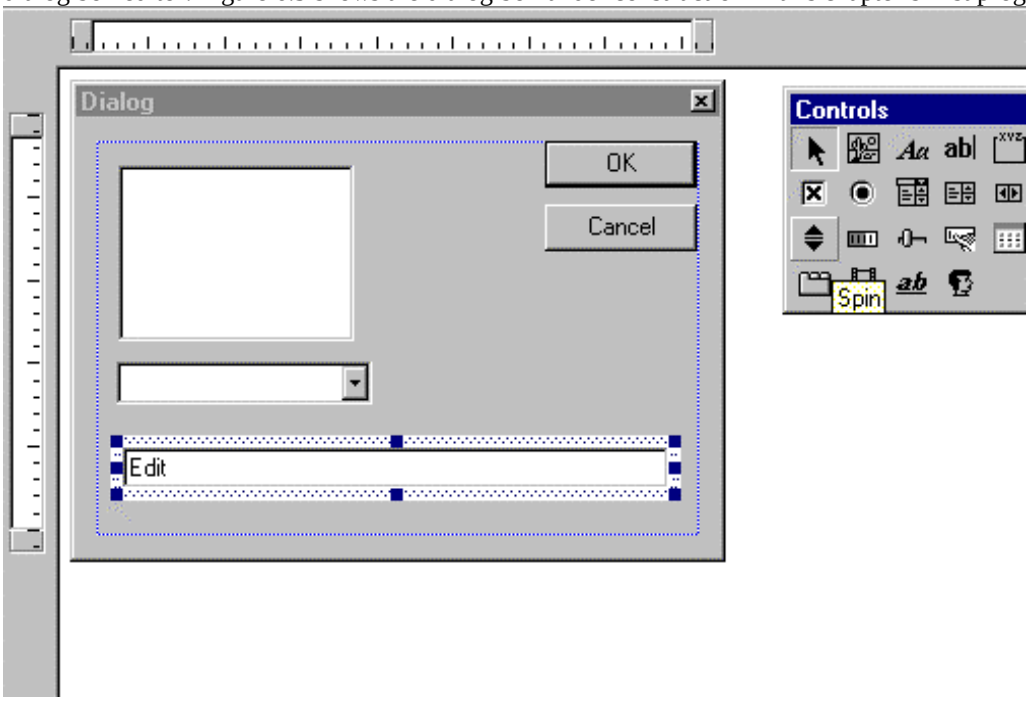


FIG. 6.9 Dialog box under construction

The dialog box under construction in Developer Studio's resource editor contains several standard Windows controls.
The controls in the dialog box are (apart from OK and Cancel buttons), a Listbox, a Combobox and Edit box.

6.7. INTRODUCING THE CONTROL1 APPLICATION

We are now going to develop an application, control1 which will show the functionality of listbox, combobox nad edit controls. When you run Control1, you see a window with a menu command Dialog. When this menu command is clicked, a dialog box appears as shown in fig6.10.

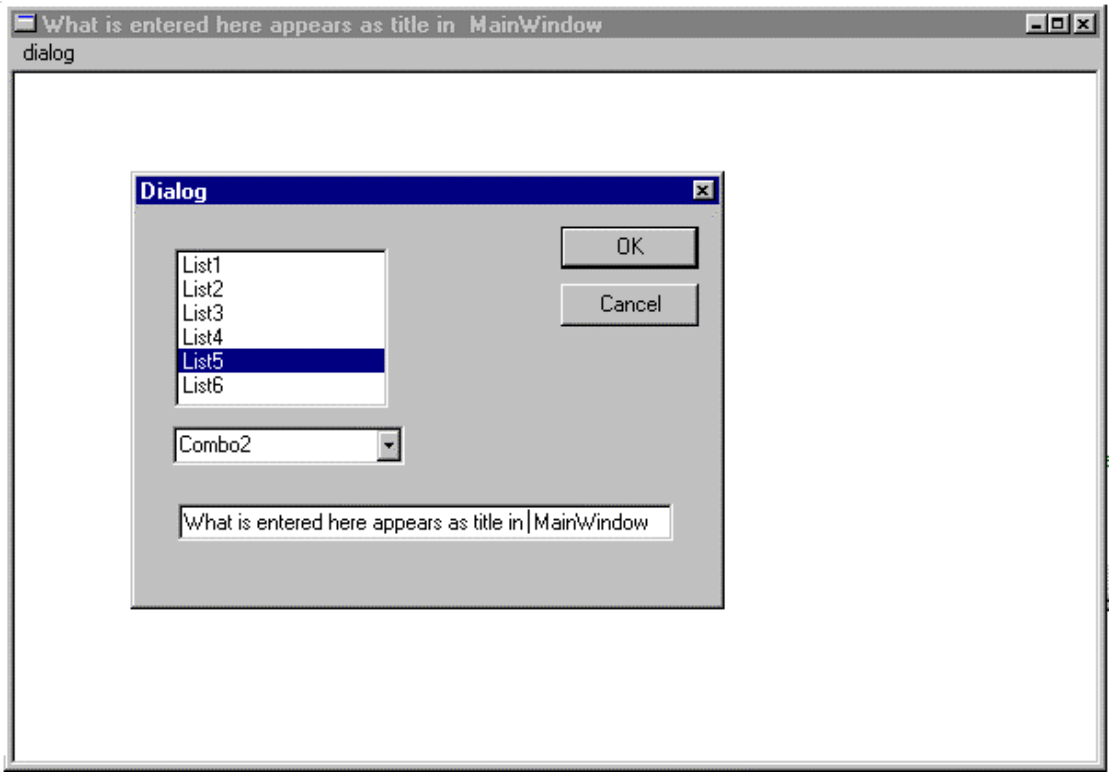


FIG6.10, MainWindow and the dialog box

The application is such that, what is entered in the Edit box control of the dialog box appears as title of the Main Window. This can be seen in Fig6.10. As you change the text in the edit control, the main window title changes. You can make a selection from the list box and combobox. On clicking OK, the dialog box disappears. The selections made in the list box and combo box will be displayed in the main window as shown in fig6.11

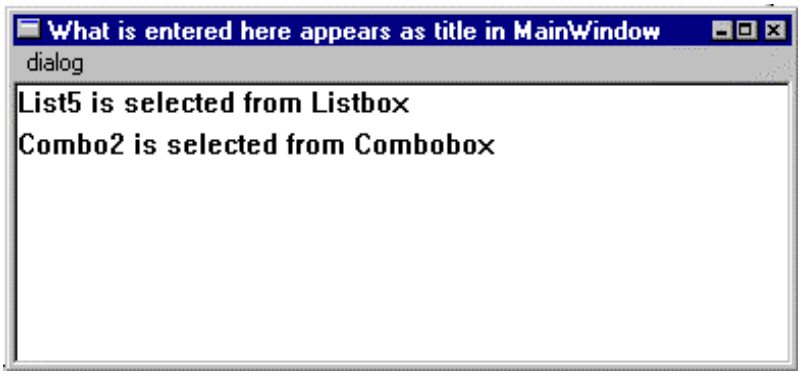


FIG. 6.11 Main window after selection is made in the Dialog box

The application's main window shows the controls' settings.

6.8. EXPLORING THE CONTROL1 APPLICATION

The controls in the application's dialog box are associated with MFC control classes so that the program can directly manipulate the controls. In this section, you'll see how this bit of MFC trickery is accomplished.

Listings 6.13 through 6.16 are the most pertinent source code files for the Control1 application. Listings 6.13 and 6.14 comprise the application's CMainFrame class, which represents the application's main window.

Listing 6.13 MAINFRM.H The Header File for the Main Window Class

```
//////////////////////////////////////////////////////////
// MAINFRM.H: Header file for the CMainFrame class, which
```

```
//          represents the application's main window.
//////////////////////////////////////////////////////////
class CMainFrame : public CFrameWnd
{
// Protected data members.
protected:
   CString m_combo;
          CString m_edit;
          CString m_list;
// Constructor and destructor.
public:
   CMainFrame();
   ~CMainFrame();
// Overrides.
protected:

// Message map functions.
protected:
   // System message handlers.
   afx_msg void OnPaint();
   // Menu command message handlers.
   afx_msg void OnDialog();
   // Update command UI handlers.
   // None.

// Protected member functions.
protected:
   DECLARE_MESSAGE_MAP()
};
```

Listing 6.14 MAINFRM.CPP The Implementation File for the Main Window Class

```
//////////////////////////////////////////////////////////
// MAINFRM.CPP: Implementation file for the CMainFrame
//          class, which represents the application's
//          main window.
//////////////////////////////////////////////////////////
#include <afxwin.h>
#include "mainfrm.h"
#include "resource.h"
#include "cntldlg.h"
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   // Message map entries for system messages.
   ON_WM_PAINT()
   // Message map entries for menu commands.
   ON_COMMAND(IDM_DIALOG, OnDialog)
   // Message map entries for update command UI handlers.
   // None.
END_MESSAGE_MAP()
//////////////////////////////////////////////////////////
// CMainFrame: Construction and destruction.
//////////////////////////////////////////////////////////
CMainFrame::CMainFrame()
{
   // Create the main frame window.
   Create(NULL, "Control App", WS_OVERLAPPEDWINDOW, rectDefault,
```

```
      NULL, MAKEINTRESOURCE(IDR_MENU1));
   // Initialize the class's data members.
         m_combo = “Combo1”;
         m_edit = “Default”;
         m_list = “List1”;
}
CMainFrame::~CMainFrame()
{
}
//////////////////////////////////////////////////////////
// Overrides.
//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
// Message map functions.
//////////////////////////////////////////////////////////
void CMainFrame::OnPaint()
{
   // Create a device context.
         CPaintDC dc(this);
         dc.TextOut(0,0,m_list+CString(" is selected from Listbox"));
         dc.TextOut(0,20,m_combo+CString(" is selected from Combobox"));
}
void CMainFrame::OnDialog()
{
   // Create and display the dialog box.
   CCntlDlg theDialog(this);
         theDialog.m_edit=m_edit;
         theDialog.m_list = m_list;
         theDialog.m_combo = m_combo;
         if(theDialog.DoModal()==IDOK)
         {
                  m_edit = theDialog.m_edit;
                  m_list = theDialog.m_list;
                  m_combo = theDialog.m_combo;

         }
      Invalidate();
   }
```

Listings 6.15 and 6.16 are the source code files for the CCntlDlg class, which represents the dialog that appears when you select the application's Dialog command.

Listing 6.15 CNTLDLG.H The Header File for the Dialog Box Class

```
//////////////////////////////////////////////////////////
// CNTLDLG.H: Header file for the CCntlDlg class.
//////////////////////////////////////////////////////////
class CMainFrame;
class CCntlDlg : public CDialog
{
// Class data members.
public:
   CString        m_combo;
         CString  m_edit;
         CString  m_list;
// Constructor.
public:
```

```
   CCntlDlg(CWnd* pParent);
// Overrides.
protected:

        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
        virtual BOOL OnInitDialog();
         void OnChangeEdit1();
        DECLARE_MESSAGE_MAP()
};
```

Listing 6.16 CNTLDLG.CPP The Implementation File for the Dialog Box Class

```
////////////////////////////////////////////////////////
// CNTLDLG.CPP: Implementation file for the CCtrlDlg class.
////////////////////////////////////////////////////////
#include <afxwin.h>
#include "resource.h"
#include "cntldlg.h"
////////////////////////////////////////////////////////
// CONSTRUCTOR
////////////////////////////////////////////////////////
CCntlDlg::CCntlDlg(CWnd* pParent) :
   CDialog(IDD_CONTROLDIALOG, pParent)
{
}
////////////////////////////////////////////////////////
// Overrides.
////////////////////////////////////////////////////////
BOOL CCntlDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
        CListBox* pList = (CListBox*)GetDlgItem(IDC_LIST1);
        pList->AddString("List1");
        pList->AddString("List2");
        pList->AddString("List3");
        pList->AddString("List4");
        pList->AddString("List5");
        pList->AddString("List6");
        CComboBox* pCombo = (CComboBox*)GetDlgItem(IDC_COMBO1);
        pCombo->AddString("Combo1");
        pCombo->AddString("Combo2");
        pCombo->AddString("Combo3");
        // TODO: Add extra initialization here
        UpdateData(FALSE);
        return TRUE;
}

void CCntlDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(MyDialog)
        DDX_CBString(pDX, IDC_COMBO1, m_combo);
        DDX_Text(pDX, IDC_EDIT1, m_edit);
        DDX_LBString(pDX, IDC_LIST1, m_list);
        //}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CCntlDlg, CDialog)
        //{{AFX_MSG_MAP(CCntlDlg)
        ON_EN_CHANGE(IDC_EDIT1, OnChangeEdit1)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CCntlDlg::OnChangeEdit1()
{

        UpdateData(TRUE);
        GetParent()->SetWindowText(m_edit);

}
```

6.8.1. Associating MFC Classes with Controls

MFC features many classes for window controls. The control classes were mentioned at the beginning of this chapter. Up until now, however, you haven't used these classes with the controls you added to your dialog boxes. This is because, in many cases, you don't to need manipulate a control at that level. You just display your dialog box and let the controls take care of themselves. There are times, though, when it's handy to be able to manipulate a control directly, which is what the many control classes such as CEdit, CButton, and CListBox enable you to do.

