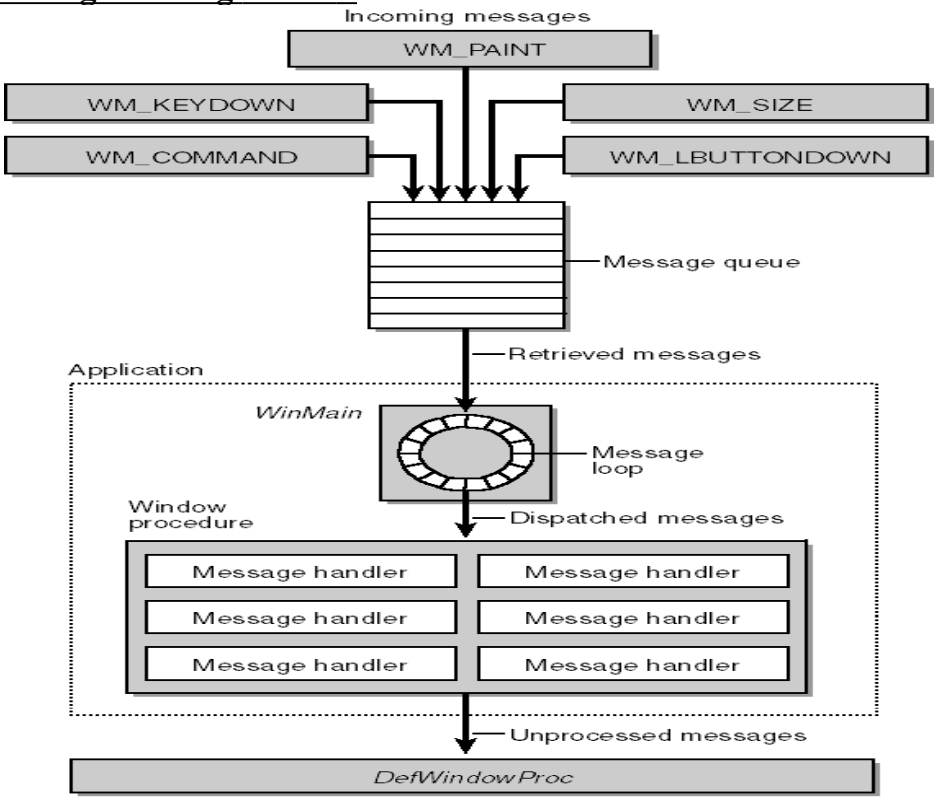
Chapter-1:

Microsoft Windows and Visual C++

Q: Explain windows programming model?

Ans:

The Windows Programming Model :



- **Message Processing:**

i. When you write an MS-DOS-based application in C, the only absolute requirement is a function named *main*. The operating system calls *main* when the user runs the program, and from that point on, you can use any programming structure you want.

ii. When the Windows operating system launches a program, it calls the program's *WinMain* function. Somewhere your application must have *WinMain*, which performs some specific tasks. Its most important task is creating the application's main window, which must have its own code to process messages that Windows sends it.

iii. An essential difference between a program written for MS-DOS and a program written for Windows is that an MS-DOS-based program calls the operating system to get user input, but a Windows-based program processes user input via messages from the operating system.

iv. Most messages in Windows are strictly defined and apply to all programs. For example, a WM_CREATE message is sent when a window is being created, a WM_LBUTTONDOWN message is sent when the user presses the left mouse button, a WM_CHAR message is sent when the user types a character, and a WM_CLOSE message is sent when the user closes a window. All messages have two 32-bit parameters that convey information such as cursor coordinates, key code, and so forth.

v. Windows sends WM_COMMAND messages to the appropriate window in response to user menu choices, dialog button clicks, and so on. Command message parameters vary depending on the window's menu layout.You can define your own messages, which your program can send to any window on the desktop.