Each of the control classes features member functions that do everything from initialize the contents of the control to respond to Windows messages. But to use these member functions, you must first associate the control with the appropriate class. For example, to manipulate an list box through MFC, you must first associate that control with the CListBox class. Then, you can use the CListBox class's member functions to manage the control.

To associate a control with a class, you must first get a pointer to the control. You can do this easily by calling the GetDlgItem() member function, which a dialog box class inherits from CWnd. You call GetDlgItem() like this:

```
CListBox* pList = (CListBox*)GetDlgItem(IDC_LIST1);
```

The GetDlgItem() function returns a pointer to a CWnd object. (Remember: Every control class has CWnd as a base class.) It's single argument is the resource ID of the control for which you want the pointer. To gain access to the member functions of a control class, the returned CWnd pointer must be cast to the appropriate type of pointer. In the above line, you can see that GetDlgItem()'s return value is being cast to a CListBox pointer.

Once you have the pointer, you can access the class's member functions through that pointer.

6.8.2. Initializing the Dialog Box's Controls

The dialog box class first declares in its header file data members for holding the information the user entered into the dialog box, as shown in Listing 6.17.

Listing 6.17 Declaring Data Members for the Dialog Box Class

```
public:
        CString  m_combo;
        CString  m_edit;
        CString  m_list;
```

As you can see, there is one variable for each dialog box control with which the user can interact.

If you recall, the Control1 application's dialog box appears with default values already selected in its controls. For example, the edit box appears with the text "Default". The List box shows a list of contents. Same is the case with the combo box. Obviously, these controls are being initialized somewhere in the program and that somewhere is the CCntlDlg class's OnInitDialog() function. The OnInitDialog() function gets called as part of

the dialog box creation process (specifically, it responds to the WM_INITDIALOG Windows message). Because OnInitDialog() is a virtual function of the CDialog class, however, you don't need to create an entry in a message map. Just override the function in your custom dialog box class.

In your overridden OnInitDialog(), you must first call the base class's version, like this:

```
CDialog::OnInitDialog();
```

Then, you can perform whatever special initialization is required by your dialog box class. In the CCntlDlg class, that initialization is setting items to be listed in the list box and combo box. Initializing a list box, takes a little extra work, as shown in Listing 6.18.

Listing 6.18 Initializing a List Box

```
CListBox* pList = (CListBox*)GetDlgItem(IDC_LIST1);
        pList->AddString("List1");
        pList->AddString("List2");
        pList->AddString("List3");
        pList->AddString("List4");
        pList->AddString("List5");
        pList->AddString("List6");
```

The AddString() member function of the CListBox class, adds a string to the contents of the list box. So, the code in Listing 6.18 adds the six selections that the user can choose from the list box.

A combo box is not unlike a list box when it comes to initialization, as you can see in Listing 6.19.

Listing 6.19 LST08_07.CPP Initializing a Combo Box

```
CComboBox* pCombo = (CComboBox*)GetDlgItem(IDC_COMBO1);
        pCombo->AddString("Combo1");
        pCombo->AddString("Combo2");
        pCombo->AddString("Combo3");
```

The preceding lines work exactly like the similar lines used to initialize the list box, adding 3 strings to the combo box's list.
The last statement in the OnInitDialog() function as follows.

```
        UpdateData(FALSE);
```

UpdateData() is used to initialize data in a dialog box or to retrieve data from the dialog box, depending on whether the argument is TRUE or FALSE.
In this application, UpdateData(TRUE) transfers data from the variables, m_combo, m_list, m_edit to their respective controls in the dialog. These variables are initialized before the dialog is being called in the CMainFrame class.

To display the contents of the Textbox as title of Main window, you have to respond to the EN_CHANGE message of the Edit control. This is easily done through the class wizard. Select the ‘View|Class Wizard’ menu. The MFC Class wizard dialog box appears. Select the dialog class name, CCntlDlg from the Class name comobobox. Choose the Edit box’s ID , IDC-EDIT1 in this case from the Object Ids list box. Double click on the EN_CHANGE message. Click OK. Selection of EN_CHANGE message is shown in figure6.12

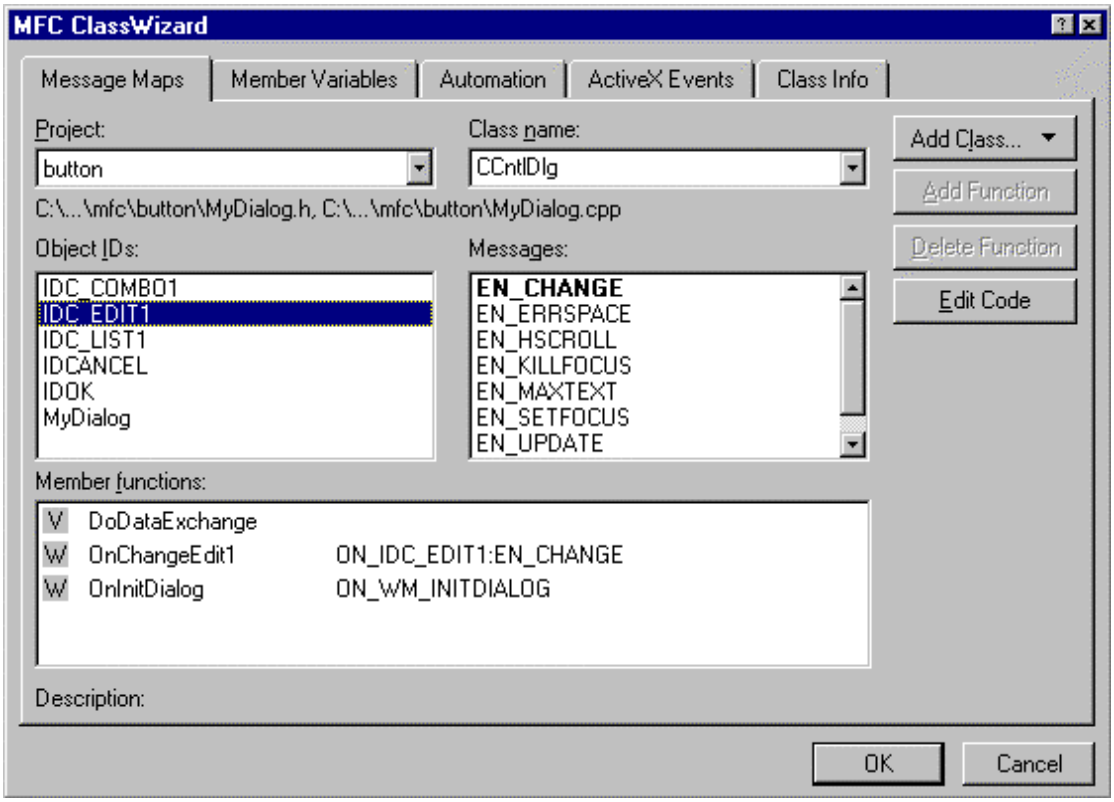


FIG6.12

The code for the OnChangeEdit1() funtion is as follows.

```
void MyDialog::OnChangeEdit1()
{
        UpdateData(TRUE);
        GetParent()->SetWindowText(m_edit);
}
```

Calling the UpdateData(TRUE) member function will retrieve the data from the dialog controls to the member variables. Thus the member variable m_edit will have the string entered in the control.
Next the GetParent()->SetWindowText(m_edit); statement is used to set this string as title of the dialog's Parent window which is the main Window. GetParent() returns a pointer to the parent window. EN_CHANGE message is a notification message which windows sends whenever the contents of the edit control is changed.

6.8.3. Handling the Dialog Box in the Window Class

The Control1 application displays the dialog box in response to its Dialog command, which is handled by the OnDialog() message response function. In that function, the program first creates the dialog box object and then initialises the dialog object's variables. It then calls DoModal() to display it. Whatever is done in OnDialog function is same as what is done in the Dialog application earlier in the chapter.

The dialog box object is automatically deleted when it goes out of scope, taking all its data with it, which is why you must copy whatever data you need from the dialog box.

7. USING BIMAPS

The term bitmap is frequently misunderstood by new Windows programmers. For most people, a bitmap is a picture that can be displayed on the screen. While this definition is generally true, there are actually two main types of bitmaps used with Windows applications. The first, called device-independent bitmaps (DIBs), are found in picture files with a .BMP file extension and are the type of bitmap with which most people are familiar.

The second type of bitmap, called a device-dependent bitmap (DDB), resides only in the computer's memory and is usually not a picture but rather some sort of image that a Windows application needs to create its display. In this chapter, you learn about DDBs and how they're used to build an application's display.

7.1. INTRODUCING DEVICE-DEPENDENT BITMAPS

As I just said, DIBs are the picture files that most people think of as bitmaps. DIBs are device independent because their file includes the color information needed to reproduce the picture on other devices. DDBs, on the other hand, don't include color tables. Instead, these types of images usually are created directly in the computer's memory and disappear when the application that created them terminates.

Because of their nature, DDBs are more utilitarian than DIBs. That is, the Windows programmer uses DDBs as tools for creating an application's display rather than displaying DDBs simply as pictures. Think of a Windows paint application. Often, a paint application enables the user to copy and paste a portion of the display. The user might, for example, copy the image of a tree and paste it down in various places on the display to create a forest. The tree image is a bitmap in the computer's memory. It's unlikely that the tree image will ever be saved to disk. On the other hand, the entire forest picture probably will be saved to disk as a DIB.

Another popular way to use DDBs (hereafter called bitmaps) is to use them to represent an application's entire display area in memory. The application makes changes to its display by drawing on the bitmap in memory and then copying the bitmap to the application's window. Using a bitmap in this way, an application can quickly update its display whenever it needs to without having to redraw the data from scratch. In the following section, you'll create a Windows application that handles its display in exactly this way.

7.2. CREATING THE BITMAP APPLICATION

As mentioned, one of the most common uses for bitmaps is to store an application's display. You'll now create the Bitmap application, which uses a series of 16 bitmaps to create Animation. The 16 bitmaps will give a motion effect when viewed successively, much like the frames of a motion picture. The files are Image01.bmp to Image16.bmp and they show a globe rotating. When the program begins, it rotates the globe using a timer, and the globe will continue to rotate until the program ends. The relevant portions of the source code are discussed in the chapter. There are 16 bitmap resources, each identified by an ID and associated with the file in which that bitmap resides.

To create the application, perform the following steps:

1. Create an application with CWinApp and CFrameWnd classes as described in the previous chapters.

2. Using the resource editor, import the 16 bitmap files (The pictures of different frames to be animated. For this Select the Insert, Resource command from Developer Studio's menu bar. The Insert Resource dialog box appears, as shown in Figure 7.1.

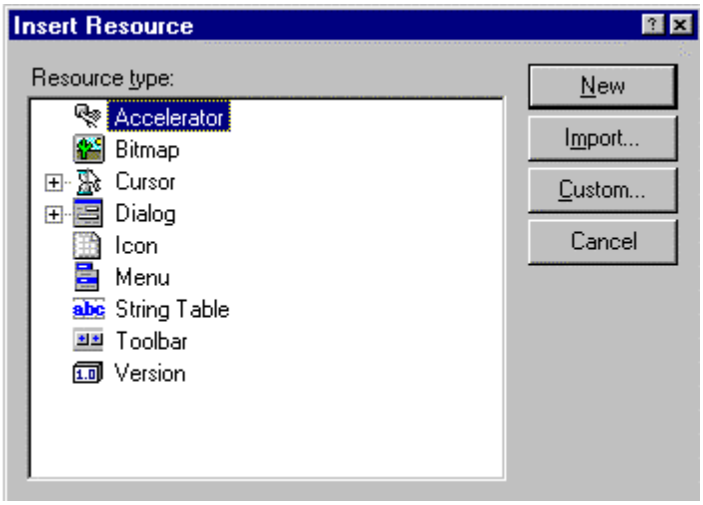


FIG. 7.1

Click on Bitmap and then the Import button.

4. Select the bitmap files to be imported and click Import Button.

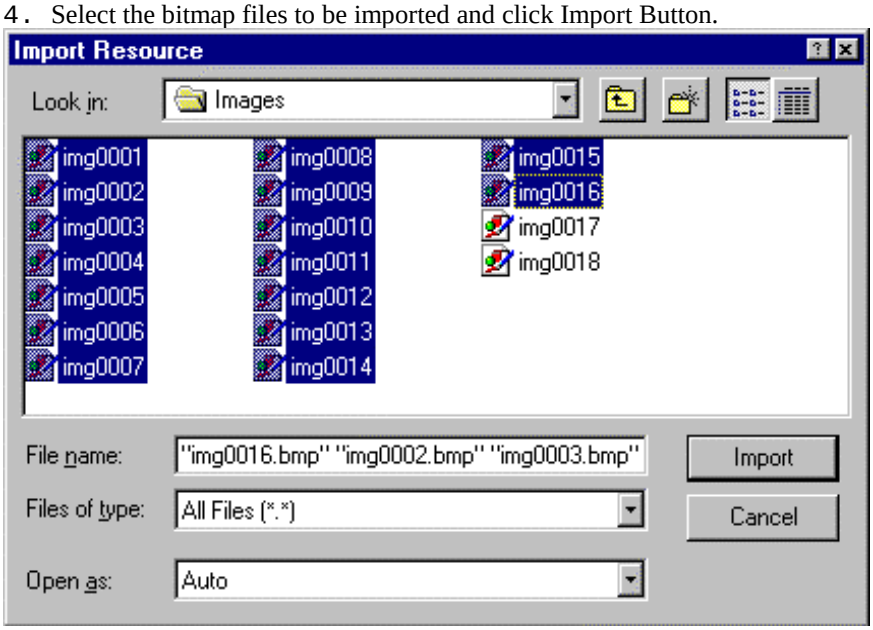


Fig 7.2

5. Add 3 private member variables to the Frame Window class as given below.

```
private:
        CBitmap* m_pBitmap[16];
        CDC* m_pMemDC[16];
        int m_nBmpNo;
```

5. Use ClassWizard to associate the WM_CREATE Windows message with the OnCreate() message response function in the Frame Window class, as shown in Figure 7.3

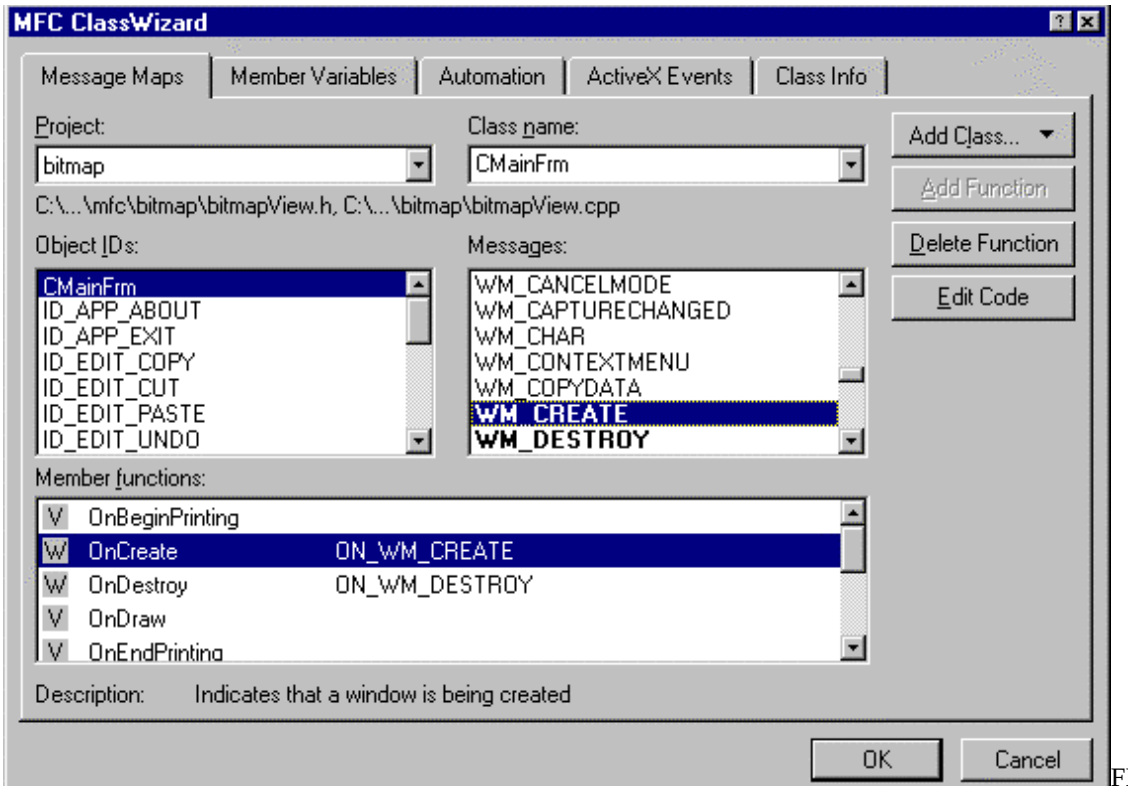


FIG. 7.3 Adding OnCreate () function to CMainFrm class (derived from CFrameWnd)

6. Add the lines shown in Listing 7.1 to the new OnCreate() function.

Listing 7.1 LST16_01.CPP Code for the OnCreate() Function

```
CClientDC dc(this);
for(int i=0;i<16;i++)
{
        m_pBitmap[i]=new CBitmap;
        m_pBitmap[i]->LoadBitmap(IDB_BITMAP1+i);

        m_pMemDC[i]=new CDC;
        m_pMemDC[i]->CreateCompatibleDC(&dc);
        m_pMemDC[i]->SelectObject(m_pBitmap[i]);
}
m_nBmpNo=0;
SetTimer(1,100,NULL);
```

7. Use ClassWizard to associate the WM_TIMER Windows message with the OnTimer() message response function in the Frame Window class, as shown in Figure 7.4.

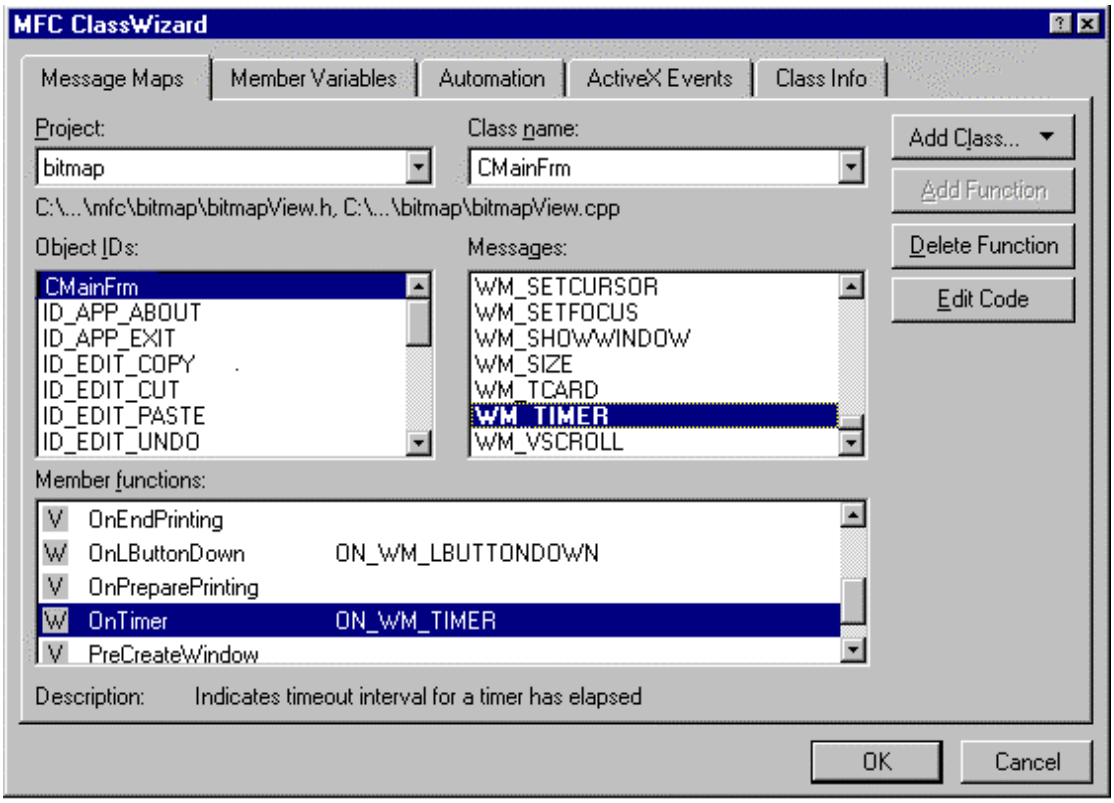


FIG. 7.4 Add the OnTimer() function to the Frame Window class.

8. Add the lines shown in Listing 7.2 to the new OnTimer() function TODO: Add your message handler code here and/or call default comment.
Listing 7.2 Code for the OnTimer() Function

```
CClientDC dc(this);
  m_nBmpNo++;
  if (m_nBmpNo>=16)
        m_nBmpNo=0;
  dc.BitBlt(10,10,100,100,m_pMemDC[m_nBmpNo],0,0,SRCCOPY);
CFrameWnd::OnTimer(nIDEvent);
```

9.Use ClassWizard to associate the WM_DESTROY Windows message with the OnDestroy() message response function in the Frame Window class.

10. Add the lines shown in Listing 7.3 to the new OnDestroy() function, before calling the CFrameWnd::OnDestroy() function.

Listing 7.3 Code for the OnDestroy() Function

```
KillTimer(1);
CFrameWnd::OnDestroy();
```

10. Add the following line to the CMainFrm class's destructor:

```
for (int i=0;i<16;i++)
        {
                delete m_pMemDC[i];
                delete m_pBitmap[i];
        }
```

This line deletes all the bitmaps, and Memory Device Contexts that are created in the OnCreate() function.

You've now completed the Bitmap application. To compile the application, select Developer Studio's Build, Build command. You can then run the program by selecting the Build, Execute command. When you do, the application's main window appears (Figure 7.5).

FIG. 7.5

7.3. EXPLORING THE BITMAP APPLICATION

Now that you've had a chance to experiment with the Bitmap application, you probably want to take a look under the hood to see how this bitmap stuff really works. In the following sections, you'll examine each of the key functions in the Bitmap application. When you're through, you should have a solid understanding of how to use bitmaps in your own applications.