- **The Windows Graphics Device Interface**

i. Many MS-DOS programs wrote directly to the video memory and the printer port. The disadvantage of this technique was the need to supply driver software for every video board and every printer model.

ii. Windows introduced a layer of abstraction called the Graphics Device Interface (GDI). Windows provides the video and printer drivers, so your program doesn't need to know the type of video board and printer attached to the system. Instead of addressing the hardware, your program calls GDI functions that reference a data structure called a devicecontext.

iii. Windows maps the device context structure to a physical device and issues the appropriate input/output instructions. The GDI is almost as fast as direct video access, and it allows different applications written for Windows to share the display.

- **Resource-Based Programming**

i. To do data-driven programming in MS-DOS, you must either code the data as initialization constants or provide separate data files for your program to read.

ii. When you program for Windows, you store data in a resource file using a number of established formats. The linker combines this binary resource file with the C++ compiler's output to generate an executable program.

iii. Resource files can include bitmaps, icons, menu definitions, dialog box layouts, and strings. They can even include custom resource formats that you define.

iv. You use a text editor to edit a program, but you generally use wysiwyg (what you see is what you get) tools to edit resources. Microsoft Visual C++ 6.0 has graphics resource editors for all standard resource formats.

- **Memory Management**

With each new version of Windows, memory management gets easier. If you've heard horror stories about locking memory handles, thunks, and burgermasters, don't worry. That's all in the past. Today you simply allocate the memory you need, and Windows takes care of the details. .

- **Dynamic Link Libraries**

i. In the MS-DOS environment, all of a program's object modules are statically linked during the build process. Windows allows dynamic linking, which means that specially constructed libraries can be loaded and linked at runtime.

ii. Multiple applications can share dynamic link libraries (DLLs), which saves memory and disk space. Dynamic linking increases program modularity because you can compile and test DLLs separately.

iii. Designers originally created DLLs for use with the C language, and C++ has added some complications. The MFC developers succeeded in combining all the application framework classes into a few ready-built DLLs.

iv. This means that you can statically or dynamically link the application framework classes into your application. In addition, you can create your own extensionDLLs that build on the MFC DLLs.

The Win32 Application Programming Interface

i. Early Windows programmers wrote applications in C for the Win16 application programming interface (API). Today, if you want to write 32-bit applications, you must use the new Win32 API, either directly or indirectly.

ii. Most Win16 functions have Win32 equivalents, but many of the parameters are different—16-bit parameters are often replaced with 32-bit parameters.

iii. The Win32 API offers many new functions, including functions for disk I/O, which was formerly handled by MS-DOS calls.

iv. With the 16-bit versions of Visual C++, MFC programmers were largely insulated from these API differences because they wrote to the MFC standard, which was designed to work with either Win16 or Win32 underneath.

Q: What are the components of VC++? Discuss briefly.
Or With the help of diagram describe VC++ application build process.
Ans:
The Visual C++ Components

i. Microsoft Visual C++ is two complete Windows application development systems in one product.
ii. Visual C++ also includes the ActiveX Template Library (ATL), which you can use to develop ActiveX controls for the Internet.
iii. A quick run-through of the Visual C++ components will help you get your bearings before you zero in on the application framework. Figure 1-1 shows an overview of the Visual C++ application build process.
iv.

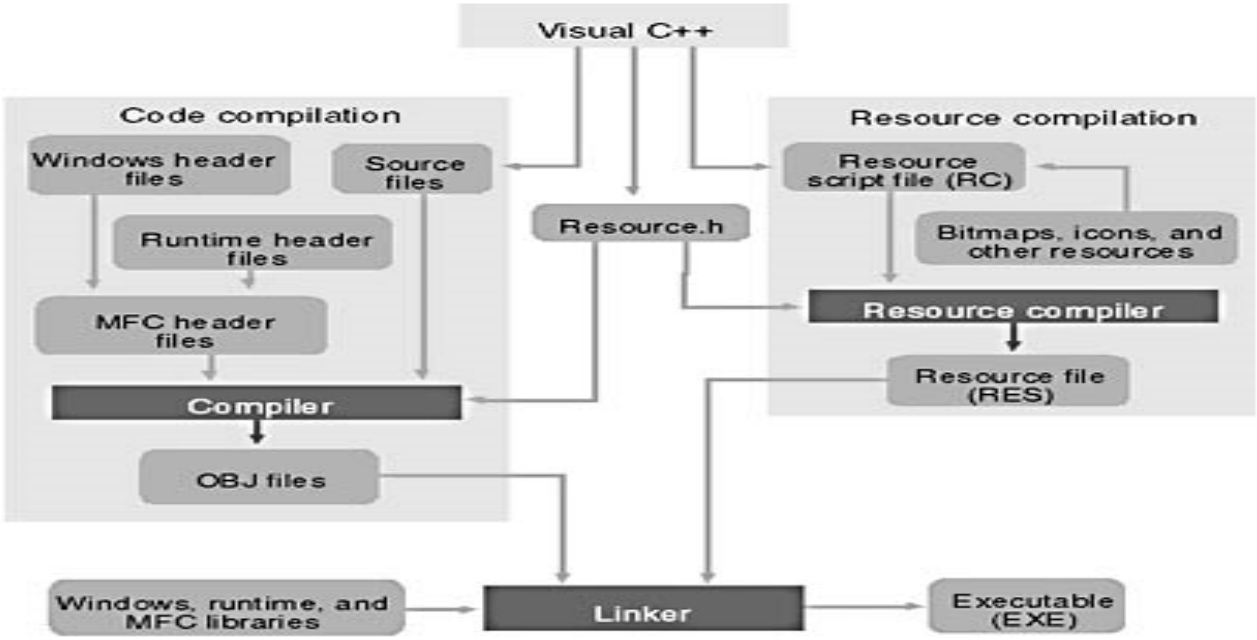


Figure 1-1. *The Visual C++ application build process.*

- **Microsoft Visual C++ 6.0 and the Build Process**

i. Figure 1-2 shows Visual C++ 6.0 in action. If you've used earlier versions of Visual C++ or another vendor's IDE, you already understand how Visual C++ 6.0 operates. But if you're new to IDEs, you'll need to know what a project is.
ii. A project is a collection of interrelated source files that are compiled and linked to make up an executable Windows-based program or a DLL.
iii. Source files for each project are generally stored in a separate subdirectory. A project depends on many files outside the project subdirectory too, such as include files and library files.
iv. Experienced programmers are familiar with makefiles. A makefile stores compiler and linker options and expresses all the interrelationships among source files. (A source code file needs specific include files, an executable file requires certain object modules and libraries, and so forth.)
v. A makeprogram reads the makefile and then invokes the compiler, assembler, resource compiler, and linker to produce the final output, which is generally an executable file. The make program uses built-in inference rules that tell it, for example, to invoke the compiler to generate an OBJ file from a specified CPP file.
vi. In a Visual C++ 6.0 project, there is no makefile (with an MAK extension) unless you tell the system to export one. A text-format projectfile (with a DSP extension) serves the same purpose.
vii. A separate text-format workspacefile (with a DSW extension) has an entry for each project in the workspace. It's possible to have multiple projects in a workspace. Visual C++ creates some intermediate files too.

The following table lists the files that Visual C++ generates in the workspace.

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

- **The Resource Editors—Workspace ResourceView**

i. When you click on the ResourceView tab in the Visual C++ Workspace window, you can select a resource for editing.
ii. The main window hosts a resourceeditor appropriate for the resource type.
iii. The window can also host a wysiwyg editor for menus and a powerful graphical editor for dialog boxes, and it includes tools for editing icons, bitmaps, and strings.
iv. Each project usually has one text-format resource script (RC) file that describes the project's menu, dialog, string, and accelerator resources.