7.3.1. Examining the OnCreate() Function

Whenever a new window is created, Windows sends the application a WM_CREATE message. Because the window has a valid handle when WM_CREATE is sent, the function OnCreate() (which responds to the WM_CREATE message in an MFC program) is a good place to do initialization that requires a valid window. In the Bitmap application, the program uses OnCreate() to load all the 16 bitmaps in memory.

For the program to display the bitmaps in the window's client area, the bitmap must be selected into a device context that's compatible with the window's device context. So the first task in OnCreate() is to create a CClientDC object for the window:

```
CClientDC dc(this);
```

The CClientDC constructor requires a pointer to the window for which the DC is being created. In this case, you just use the this pointer.

Then the program constructs a CBitmap object and loads the bitmap resource identified by the ID number IDB_BITMAP1+I from the application's executable file. The loaded bitmap is attached to the CBitmap object. This is done by the following code.

```
m_pBitmap[i] = new CBitmap;
m_pBitmap[i]->LoadBitmap(IDB_BITMAP1+i);
```

Next a DC that is compatible with the Screen DC is created in memory

```
m_pMemDC[i]=new CDC;
m_pMemDC[i]->CreateCompatibleDC(&dc);
```

m_pBitmap[16] and m_pMemDC[16] are pointers to Cbitmap and CDC respectively, declared as private variables of the FrameWindow class.

The memory DC can be an object of the CDC class. Calling the class's CreateCompatibleDC() function (which is an MFC version of a Windows API function of the same name) ensures that the memory DC is compatible with the client window's DC. The function's single argument is the address of the DC with which the new DC should be compatible.

When a memory device context is created, GDI automatically selects a 1-by-1 monochrome stock bitmap for it. GDI output functions can be used with a memory device context only if a bitmap has been created and selected into that context.

So the next step is to select a bitmap into the memory device context.

```
m_pMemDC.SelectObject(m_pBitmap);
```

The SelectObject() function, the CDC class's version of a Windows API function of the same name, takes care of this task. Its single argument is a pointer to the bitmap.
Thus in the OnCreate() function, 16 Device Contexts are created in memory, each having a bitmap selected into it.
The last statement in the OnCreate() function is a call to CWnd's SetTimer() member funtion. The SetTimer() function installs a system timer. A time-out value is specified, and every time a time-out occurs, the system posts a WM_TIMER message to the installing application's message queue or passes the message to an application-defined TimerProc callback function. The SetTimer() function takes 3 arguments.
SetTimer(UINT nIDEvent, UINT nElapse, lpfnTimer)
nIDEvent: Specifies a nonzero timer identifier.
NElapse: Specifies the time-out value, in milliseconds.
LpfnTimer: Specifies the address of the application-supplied TimerProc callback function that processes the WM_TIMER messages. If this parameter is NULL, the WM_TIMER messages are placed in the application's message queue and handled by the CWnd object.

In this application the following statement is used.
SetTimer(1,100,NULL);
A WM_TIMER message is sent to the message queue every 100ms. The OnTimer function is called every 100ms.

7.3.2. Examining the OnTimer() Function

As this function is called every 100ms, the code for displaying different frames of a moving picture can be included here. m_nBmpNo is a member variable to keep the number of the bitmap or memory device context that will be displayed. Each time the OnTimer function is called, m_nBmpNo is incremented, until it becomes 16. Once the 16^{th} bitmap is chose, m_nBmpNo is made 0.

```
m_nBmpNo++;
if (m_nBmpNo>=16)
        m_nBmpNo=0;
```

Finally, a call to the window DC's BitBlt() function transfers the bitmap's image from the memory DC to the window's client area:

```
dc.BitBlt(10,10,100,100,m_pMemDC[m_nBmpNo],0,0,SRCCOPY);
```

BitBlt(), which is the CDC class's version of a Windows API function, takes eight arguments: the X,Y coordinates of the bitmap's destination rectangle; the width and height of the destination rectangle; a pointer to the source DC (the one containing the bitmap); the X,Y coordinates of the rectangle in the bitmap to copy; and a flag indicating how the source rectangle should be combined with the destination rectangle. The flag SRCCOPY means that the bitmap will completely overwrite any other image in the window.
Thus 16 bitmaps are shown cyclically on the screen to have an animated effect.

8. USING APPWIZARD TO CREATE AN MFC PROGRAM

There are two approaches you can take when creating an MFC program. The first approach is to write all of the source code from scratch. This approach is the one which gives you a better understanding of the program.
The second approach you can take when creating an MFC program is to let Visual C++ AppWizard do a lot of the work for you. AppWizard can greatly speed program development by automatically creating a skeleton application that you can build upon to create your own specific application.

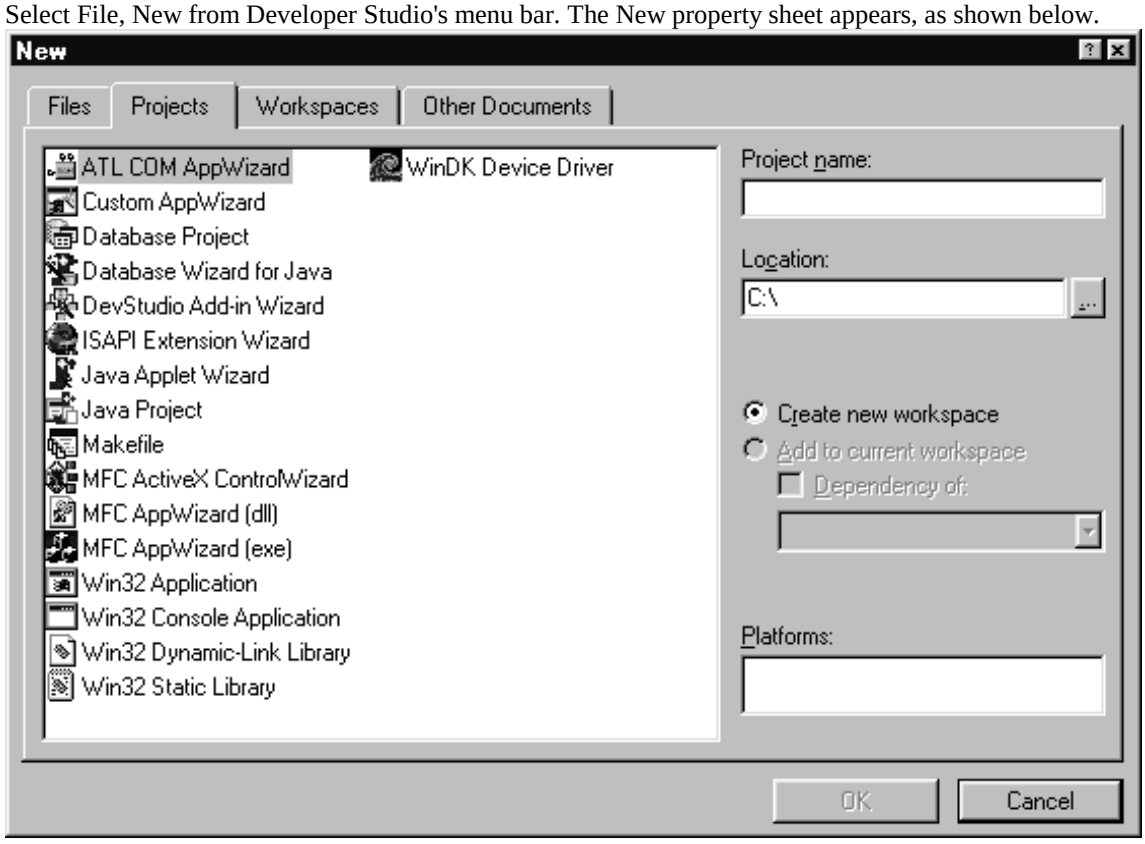
8.1. UNDERSTANDING WIZARDS

A wizard is simply an automated task, which helps the user to do things more easily. A Word for Windows user, for example, might use a letter wizard to start a letter. The wizard guides the user, step-by-step, through the process of creating the letter, requesting information in dialog boxes and finally creating a blank letter for the user to fill in with her text.
Visual C++ has its share of wizards, too. The two most important are AppWizard and ClassWizard. AppWizard guides you through the creation of a skeleton MFC application. After you have this skeleton application built, you add your own code to make the application do what you want it to do. ClassWizard, on the other hand, helps you manage the many classes, data members, and member functions in your program.

8.2. CREATING MFC PROGRAM WITH APPLICATION WIZARD

The steps to create your first MFC program are as follows

- Select File, New from Developer Studio's menu bar. The New property sheet appears, as shown below.



- Make sure `MFC AppWizard(exe)` is selected in the left hand pane. Then type **app1** into the Project Name box and the path of the folder into which you want the project files stored into the Location box. (If you like, you can use the Browse button to locate the folder into which you want the files stored.)
- Click the OK button. The MFC AppWizard - Step 1 dialog box appears. On this dialog box, you can choose to create an SDI, MDI, or dialog-based application. You can also choose a language.
- Leave the default options selected, and click the Next button. The MFC AppWizard - Step 2 of 6 dialog box appears. If you were creating a database application, you could now choose the type of database support you needed.

- Leave the database support set to None, and click the Next button. The MFC AppWizard - Step 3 of 6 dialog box appears. If you were creating an OLE application, you could select OLE support from the given options on this dialog box..
- Accept the default OLE options (None) by clicking the Next button. The MFC AppWizard - Step 4 of 6 dialog box appears. On this page of the wizard, you select the features you want to include in your application, including a toolbar, a status bar, and printing and messaging abilities.
- Leave the default features selected and click the Next button. The MFC AppWizard - Step 5 of 6 dialog box appears. Here, you can choose whether the AppWizard-generated source code will include comments and whether the MFC library should be loaded as a DLL (shared) or linked directly into your application's executable file (static).
- Accept the default settings by clicking the Next button. The MFC AppWizard - Step 6 of 6 dialog box appears, which lists the classes that AppWizard is about to create for you. Although you can change the classes' names, you'll usually leave them as they are..
- Leave the classes named as they are, and click the Finish button. The New Project Information dialog box appears. This dialog box displays a summary of the application you've decided to build.
- Click the OK button to create your new MFC application. AppWizard generates the source code for the application you selected.
- Click the Build button on the project toolbar or select Build, Build from the menu bar. Developer Studio compiles and links your new MFC application.

8.3. RUNNING YOUR FIRST MFC APPLICATION

After Developer Studio finishes compiling and linking the application, select Build, Execute from the menu bar to run the program. Except for the fact that your new application doesn't process any kind of data, it's surprisingly complete. You can, for example, create new document windows simply by clicking the toolbar's New button or by selecting File, New from the menu bar. If you select Help, About App1, the application's About dialog box appears. You can grab the toolbar with your mouse pointer and drag it somewhere else in the window, changing it from a toolbar to a toolbox. If you select the File, Open command, the Open dialog box appears, enabling you to choose a file to load. Although the file won't actually load, if you select a file, a new document window will appear with the file's name in the document window's title bar. Your new application even features cool stuff like tool tips those little hint boxes that appear when you leave the mouse pointer over a toolbar button for a second or two.