- **The C/C++ Compiler**

The Visual C++ compiler can process both C source code and C++ source code. It determines the language by looking at the source code's filename extension.

- **The Source Code Editor**

Visual C++ 6.0 includes a sophisticated source code editor that supports many features such as dynamic syntax coloring, auto-tabbing, keyboard bindings for a variety of popular editors.In Visual C++ 6.0, an exciting new feature named AutoComplete has been added.

- **The Resource Compiler**

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

- **The Linker**

The linker reads the OBJ and RES files produced by the C/C++ compiler and the resource compiler, and it accesses LIB files for MFC code, runtime library code, and Windows code. It then writes the project's EXE file. An incremental link option minimizes the execution time when only minor changes have been made to the source files.

- **The Debugger**

i. The Visual C++ debugger has been steadily improving, but it doesn't actually fix the bugs yet. The debugger works closely with Visual C++ to ensure that breakpoints are saved on disk.
ii. If you position the cursor on a simple variable, the debugger shows you its value in a little window. To debug a program, you must build the program with the compiler and linker options set to generate debugging information.
iii. Visual C++ 6.0 adds a new twist to debugging with the Edit And Continue feature. Edit And Continue lets you debug an application, change the application, and then continue debugging with the new code.

- **AppWizard**

i. AppWizard is a code generator that creates a w++orking skeleton of a Windows application with features, class names, and source code filenames that you specify through dialog boxes.

ii. AppWizard code is minimalist code; the functionality is inside the application framework base classes.

iii. AppWizard gets you started quickly with a new application.

- **ClassWizard**

ClassWizard is a program (implemented as a DLL) that's accessible from Visual C++'s View menu. ClassWizard writes the prototypes, the function bodies, and (if necessary) the code to link the Windows message to the function. ClassWizard can update class code that you write, so you avoid the maintenance problems common to ordinary code generators.

- **The Source Browser**

The Visual C++ Source Browser (the browser, for short) lets you examine (and edit) an application from the class or function viewpoint instead of from the file viewpoint. The browser has the following viewing modes:

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

- **Online Help**

In Visual C++ 6.0, the help system has been moved to a separate application named the MSDN Library Viewer. This help system is based on HTML. Each topic is covered in an individual HTML document; then all are combined into indexed files. Visual C++ 6.0 allows you to access help in four ways:

- **By book**—When you choose Contents from Visual C++'s Help menu, the MSDN Library application switches to a contents view. Here Visual Studio, Visual C++, Win32 SDK documentation, and more is organized hierarchically by books and chapters.
- **By topic**—When you choose Search from Visual C++'s Help menu, it automatically opens the MSDN Library Viewer. You can then select the Index tab, type a keyword, and see the topics and articles included for that keyword.
- **By word**—When you choose Search from Visual C++'s Help menu, it invokes the MSDN Library with the Search tab active. With this tab active, you can type a combination of words to view articles that contain those words.
- **F1 help**—This is the programmer's best friend. Just move the cursor inside a function, macro, or class name, and then press the F1 key and the help system goes to work. If the name is found in several places—in the MFC and Win32 help files, for example—you choose the help topic you want from a list window.

- **Windows Diagnostic Tools**

Visual C++ 6.0 contains a number of useful diagnostic tools. SPY++ gives you a tree view of your system's processes, threads, and windows. It also lets you view messages and examine the windows of running applications.

- **The Gallery**

The Visual C++ Components and Controls Gallery lets you share software components among different projects. The Gallery manages three types of modules:

- **ActiveX controls**—When you install an ActiveX control (OCX—formerly OLE control), an entry is made in the Windows Registry. All registered ActiveX controls appear in the Gallery's window, so you can select them in any project.
- **C++ source modules**—When you write a new class, you can add the code to the Gallery. The code can then be selected and copied into other projects. You can also add resources to the Gallery.
- **Visual C++ components**—The Gallery can contain tools that let you add features to your project. Such a tool could insert new classes, functions, data members, and resources into an existing project. Some component modules are supplied by Microsoft (Idle time processing, Palette support, and Splash screen, for example) as part of Visual C++. Others will be supplied by third-party soft-ware firms.

Chapter-2:

The Microsoft Foundation Class Library Application Framework

Q: What is an application framework? Why Use the Application Framework?

Ans:

Application Framework is an integrated collection of object oriented software components that offers all that's needed for a generic application. It is a superset of class library. An ordinary library is an isolated set of classes designed to incorporated into any program but an application framework defines the structure of program itself.

- **The MFC library is the C++ Microsoft Windows API:**

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

- **Application framework applications use a standard structure:**

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

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

- **Application framework applications are small and fast:**

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

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

- **The Visual C++ tools reduce coding drudgery:**

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

- **The MFC library application framework is feature rich:**

The MFC library version 1.0 classes, introduced with Microsoft C/C++ version 7.0, included the following features:

- A C++ interface to the Windows API
- General-purpose (non-Windows-specific) classes, including:
  - Collection classes for lists, arrays, and maps
  - A useful and efficient string class
  - Time, time span, and date classes
  - File access classes for operating system independence
  - Support for systematic object storage and retrieval to and from disk
- A "common root object" class hierarchy
- Streamlined Multiple Document Interface (MDI) application support
- Some support for OLE version 1.0

The MFC library version 2.0 classes (in Visual C++ version 1.0) picked up where the version 1.0 classes left off by supporting many user interface features that are found in current Windows-based applications, plus they introduced the application framework architecture. Here's a summary of the important new features:

- Full support for File Open, Save, and Save As menu items and the most recently used file list
- Print preview and printer support
- Support for scrolling windows and splitter windows
- Support for toolbars and status bars
- Access to Visual Basic controls
- Support for context-sensitive help
- Support for automatic processing of data entered in a dialog box
- An improved interface to OLE version 1.0
- DLL support

The MFC library version 2.5 classes (in Visual C++ version 1.5) contributed the following:

- Open Database Connectivity (ODBC) support that allows your application to access and update data stored in many popular databases such as Microsoft Access, FoxPro, and Microsoft SQL Server
- An interface to OLE version 2.01, with support for in-place editing, linking, drag and drop, and OLE Automation

Visual C++ version 2.0 was the first 32-bit version of the product. It included support for Microsoft Windows NT version 3.5. It also contained MFC version 3.0, which had the following new features:

- Tab dialog (property sheet) support (which was also added to Visual C++ version 1.51, included on the same CD-ROM)
- Docking control bars that were implemented within MFC
- Support for thin-frame windows
- A separate Control Development Kit (CDK) for building 16-bit and 32-bit OLE controls, although no OLE control container support was provided

A subscription release, Visual C++ 2.1 with MFC 3.1, added the following:

- Support for the new Microsoft Windows 95 (beta) common controls
- A new ODBC Level 2 driver integrated with the Access Jet database engine

- Winsock classes for TCP/IP data communication

Microsoft decided to skip Visual C++ version 3.0 and proceeded directly to 4.0 in order to synchronize the product version with the MFC version. MFC 4.0 contains these additional features:

- New OLE-based Data Access Objects (DAO) classes for use with the Jet engine
- Use of the Windows 95 docking control bars instead of the MFC control bars
- Full support for the common controls in the released version of Windows 95, with new tree view and rich-edit view classes
- New classes for thread synchronization
- OLE control container support

Visual C++ 4.2 was an important subscription release that included MFC version 4.2. The following new features were included:

- WinInet classes
- ActiveX Documents server classes
- ActiveX synchronous and asynchronous moniker classes
- Enhanced MFC ActiveX Control classes, with features such as windowless activation, optimized drawing code, and so forth
- Improved MFC ODBC support, including recordset bulk fetches and data transfer without binding

Visual C++ 5.0 included MFC version 4.21, which fixed some 4.2 bugs. Visual C++ 5.0 introduced some worthwhile features of its own as well:

- A redesigned IDE, Developer Studio 97, which included an HTML-based online help system and integration with other languages, including Java
- The Active Template Library (ATL) for efficient ActiveX control construction for the Internet
- C++ language support for COM (Component Object Model) client programs with the new *#import* statement for type libraries.