8.4. EXPLORING APPWIZARD'S FILES AND CLASSES

If you take a quick look into your project's folder, you'll discover that AppWizard created a whole slew of source code and data files for your application. What files appear in your project folder depends on the selections you made when you ran AppWizard. If you carefully followed the steps for creating the application, AppWizard generated a folder called RES that contains some of your project's resources, as well as a folder called Debug or Release (depending on whether you're creating a debugging or release version of the program) that contains all of the project's output files. It also generates 18 other files that make up the application's source code. The files AppWizard generates for the app1 project are the following:

File Name	Description
MAINFRM.CPP	The frame window class's implementation
MAINFRM.H	The frame window class's header file
README.TXT	Project description
RESOURCE.H	Resource header file
APP1DOC.CPP	The document class's implementation
APP1DOC.H	The document class's header file
APP1.CLW	ClassWizard data file
APP1.CPP	The application class's implementation
APP1.H	The application class's header file
APP1.DSW	The Project's workspace file
APP1.RC	The application's main resource file
APP1VIEW.CPP	The view class's implementation
APP1VIEW.H	The view class's header file
CHILDFRM.CPP	The MDI child window class's implementation

CHILDFRM.H	The MDI child window class's header file
STDAFX.CPP	Precompiled header file
STDAFX.H	Precompiled header file

The `CApp1App` class (declared and defined in the APP1.H and APP1.CPP files) represents the program's application object, as derived from MFC's `CWinApp`. Every Visual C++ MFC program must have an application object.

The `CMainFrame` class (MAINFRM.H and MAINFRM.CPP) represents the application's frame window. The frame window is the main window you see when you start the application.

The `CChildFrame` class, derived from `CMDIChildWnd`, represents the application's child document windows. One of these windows appears whenever you select the New or Open commands.

The `CApp1Doc` class (APP1DOC.H and APP1DOC.CPP) represents the application's document class and is derived from MFC's `CDocument`. In a Visual C++ program, the document class holds the data that makes up the current document, such as the text in a word processor document.

Finally, the `CApp1View` class (APP1VIEW.H and APP1VIEW.CPP) represents the application's view. In an AppWizard MFC program, the view is responsible for displaying the data stored in the document class. The view also enables the user to edit the document.

As you work with MFC applications, you'll better understand how you use the different classes to create the kind of application you want.

9. DOCUMENTS AND VIEWS

MFC's document/view architecture, is a way to separate an application's data from the way the user actually views and manipulates that data. Simply, the document object is responsible for storing, loading, and saving the data, whereas the view object enables the user to see the data on the screen and to edit that data as is appropriate to the application.

9.1. UNDERSTANDING THE DOCUMENT CLASS

When you looked over the various files generated by AppWizard in the previous chapter, you discovered a class called CApp1Doc, which was derived from MFC's CDocument class. In the app1 application, CApp1Doc is the class from which the application instantiates its document object, which is responsible for holding the application's document data. Because you didn't modify the AppWizard-generated files, the CApp1Doc class really holds no data. It's up to you to add storage for the document by adding data members to the CApp1Doc class.

To see how this works, look at listing below which shows the header file, AppWizard, created for the CApp1Doc class.

```
// App1Doc.h : interface of the CApp1Doc class
//
/////////////////////////////////////////////////////////////////////////////
//

#if
!defined(AFX_APP1DOC_H__90333ACE_3691_11D2_BD8A_00A0C95C4518__INCLUDED_)
#define AFX_APP1DOC_H__90333ACE_3691_11D2_BD8A_00A0C95C4518__INCLUDED_

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


class CApp1Doc : public CDocument
{
protected: // create from serialization only
      CApp1Doc();
      DECLARE_DYNCREATE(CApp1Doc)

// Attributes
public:

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CApp1Doc)
      public:
      virtual BOOL OnNewDocument();
      virtual void Serialize(CArchive& ar);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CApp1Doc();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

protected:
```

```
// Generated message map functions
protected:
      //{{AFX_MSG(CApp1Doc)
            // NOTE - the ClassWizard will add and remove member functions
here.
            //     DO NOT EDIT what you see in these blocks of generated
code !
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_APP1DOC_H__90333ACE_3691_11D2_BD8A_00A0C95C4518__INCLUDED_)
```

Near the top of the listing, you can see the class declaration's Attributes section, which is followed by the public keyword. This is where you declare the data members that will hold your application's data.

Notice also in the class's header file that the CApp1Doc class includes two virtual member functions called OnNewDocument() and Serialize(). MFC calls the OnNewDocument() function whenever the user chooses the File, New command. You can use this function to perform whatever initialization must be performed on your document's data. The Serialize() member function is where the document class loads and saves its data.

9.2. UNDERSTANDING THE VIEW CLASS

The view class is responsible for displaying, and enabling the user to modify, the data stored in the document object. To do this, the view object must be able to obtain a pointer to the document object. After obtaining this pointer, the view object can access the document's data members in order to display or modify them. If you look at the listing below, you can see how the CApp1View class, which you created in the previous chapter, obtains pointers to the document object.

```
#if !defined(APP1VIEW_H__99206D2E_7535_11D0_847F_444553540000__INCLUDED_)
#define APP1VIEW_H__99206D2E_7535_11D0_847F_444553540000__INCLUDED_
// app1View.h : interface of the CApp1View class
//
/////////////////////////////////////////////////////////////////////////////
//
class CApp1View : public CView
{
protected: // create from serialization only
    CApp1View();
    DECLARE_DYNCREATE(CApp1View)
// Attributes
public:
    CApp1Doc* GetDocument();
// Operations
public:
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CApp1View)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL
```

```
// Implementation
public:
    virtual ~CApp1View();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
    //{{AFX_MSG(CApp1View)
        // NOTE - the ClassWizard will add and remove member functions
here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG  // debug version in app1View.cpp
inline CApp1Doc* CApp1View::GetDocument()
   { return (CApp1Doc*)m_pDocument; }
#endif
/////////////////////////////////////////////////////////////////////////////
//
//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.
#endif                                                                  //
!defined(APP1VIEW_H__99206D2E_7535_11D0_847F_444553540000__INCLUDED)
```

Near the top of the listing, you can see the class's public attributes, where it declares the GetDocument() function as returning a pointer to a CApp1Doc object. Anywhere in the view class that you need to access the document's data, you can call GetDocument() to obtain a pointer to the document.

The view class, like the document class, also overrides a number of virtual functions from its base class. As you'll soon see, the OnDraw() function, which is the most important of these virtual functions, is where you paint your window's display. As for the other functions, MFC calls PreCreateWindow() before the window element (that is, the actual Windows window) is created and attached to the MFC window class, giving you a chance to modify the window's attributes (such as size and position). Finally, the OnPreparePrinting() function enables you to modify the Print dialog box before it's displayed to the user; the O nBeginPrinting() function gives you a chance to create GDI objects like pens and brushes that you need to handle the print job; and OnEndPrinting() is where you can destroy any objects you may have created in OnBeginPrinting().

9.3. AN EXAMPLE OF DOCUMENT-VIEW APPLICATION

The following example introduces capabilities that are integral to the document-view four-class architecture, mainly filing and prinitng. This is a Single Document Interface application which draws an ellipse on the screen. The co-ordinates of the ellipse can be modified through a dialog box. You can save the details of the ellipse in a file and can be later retrieved. You can also print the ellipse using the Print menu.

9.3.1. Creating AppWizard Project

We create an AppWizard starter application and then add our code to it in the appropriate places. Follow the steps given below while creating the starter application:

⇒ Choose the Single document application in Step1
⇒ Choose None in the next step
⇒ Again choose None in Step 3
⇒ In Step 4 choose only “Printing and print preview”, deselect all other choices and then choose the “Advanced” button.
    * In the “Advanced Options” query box, choose the tab “Document Template Strings” and fill in the “File extension” that you wish your files to have.
⇒ In Step 5 select your options for comments and MFC library

⇒ In the final step change the names of the classes as shown below

- Frame window class - CMainFrame
- Application class - CSdiApp
- View class - CSdiView
- Document class - CSdiDoc

Now build your application and execute it. AppWizard has provided a menu with items “File”, “Edit” and “Help”. We will be using the “File” menu item to save a document file, to initialize or load the application from a file, and preview and print the document. The menu item “Edit is used only for applications which needs edit facility. In this example we don’t need any edit facility and hence this menu item can be deleted. The menu item “Help” gives an “About” dialog box. This application cannot store any data or display anything on the view.

There are two main activities that need to take place when developing a document-view application; the first activity is to design the data that is specific to the application, and the other activity is to design the user interface for viewing and modifying this data. When designing the data, the emphasis is on the effectiveness of the data definition, the efficiency of the data manipulation routines and the data storage routines, whereas when designing the user interface, the emphasis is on ease of use and a pleasant and consistent appearance.

9.3.2. Designing the Application’s Data

The main object of our application is the ellipse object. The data of any application is kept in the document object. The ellipse object will be declared as an object data member of the document class; the ellipse object can be thought of as the document’s data. We will be filing and retrieving from the file all the data necessary to generate an ellipse object. We define a class CEllipse that contains all the data members and the member functions needed by an ellipse object.

The class of the document object, CSdiDoc is derived from the CDocument class. We place our CEllipse class in its own header and implementation files. We create the files Ellipse.h as the header file and Ellipse.cpp and implementation file for the CEllipse class. Add the Ellipse.cpp file to your project.

Add to the Ellipse.h file the following lines of code:

```
//Ellipse.h - declaration of CEllipse class
#include "stdafx.h"

class CEllipse : public CObject
{
      DECLARE_SERIAL(CEllipse)
public:

      void DrawEllipse(CDC* cdc);
      int Bottom;
      int Right;
      int Top;
      int Left;

      CEllipse();
      virtual ~CEllipse();

};
```

Notice that the class CEllipse is derived from CObject and the macro DECLARE_SERIAL(CEllipse) was included in the class. These are necessary conditions to serialize objects of this class. The process of saving objects to a file and restoring objects from a file is called serialization. Objects are serialized when they are filed or loaded from the file.

We must add the declaration of a CEllipse object, named m_Ellipse, to the class CSdiDoc in the file SdiDoc.h. This makes m_Ellipse an object data member of the class CSdiDoc. We will use the m_Ellipse object to access the data member function Serialize(). We also will use the m_Ellipse object to access the member variables and functions of its own class. Create an object of CEllipse class in CSdiDoc class as shown below.

```
// SdiDoc.h : interface of the CSdiDoc class
//
/////////////////////////////////////////////
class CSdiDoc : public CDocument
{
      . . . . . .
// Attributes
public:
      CEllipse m_Ellipse;
      . . . . . .


};
```

The implementation of the member function of the class CEllipse will be placed in the implementation file for the CEllipse class, namely Ellipse.cpp. Enter the code for the definition of the DrawEllipse() member function of the class CEllipse in the file Ellipse.cpp as shown below.

```
// Ellipse.cpp: implementation of the CEllipse class.
//
/////////////////////////////////////////////////////

. . . . . .

IMPLEMENT_SERIAL(CEllipse, CObject,1)

void CEllipse::DrawEllipse(CDC * cdc)
{
      cdc->Ellipse(Left, Top, Right, Bottom);
}
. . . . . .
```

The member variables of the class CEllipse will store the current data for the ellipse and the member function will draw the ellipse.

The implementation file includes the statement `IMPLEMENT_SERIAL(CEllipse, CObject,1).` This makes the class serializable. To summarize making a class serializable, the following three conditions are necessary.

1. The class must be derived directly or indirectly from CObject.
2. The class's declaration must contain the macro call DECLARE_SERIAL(class name).
3. The class's implementation file must contain the macro call IMPLEMENT_SERIAL(class name, baseclass name, version number). Version number encoded in the archive enables a de-serializing program to identify and handle data created by earlier program versions.

9.3.3. Designing the User Interface

The class that takes care of the user interaction with the document is the view class, CSdiView. It also takes care of drawing. When the application is first called, we want the ellipse with the initial values of its variables to be drawn in the client area. The member function that allows us to draw the ellipse in the client area of the view is OnDraw().

We access the member variables stored in the document class by using the pointer to the document class pDoc. We draw the ellipse in the client area by making a call to the DrawEllipse() member function of the CEllipse object, m_Ellipse. pDoc is the pointer to the document class and we access the m_Ellipse using the pDoc pointer. Modify the OnDraw() function as shown below to draw the ellipse.

```
// SdiView.cpp : implementation of the CSdiView class
//
. . . . .
. . . . .
```

```
/////////////////////////////////////////////////////////////////////////////
//
// CSdiView drawing


void CSdiView::OnDraw(CDC* pDC)
{
      CSdiDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      pDoc->m_Ellipse.DrawEllipse(pDC);
}
. . . . .
. . . . .
```

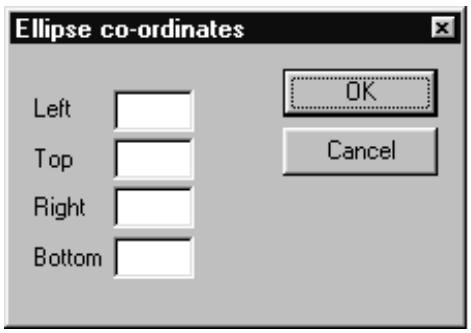
9.3.4. The OnNewDocument() Function

The function CDocument::OnNewDocument() should always be used to initialize data in a new document; this function is called automatically when the user selects “File” and then “New” from the menu. In an SDI application, the document object is being re-used each time a different file is opened. Thus when the users selects “File”, “New” we want to re-initialize the document’s contents. We do this in the function OnNewDocument(). AppWizard has already inserted this function into our code, so all we need to do is to customize it. This is done with the following code.

```
// SdiDoc.cpp : implementation of the CSdiDoc class
//

. . . . .
. . . . .

BOOL CSdiDoc::OnNewDocument()
{
      if (!CDocument::OnNewDocument())
            return FALSE;

      m_Ellipse.Left=10;
      m_Ellipse.Top=10;
      m_Ellipse.Right=100;
      m_Ellipse.Bottom=100;

      return TRUE;
}
. . . . .
. . . . .
```

In the above code, we initialize the four data members of the m_Ellipse object.

Now create a dialog box to change the co-ordinates of the ellipse as shown below.



Create a class CElipDlg for this dialog and add integer member variables m_nLeft, m_nTop, m_nRight and m_nBottom to store the co-ordinates of the ellipse.

Add a menu item “Dimension” to display the above dialog. Add a handler function OnDimension() for this menu item. The code for OnDimension() handler function is shown below.

```
// SdiView.cpp : implementation of the CSdiView class
//

. . . . .
. . . . .

void CSdiView::OnDimension()
{
      CSdiDoc* pDoc = GetDocument();
      CElipDlg edlg;

      edlg.m_nLeft=pDoc->m_Ellipse.Left;
      edlg.m_nTop=pDoc->m_Ellipse.Top;
      edlg.m_nRight=pDoc->m_Ellipse.Right;
      edlg.m_nBottom=pDoc->m_Ellipse.Bottom;

      if(edlg.DoModal()==IDCANCEL)
            return;

      pDoc->m_Ellipse.Left=edlg.m_nLeft;
      pDoc->m_Ellipse.Top=edlg.m_nTop;
      pDoc->m_Ellipse.Right=edlg.m_nRight;
      pDoc->m_Ellipse.Bottom=edlg.m_nBottom;

      Invalidate();
}
. . . . .
. . . . .
```

The user interface is now complete. Now you can draw ellipses of various sizes using the dialog box.

9.3.5. Filing the Document’s Data

So far, the data belonging to the m_Ellipse object will vanish as soon as the application is closed. We need a way to save this data in files. The Serialize() function is the means to achieve this.

Since the document data consists if just one object m_Ellipse of class CEllipse, we need to call the Serialize() member function of the class CEllipse. We need to add the declaration for the Serialize() function to the CEllipse class definiton as shown below.

```
// Ellipse.h: interface for the CEllipse class.
//
//////////////////////////////////////////////////////////////////////
. . . . .
. . . . .

class CEllipse : public CObject
{
      DECLARE_SERIAL(CEllipse)
public:
      virtual void Serialize(CArchive& ar);
      . . . . .
      . . . . .

};
```

Modify the Serialize() function of the class CSdiDoc() which AppWizard has already introduced into the SdiDoc.cpp file as follows:

```
// SdiDoc.cpp : implementation of the CSdiDoc class
```

```
//
. . . . .
. . . . .

void CSdiDoc::Serialize(CArchive& ar)
{
      m_Ellipse.Serialize(ar);
}

. . . . .
```

9.3.6. Serialization and CArchive

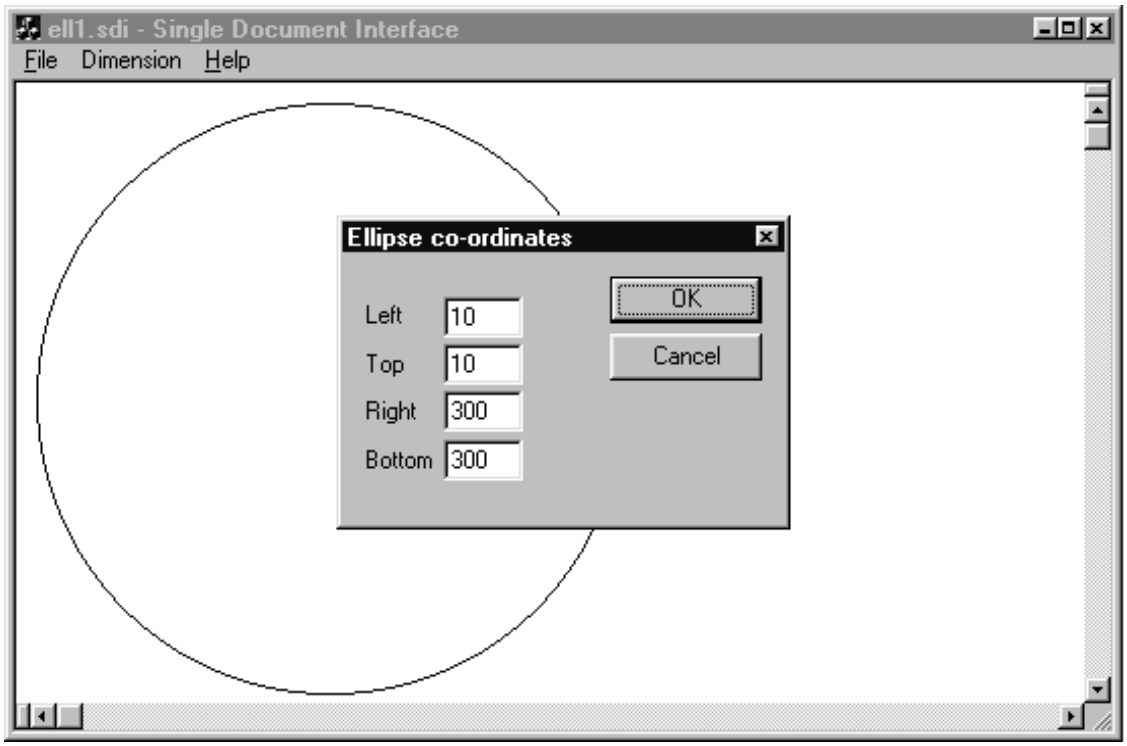
The process of saving objects to a file and restoring objects from a file is called serialization. All the data associated with a serializable object is sequentially read from or written to a single disk file.

All you have to do in the Serialize() function is to load data from or store data in an archive object. The disk file on which the data will be stored is represented by an object of class CFile; between the Serialize() function and the CFile object is an archive object of class CArchive. The application's CArchive object is named "ar". The CArchive::IsStoring() member function tells us whether the archive is currently being used for storing to a file or loading from a file. The CArchive class has overloaded insertion operator (<<) for writing the data to the file and extraction operator (>>) for reading the data from the file.

In our example, we want to store the ellipse data to a file and retrieve it. To do this, add the following implementation of the Serialize() function to the implementation file "ellipse.cpp"

```
void CEllipse::Serialize(CArchive & ar)
{
      if(ar.IsStoring())
      {
            ar<<Left;
            ar<<Top;
            ar<<Right;
            ar<<Bottom;
      }
      else
      {
            ar>>Left;
            ar>>Top;
            ar>>Right;
            ar>>Bottom;
      }
}
```

Now, rebuild the application and run it. Experiment with the use of the File menu items: "Open", "Save" and "Save As". The following figure shows the application window with the dialog box for changing the dimensions of the ellipse.



9.4. PRINTING THE VIEW

MFC provides built-in priniting support for the view. The printing support that MFC provides to your application is device-independent. This means that the same code written for the OnDraw() function can be used to draw on the screen or to print to the printer. When you ask to print a document using the standard "File" | "Print" command, MFC calls the OnDraw() member function with a special device context that is aware of the current printer and knows how to translate your screen display into appropriate printed output. The function OnDraw() is called by the framework to render an image of the document. The framework calls this function to perform screen display, printing and print preview, passing a different device context in each case. If you are printing, OnDraw() is called by another CView class member, OnPrint(), with a printer device context. If you are displaying, OnPaint() calls OnDraw(), and the device context is of class CPaintDC. In the print preview mode, the CDC object is linked to another device context object of class CPreviewDC, but that linkage is transparent to the user.

9.4.1. The function OnPrepareDC()

When you print your document by choosing "File" | "Print" from the menu you will get a very small ellipse. This is because our drawing units are display pixels and when we put these on a laser with a resolution of 300 or 600 dpi it makes a very small figure indeed. To draw graphics, we need to provide scale factors, which means that we need to change the mapping mode from the default value, which is, MM_TEXT.

To change the mapping mode for printing, we need to change the mapping mode in the CView::OnPrepareDC() function. The OnPrepareDC() function is called for OnPaint(), for OnDraw() and for OnPrint(). The OnPrepareDC() function is called in OnPaint() immediately before the call to OnDraw(). The mapping mode is set before painting the view. If you are prinitng, the same OnPrepareDC() function is called, this time immediately before the application framework calls OnPrint(). The mapping mode is set before the printing of the page. We want to change the mapping mode only if we are printing, so we will call the CDC::IsPrinting()function and if we are printing, we will then override the function and change the mapping mode.

The function OnPrepareDC() has two parameters. The first is the device context. The second parameter is a pointer to a CPrintInfo object, which is valid only if OnPrepareDC() is being called prior to printing. Test for this condition by calling the CDC function , IsPrinting().

For our application, we need to override the CView's virtual function OnPrepareDC(). We will first insert the function's prototype in the SdiView.h file, since AppWizard did not provide it. We will add the function prototype as shown below:

```
// SdiView.h : interface of the CSdiView class
//
/////////////////////////////////////////////////////////////////////////
. . . . . .
. . . . . .

class CSdiView : public CView
{
      . . . . . .
      . . . . . .
public:

      virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
      . .  . . . .
      . . . .  . .
};
```

We decide which mapping mode to use in our overriding function, OnPrepareDC().

9.4.2. Mapping Modes

Graphics and text output are passed co-ordinates in the device context that specify where they should be painted. We have been using the default system (MM_TEXT) where the units are called device units and we have one screen pixel, or printer dot, per unit. For the screen, device units are the number of pixels measured from the upper left corner of a window area. For a printer, device units are the number of dots measured from the upper left corner of the printed page. When we translate screen pixels to printer dots, we have a very tiny picture.

Mapping modes permit the user to define logical units, and then the operating system figures out how to map it to your selected output device. Logical units can be in inches or millimeters, in which case the operating system figures out how big an inch or a millimeter is, in screen pixels or printer dots, and plots or prints the output there. A mapping mode is known as fixed scale when its logical units are specified in inches or millimeters. Table below gives the mapping modes and their meaning:

Mapping Mode	**Meaning**
MM_TEXT	Default mapping mode; each logical unit equals one device unit(screen pixel or printer dot). X increases to the right, Y increases downward.
MM_LOENGLISH	Each logical unit is 0.01 inch (low resolution).
MM_HIENGLISH	Each logical unit is 0.001 inch (high resolution).
MM_LOMETRIC	Each logical unit is 0.1 millimeter (low resolution).
MM_HIMETRIC	Each logical unit is 0.01 millimeter (high resolution).
MM_TWIPS	Used with text fonts. Each logical unit is 1/20 point, or 1/1440 of an inch.
MM_ISOTROPIC	Arbitrary scaling of the axes, but the X and Y scaling must be the same.
MM_ANISOTROPIC	Either axis can have any scaling factor. This is the most flexible mode.