The latest edition of Visual C++, 6.0, includes MFC 6.0. (Notice that the versions are now synchronized again.) Many of the features in MFC 6.0 enable the developer to support the new Microsoft Active Platform, including the following:

- MFC classes that encapsulate the new Windows common controls introduced as part of Internet Explorer 4.0
- Support for Dynamic HTML, which allows the MFC programmer to create applications that can dynamically manipulate and generate HTML pages
- Active Document Containment, which allows MFC-based applications to contain Active Documents
- OLE DB Consumers and Providers Template support and Active Data Objects (ADO) data binding, which help database developers who use MFC or ATL

An Application Framework Example:

Following is the source code for the header and implementation files for our MYAPP application. The classes *CMyApp* and *CMyFrame* are each derived from MFC library base classes. First, here is the MyApp.h header file for the MYAPP application:

```
// application class
class CMyApp : public CWinApp
{
public:
   virtual BOOL InitInstance();
};
// frame window class
class CMyFrame : public CFrameWnd
{
```

```
public:
CMyFrame();
protected:
   // "afx_msg" indicates that the next two functions are part
   //  of the MFC library message dispatch system
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnPaint();
   DECLARE_MESSAGE_MAP()
};
```

And here is the MyApp.cpp implementation file for the MYAPP application:

```
#include <afxwin.h> // MFC library header file declares base classes
#include "myapp.h"
CMyApptheApp; // the one and only CMyApp object
BOOL CMyApp::InitInstance()
{
m_pMainWnd = new CMyFrame();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
   return TRUE;
}
BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd)
   ON_WM_LBUTTONDOWN()
   ON_WM_PAINT()
END_MESSAGE_MAP()
CMyFrame::CMyFrame()
{
   Create(NULL, "MYAPP Application");
}
void CMyFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
   TRACE("Entering CMyFrame::OnLButtonDown - %lx, %d, %d\n",
        (long) nFlags, point.x, point.y);
}
void CMyFrame::OnPaint()
{
CPaintDC dc(this);
dc.TextOut(0, 0, "Hello, world!");
}
```

Here are some of the program elements:

The *WinMain* function—Remember that Windows requires your application to have a *WinMain* function. You don't see *WinMain* here because it's hidden inside the application framework.

The *CMyApp* class—An object of class *CMyApp* represents an application. The program defines a single global *CMyApp* object, *theApp*. The *CWinApp* base class determines most of *theApp*'s behavior.

Application startup—When the user starts the application, Windows calls the application framework's built-in *WinMain* function, and *WinMain* looks for your globally constructed application object of a class derived from *CWinApp*. Don't forget that in a C++ program global objects are constructed before the main program is executed.

The *CMyApp::InitInstance* member function—When the *WinMain* function finds the application object, it calls the virtual *InitInstance* member function, which makes the calls needed to construct and

display the application's main frame window. You must override *InitInstance* in your derived application class because the *CWinApp* base class doesn't know what kind of main frame window you want.

The *CWinApp::Run* member function—The *Run* function is hidden in the base class, but it dispatches the application's messages to its windows, thus keeping the application running. *WinMain* calls *Run* after it calls *InitInstance*.

The *CMyFrame* class—An object of class *CMyFrame* represents the application's main frame window. When the constructor calls the *Create* member function of the base class *CFrameWnd*, Windows creates the actual window structure and the application framework links it to the C++ object. The *ShowWindow* and *UpdateWindow* functions, also member functions of the base class, must be called in order to display the window.