Two mapping modes, MM_ISOTROPIC and MM_ANISOTROPIC, allow you to change the scale factor as well as the origin. When you change the scaling factor, you can think of it as stretching or compressing the co-ordinate system. 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.
With the scalable mapping modes, you use the two functions CDC::SetWindowExt(X,Y) and CDC::SetViewportExt(X,Y) together to set the scaling of the co-ordinate system. For example, to create a co-ordinate system where each logical unit is six times the default device unit co-ordinates, use the code as shown below:

```
// sdiView.cpp : implementation of the CSdiView class
//
. . . . .
```

```
. . . . .

void CSdiView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
      if(pDC->IsPrinting())
      {
            pDC->SetMapMode(MM_ISOTROPIC);
            pDC->SetViewportExt(6,6);
            pDC->SetWindowExt(1,1);
      }
      CView::OnPrepareDC(pDC, pInfo);
}
. . . . .
. . . . .
```

The function SetWindowExt(X,Y) and SetViewportExt(X,Y) work together to the set the scale, based on the window's scale. SetWindowExt(X, Y) defines the window's co-ordinates. SetViewportExt(X,Y) defines the printer's co-ordinates. The scale factor is the ratio of the values provided in these two functions. To stretch the output by 500%, uses 5 to 1 ratio. To compress the output by 66 percent, use 2 to 3 scaling.

9.4.3. Functions for Prinitng

The CView class defines several member functions that are called by the framework during printing. By overriding these functions in your view class, you provide the connections between the framework's printing logic and your view class's printing logic. The table below lists these member functions in the order in which they are called by the framework:

Function	**Reason for Overriding**
OnPreparePrinitng()	To insert values in the Print dialog box, especially the length of the document.
OnBeginPrinting()	To allocate fonts or other GDI resources.
OnPrepareDC()	To adjust attributes of the device context for a given page, or to do print-time pagination.
OnPrint()	To print a given page.
OnEndPrinting()	To deallocate GDI resources.

AppWizard has automatically inserted the stubbed-out OnPreparePrinting(), OnBeginPrinting() and OnEndPrinting() functions in your code.

The framework stores much of the information about a print job in a CPrintInfo object. Several of the values in CPrintInfo pertain to pagination and are accessible through the member function or member variables as shown in the following table:

Member Variable/Function Name	**Reference to Page Number**
GetMinPage()/SetMinPage()	First page of document
GetMaxPage()/SetMaxPage()	Last page of document
GetFromPage()	First page to be printed
GetToPage()	Last page to be printed
m_nCurPage	Page currently being printed

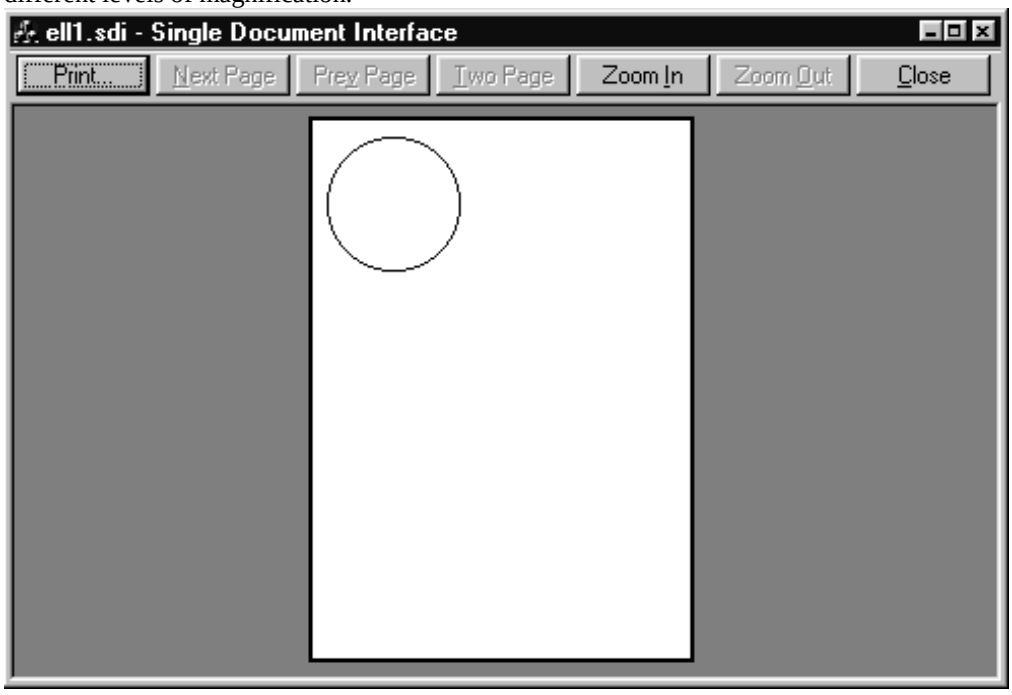
A maximum page number must be inserted in the print dialog box. If the maximum page is unspecified, it spools through endless pages. Since it is too easy to forget to enter the maximum page, this problem can be fixed by giving the framework the maximum page to enter into the print dialog box. To do this use the overriding OnPreparePrinting() function, and within that function set the maximum page. This is done with the following code:

```
. . . . .
/////////////////////////////////////////////////////
// CSdiView printing

BOOL CSdiView::OnPreparePrinting(CPrintInfo* pInfo)
{
```

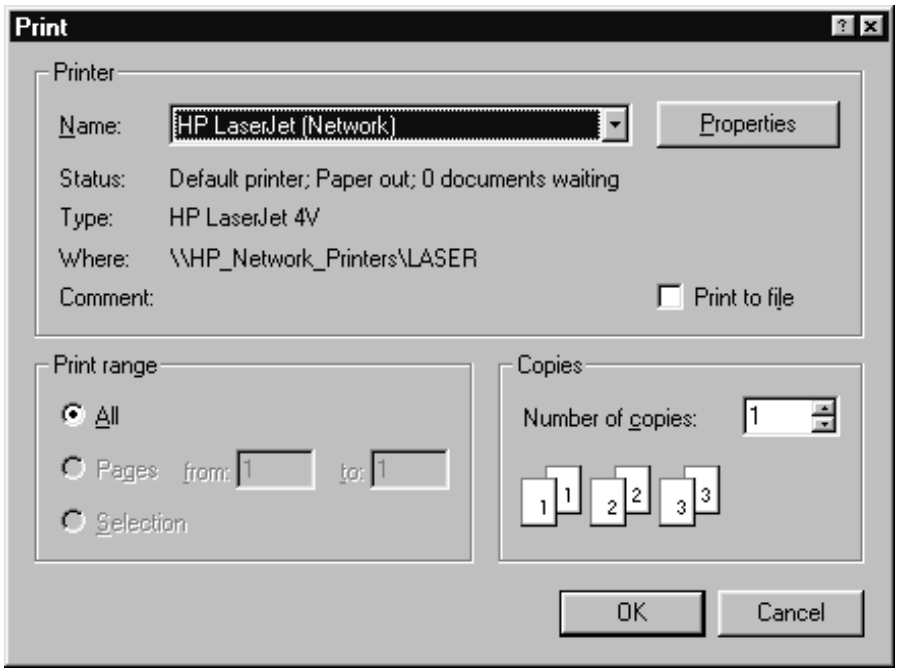
```
      pInfo->SetMaxPage(1);
      // default preparation
      return DoPreparePrinting(pInfo);
}
. . . . .
```

9.4.4. Print Preview and Print Setup

When you choose “File” | “Print Preview”, you are accessing a lot of already-coded capability, which the MFC framework supplies. AppWizard has inserted all the necessary code into your application to access the print preview capability. Print preview shows a reduced image of either one or two pages of a document as it would appear when printed on the currently selected printer. I provides a standard user interface, as shown below, for navigating between pages, toggling between one- and two-page viewing and zooming the display in and out to different levels of magnification.



When you choose “File” | “Print Setup”, the MFC framework automatically provides a dialog box as shown below from which you can make selections. You do not have to supply any of your own code for this. AppWizard has inserted all the code necessary for your application to access this capability.