The *CMyFrame::OnLButtonDown* function—This function is a sneak preview of the MFC library's message-handling capability. We've elected to "map" the left mouse button down event to a *CMyFrame* member function. For the time being, accept that this function gets called when the user presses the left mouse button. The function invokes the MFC library *TRACE* macro to display a message in the debugging window.

The *CMyFrame::OnPaint* function—The application framework calls this important mapped member function of class *CMyFrame* every time it's necessary to repaint the window: at the start of the program, when the user resizes the window, and when all or part of the window is newly exposed. The *CPaintDC* statement relates to the Graphics Device Interface (GDI) and is explained in later chapters. The *TextOut* function displays "Hello, world!"

Application shutdown—The user shuts down the application by closing the main frame window. This action initiates a sequence of events, which ends with the destruction of the *CMyFrame* object, the exit from *Run*, the exit from *WinMain*, and the destruction of the *CMyApp* object.

Look at the code example again. This time try to get the big picture. Most of the application's functionality is in the MFC library base classes *CWinApp* and *CFrameWnd*. In writing MYAPP, we've followed a few simple structure rules and we've written key functions in our derived classes. C++ lets us "borrow" a lot of code without copying it. Think of it as a partnership between us and the application framework. The application framework provided the structure, and we provided the code that made the application unique.

Now you're beginning to see why the application framework is more than just a class library. Not only does the application framework define the application structure but it also encompasses more than C++ base classes. You've already seen the hidden*WinMain* function at work. Other elements support message processing, diagnostics, DLLs, and so forth.

Chapter-3:

Q: Write short note on Documents and Views?

Ans:

- **Documents and Views**

i. In simple terms, the document-view architecture separates data from the user's view of the data. One benefit is multiple views of the same data.
ii. The document base class code interacts with the File Open and File Save menu items; the derived document class does the actual reading and writing of the document object's data.
iii. The view base class represents a window contained inside a frame window; the derived view class interacts with its associated document class and does the application's display and printer I/O.
iv. The derived view class and its base classes handle Windows messages. The MFC library orchestrates all interactions among documents, views, frame windows, and the application object, mostly through virtual functions.

- **Views:**

i. From a user standpoint a view is an ordinary window that the user can size, move and close in the same way as any other windows based application.
ii. From a programmer perspective a view is C++ object of color class derived from MFC library CView class.
iii. The view class code is divided into two source modules.
   a. The header file (.h),
   b. The implementation file (.cpp).

Q: Explain the difference between SDI and MDI.

An SDI application has, from the user's point of view, only one window. If the application depends on disk-file "documents," only one document can be loaded at a time. The original Windows Notepad is an example of an SDI application.

An MDI application has multiple childwindows, each of which corresponds to an individual document. Microsoft Word is a good example of an MDI application.

The *OnDraw* Member Function

OnDraw is a virtual member function of the *CView* class that the application framework calls every time the view window needs to be repainted. A window needs to be repainted if the user resizes the window or reveals a previously hidden part of the window, or if the application changes the window's data. If the user resizes the window or reveals a hidden area, the application framework calls *OnDraw*, but if a function in your program changes the data, it must inform Windows of the change by calling the view's inherited *Invalidate* (or *InvalidateRect*) member function

Even though you can draw inside a window at any time, it's recommended that you let window changes accumulate and then process them all together in the *OnDraw* function. That way your program can respond both to program-generated events and to Windows-generated events such as size changes.

The Windows Device Context

In the MFC library, the device context is a C++ object of class *CDC* that is passed (by pointer) as a parameter to *OnDraw*. After you have the device context pointer, you can call the many *CDC* member functions that do the work of drawing.

Q: Explain win-32 debug target v/s win-32 release target?