9.4.5. Program Listing

The following listings shows the code for the above program:

```
// Ellipse.h: interface for the CEllipse class.
//
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_ELLIPSE_H__29F5FDF5_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_ELLIPSE_H__29F5FDF5_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

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

#include "stdafx.h"

class CEllipse : public CObject
{
      DECLARE_SERIAL(CEllipse)
public:
      virtual void Serialize(CArchive& ar);

      void DrawEllipse(CDC* cdc);
      int Bottom;
      int Right;
      int Top;
      int Left;
};

#endif                                                                  //
!defined(AFX_ELLIPSE_H__29F5FDF5_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)

----------------------------------------------------------------------

// Ellipse.cpp: implementation of the CEllipse class.
//
//////////////////////////////////////////////////////////////////////
```

```
#include "Ellipse.h"

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


void CEllipse::DrawEllipse(CDC * cdc)
{
	cdc->Ellipse(Left, Top, Right, Bottom);
}

void CEllipse::Serialize(CArchive & ar)
{
	if(ar.IsStoring())
	{
		ar<<Left;
		ar<<Top;
		ar<<Right;
		ar<<Bottom;
	}
	else
	{
		ar>>Left;
		ar>>Top;
		ar>>Right;
		ar>>Bottom;
	}
}

-------------------------------------------------------------------------

#if
!defined(AFX_ELIPDLG_H__29F5FDF6_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_ELIPDLG_H__29F5FDF6_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// ElipDlg.h : header file
//

/////////////////////////////////////////////////////////////////////////////
//
// CElipDlg dialog

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

// Dialog Data
	//{{AFX_DATA(CElipDlg)
	enum { IDD = IDD_DIALOG1 };
	int		m_nLeft;
	int		m_nBottom;
	int		m_nRight;
	int		m_nTop;
	//}}AFX_DATA


// Overrides
	// ClassWizard generated virtual function overrides
```

```
	//{{AFX_VIRTUAL(CElipDlg)
	protected:
	virtual  void  DoDataExchange(CDataExchange*  pDX);        //  DDX/DDV
support
	//}}AFX_VIRTUAL

// Implementation
protected:

	// Generated message map functions
	//{{AFX_MSG(CElipDlg)
		// NOTE: the ClassWizard will add member functions here
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
//   Microsoft   Developer   Studio   will   insert   additional   declarations
immediately before the previous line.

#endif                                                                      //
!defined(AFX_ELIPDLG_H__29F5FDF6_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)

-----------------------------------------------------------------------

// ElipDlg.cpp : implementation file
//

#include "stdafx.h"
#include "sdi.h"
#include "ElipDlg.h"

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

/////////////////////////////////////////////////////////////////////////////
//
// CElipDlg dialog


CElipDlg::CElipDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CElipDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CElipDlg)
	m_nLeft = 0;
	m_nBottom = 0;
	m_nRight = 0;
	m_nTop = 0;
	//}}AFX_DATA_INIT
}


void CElipDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CElipDlg)
	DDX_Text(pDX, IDC_LEFT, m_nLeft);
	DDX_Text(pDX, IDC_BOTTOM, m_nBottom);
	DDX_Text(pDX, IDC_RIGHT, m_nRight);
	DDX_Text(pDX, IDC_TOP, m_nTop);
	//}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CElipDlg, CDialog)
      //{{AFX_MSG_MAP(CElipDlg)
            // NOTE: the ClassWizard will add message map macros here
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

-----------------------------------------------------------------------

// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////
//

#if
!defined(AFX_MAINFRM_H__29F5FDE9_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_MAINFRM_H__29F5FDE9_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

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

class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
      CMainFrame();
      DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CMainFrame)
      public:
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      protected:
      virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
                                  pContext);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CMainFrame();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
      //{{AFX_MSG(CMainFrame)
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.
```

```
#endif                                                                  //
!defined(AFX_MAINFRM_H__29F5FDE9_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)

-----------------------------------------------------------------------

// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "sdi.h"

#include "MainFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
//
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
      //{{AFX_MSG_MAP(CMainFrame)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
}

CMainFrame::~CMainFrame()
{
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
      return CFrameWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
//
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
      CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
      CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
//
```

```
// CMainFrame message handlers


BOOL   CMainFrame::OnCreateClient(LPCREATESTRUCT   lpcs,   CCreateContext*
pContext)
{
      return CFrameWnd::OnCreateClient(lpcs, pContext);
}

-------------------------------------------------------------------------

// sdi.h : main header file for the SDI application
//

#if !defined(AFX_SDI_H__29F5FDE5_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_SDI_H__29F5FDE5_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

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

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
//
// CSdiApp:
// See sdi.cpp for the implementation of this class
//

class CSdiApp : public CWinApp
{
public:
      CSdiApp();

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CSdiApp)
      public:
      virtual BOOL InitInstance();
      //}}AFX_VIRTUAL

// Implementation

      //{{AFX_MSG(CSdiApp)
      afx_msg void OnAppAbout();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
//   Microsoft   Developer   Studio   will   insert   additional   declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_SDI_H__29F5FDE5_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)

-------------------------------------------------------------------------
```

```
// sdi.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "sdi.h"
#include "MainFrm.h"
#include "sdiDoc.h"
#include "sdiView.h"

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

/////////////////////////////////////////////////////////////////////////////
//
// CSdiApp

BEGIN_MESSAGE_MAP(CSdiApp, CWinApp)
      //{{AFX_MSG_MAP(CSdiApp)
      ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
      //}}AFX_MSG_MAP
      // Standard file based document commands
      ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
      ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
      // Standard print setup command
      ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CSdiApp construction

CSdiApp::CSdiApp()
{
}

/////////////////////////////////////////////////////////////////////////////
//
// The one and only CSdiApp object

CSdiApp theApp;

/////////////////////////////////////////////////////////////////////////////
//
// CSdiApp initialization

BOOL CSdiApp::InitInstance()
{
      // Standard initialization

      // Change the registry key under which our settings are stored.
      SetRegistryKey(_T("Local AppWizard-Generated Applications"));

      LoadStdProfileSettings();//Load standard INI file options

      // Register document templates

      CSingleDocTemplate* pDocTemplate;
      pDocTemplate = new CSingleDocTemplate(
            IDR_MAINFRAME,
            RUNTIME_CLASS(CSdiDoc),
            RUNTIME_CLASS(CMainFrame),       // main SDI frame window
            RUNTIME_CLASS(CSdiView));
      AddDocTemplate(pDocTemplate);
```

```
	// Enable DDE Execute open
	EnableShellOpen();//To open file from fil manager
	RegisterShellFileTypes(TRUE);

	// Parse command line for standard shell commands, DDE, file open
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	// Enable drag/drop open
	m_pMainWnd->DragAcceptFiles();

	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
//
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	//}}AFX_DATA

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

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
		// No message handlers
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

// App command to run the dialog
void CSdiApp::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}

-----------------------------------------------------------------------

// sdiDoc.h : interface of the CSdiDoc class
//
/////////////////////////////////////////////////////////////////////////
//

#if !defined(AFX_SDIDOC_H__29F5FDEB_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_SDIDOC_H__29F5FDEB_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

#include "Ellipse.h"    // Added by ClassView
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000


class CSdiDoc : public CDocument
{
protected: // create from serialization only
	CSdiDoc();
	DECLARE_DYNCREATE(CSdiDoc)

// Attributes
public:
	CEllipse m_Ellipse;
// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CSdiDoc)
	public:
	virtual BOOL OnNewDocument();
	virtual void Serialize(CArchive& ar);
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CSdiDoc();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CSdiDoc)
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
```

```
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_SDIDOC_H__29F5FDEB_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)

-----------------------------------------------------------------------

// sdiDoc.cpp : implementation of the CSdiDoc class
//

#include "stdafx.h"
#include "sdi.h"
#include "sdiDoc.h"

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

/////////////////////////////////////////////////////////////////////////////
//
// CSdiDoc

IMPLEMENT_DYNCREATE(CSdiDoc, CDocument)

BEGIN_MESSAGE_MAP(CSdiDoc, CDocument)
      //{{AFX_MSG_MAP(CSdiDoc)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CSdiDoc construction/destruction

CSdiDoc::CSdiDoc()
{
}

CSdiDoc::~CSdiDoc()
{
}

BOOL CSdiDoc::OnNewDocument()
{
      if (!CDocument::OnNewDocument())
            return FALSE;

      m_Ellipse.Left=10;
      m_Ellipse.Top=10;
      m_Ellipse.Right=100;
      m_Ellipse.Bottom=100;

      return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
//
// CSdiDoc serialization

void CSdiDoc::Serialize(CArchive& ar)
{
      m_Ellipse.Serialize(ar);
}
```

```
/////////////////////////////////////////////////////////////////////////////
//
// CSdiDoc diagnostics

#ifdef _DEBUG
void CSdiDoc::AssertValid() const
{
      CDocument::AssertValid();
}

void CSdiDoc::Dump(CDumpContext& dc) const
{
      CDocument::Dump(dc);
}
#endif //_DEBUG

-------------------------------------------------------------------------

// sdiView.h : interface of the CSdiView class
//
/////////////////////////////////////////////////////////////////////////////
//

#if
!defined(AFX_SDIVIEW_H__29F5FDED_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_SDIVIEW_H__29F5FDED_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

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

#include "ElipDlg.h"

class CSdiView : public CView
{
protected: // create from serialization only
      CSdiView();
      DECLARE_DYNCREATE(CSdiView)

// Attributes
public:
      CSdiDoc* GetDocument();

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CSdiView)
      public:
      virtual void OnDraw(CDC* pDC);  // overridden to draw this view
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      virtual void OnPrepareDC(CDC* pDC, CPrintInfo* pInfo = NULL);
      protected:
      virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
      virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
      virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CSdiView();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected:

// Generated message map functions
protected:
      //{{AFX_MSG(CSdiView)
      afx_msg void OnDimension();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in sdiView.cpp
inline CSdiDoc* CSdiView::GetDocument()
   { return (CSdiDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_SDIVIEW_H__29F5FDED_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)

-------------------------------------------------------------------------

// sdiView.cpp : implementation of the CSdiView class
//

#include "stdafx.h"
#include "sdi.h"

#include "sdiDoc.h"
#include "sdiView.h"

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

/////////////////////////////////////////////////////////////////////////////
//
// CSdiView

IMPLEMENT_DYNCREATE(CSdiView, CView)

BEGIN_MESSAGE_MAP(CSdiView, CView)
      //{{AFX_MSG_MAP(CSdiView)
      ON_COMMAND(ID_DIMENSION, OnDimension)
      //}}AFX_MSG_MAP
      // Standard printing commands
      ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
      ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
      ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CSdiView construction/destruction

CSdiView::CSdiView()
{
}
```

```
CSdiView::~CSdiView()
{
}

BOOL CSdiView::PreCreateWindow(CREATESTRUCT& cs)
{
	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
//
// CSdiView drawing

void CSdiView::OnDraw(CDC* pDC)
{
	CSdiDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDoc->m_Ellipse.DrawEllipse(pDC);
}

/////////////////////////////////////////////////////////////////////////////
//
// CSdiView printing

BOOL CSdiView::OnPreparePrinting(CPrintInfo* pInfo)
{
	pInfo->SetMaxPage(1);
	// default preparation
	return DoPreparePrinting(pInfo);
}

void CSdiView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
}

void CSdiView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
}

/////////////////////////////////////////////////////////////////////////////
//
// CSdiView diagnostics

#ifdef _DEBUG
void CSdiView::AssertValid() const
{
	CView::AssertValid();
}

void CSdiView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

CSdiDoc* CSdiView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CSdiDoc)));
	return (CSdiDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
//
// CSdiView message handlers
```

```
void CSdiView::OnDimension()
{
	CSdiDoc* pDoc = GetDocument();
	CElipDlg edlg;

	edlg.m_nLeft=pDoc->m_Ellipse.Left;
	edlg.m_nTop=pDoc->m_Ellipse.Top;
	edlg.m_nRight=pDoc->m_Ellipse.Right;
	edlg.m_nBottom=pDoc->m_Ellipse.Bottom;

	if(edlg.DoModal()==IDCANCEL)
		return;

	pDoc->m_Ellipse.Left=edlg.m_nLeft;
	pDoc->m_Ellipse.Top=edlg.m_nTop;
	pDoc->m_Ellipse.Right=edlg.m_nRight;
	pDoc->m_Ellipse.Bottom=edlg.m_nBottom;

	Invalidate();

	pDoc->UpdateAllViews(NULL);
}

void CSdiView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
	if(pDC->IsPrinting())
	{
		pDC->SetMapMode(MM_ISOTROPIC);
		pDC->SetViewportExt(6,6);
		pDC->SetWindowExt(1,1);
	}
	CView::OnPrepareDC(pDC, pInfo);
}

-----------------------------------------------------------------------

// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__29F5FDE7_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
#define AFX_STDAFX_H__29F5FDE7_9E04_11D1_BD88_0000C039DEC8__INCLUDED_

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

#define VC_EXTRALEAN		//  Exclude  rarely-used  stuff  from  Windows
headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>			//  MFC  support  for  Windows  Common
Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_STDAFX_H__29F5FDE7_9E04_11D1_BD88_0000C039DEC8__INCLUDED_)
```

```
---------------------------------------------------------------------------

// stdafx.cpp : source file that includes just the standard includes
//    sdi.pch will be the pre-compiled header
//    stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

10. EXERCISE

1. Write a Windows SDK program and load various predefined icons in Windows
2. Write a Windows SDK program and load various predefined cursors in Windows
3. Write a Windows SDK program and try various background colours using stock objects
4. Write a Windows SDK program and try various window styles (WS_XXXXX) and sizes
5. Show the window using different ShowWindow options (SW_XXXXX)
6. Write a Windows SDK program which handles the up, down and double click of the left and right buttons of the mouse. Display message boxes for each message.
7. Write a Windows SDK program which handles the WM_PAINT message
8. Write a Windows SDK program which displays all types of message boxes. Provide menu item for displaying each type of message box. Also display various icons in the message boxes
9. Write a Windows SDK program which asks the confirmation from the user before it closes the window. (Hint - use WM_CLOSE message)
10. Write a program which draws a rectangle by tracking the mouse movement. The left-top co-ordinate of the rectangle should be the point where the mouse button is clicked. Keep the left mouse button pressed and drag the mouse for varying the rectangle size. The right-bottom co-ordinate should be the point at which the right button is released.
11. Enhance the above program to draw other graphical shapes using the mouse.
12. Write a program which draws various graphical shapes with different pen colours, styles and thickness. Menu should be provided to select the shape of the figure, pen styles, pen thickness, pen colour, brush style, brush colour.
13. Write a program to display text with various fonts. Menu should be provided to select thickness, size, escapement, underline, italics and other styles of the font.
14. Write a program to accept data through Edit Box, List Box and Combo Box controls and display it on the window.
15. Write a program to store the details of 10 computers. Enter the details of computer through dialog box. Also display the details using a dialog box through another menu. Provide *Next* and *Previous* buttons in the dialog box to display the details of the next and previous computer. Details of the computer include name, RAM, speed, hard disk capacity, CD-ROM speed, CPU, cache memory, monitor resolution, monitor size, graphics card name, network card name. Use edit boxes for name and RAM, list boxes for monitor size and resolution and combo boxes for other details.
16. Write a program to draw various graphical shapes. Provide dialog boxes to select pen color, pen size, pen style, brush colour and brush style. Provide edit box to enter pen size. Use colour dialog to specify pen colour and brush colour. Use Combo box to select brush style and list box to select pen style.