Ans:

If you open the drop-down list on the Build toolbar, you'll notice two items: Win32 Debug and Win32 Release. (The Build toolbar is not present by default, but you can choose Customize from the Tools menu to display it.) These items are targets that represent distinct sets of build options. When AppWizard generates a project, it creates two default targets with different settings. These settings are summarized in the following table.

Option	Release Build	Debug Build
Source code debugging	Disabled	Enabled for both compiler and linker
MFC diagnostic macros	Disabled (NDEBUG defined)	Enabled (_DEBUG defined)
Library linkage	MFC Release library	MFC Debug libraries
Compiler optimization	Speed optimization (not available in Learning Edition)	No optimization (faster compile)

Chapter-4:

Basic Event Handling, Mapping Modes, and a Scrolling View

Q: Write short note on Mapping modes?

Ans:

Mapping Modes:

There are three types of mapping modes.

1. The *MM_TEXT* Mapping Mode:

i. *MM_TEXT* appears to be no mapping mode at all, but rather another name for device coordinates.

ii. In *MM_TEXT*, coordinates map to pixels, values of *x* increase as you move right, and values of *y* increase as you move down, but you're allowed to change the origin through calls to the *CDC* functions *SetViewportOrg* and *SetWindowOrg*.

iii. Here's some code that sets the window origin to (100, 100) in logical coordinate space and then draws a 200-by-200-pixel square offset by (100, 100). (An illustration of the output is shown in Figure 4-2.) The logical point (100, 100) maps to the device point (0, 0). A scrolling window uses this kind of transformation.

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

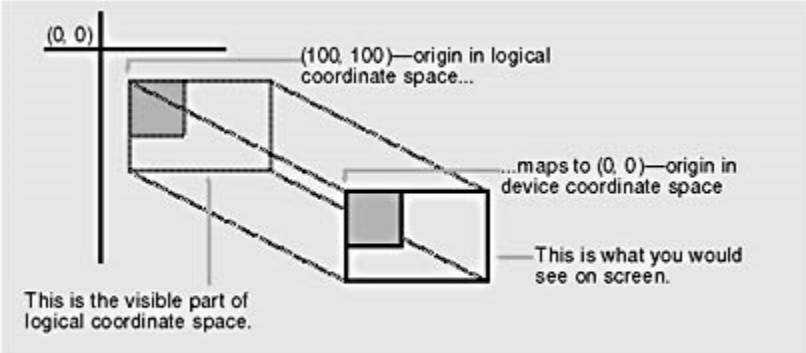


Figure 4-2. *A square drawn after the origin has been moved to (100, 100).*

2. The Fixed-Scale Mapping Modes:

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

Mapping Mode	Logical Unit
MM_LOENGLISH	0.01 inch
MM_HIENGLISH	0.001 inch
MM_LOMETRIC	0.1 mm
MM_HIMETRIC	0.01 mm
MM_TWIPS	$^{1}/_{1440}$ inch

3. The Variable-Scale Mapping Modes:

i. Windows provides two mapping modes, *MM_ISOTROPIC* and *MM_ANISOTROPIC*, that allow you to change the scale factor as well as the origin.

ii. With the *MM_ISOTROPIC* mode, a 1:1 aspect ratio is always preserved.

iii. In other words, a circle is always a circle as the scale factor changes.

iv. With the *MM_ANISOTROPIC* mode, the *x* and *y* scale factors can change independently. Circles can be squished into ellipses.

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

```
void CMyView::OnDraw(CDC* pDC)
{
CRectrectClient;
GetClientRect(rectClient);
pDC->SetMapMode(MM_ANISOTROPIC);
pDC->SetWindowExt(1000, 1000);
pDC->SetViewportExt(rectClient.right, -rectClient.bottom);
pDC->SetViewportOrg(rectClient.right / 2, rectClient.bottom / 2);
pDC->Ellipse(CRect(-500, -500, 500, 500));
}
```

Chapter-5:

The Graphics Device Interface, Colors, and Fonts

Q: What are GDI objects? Explain the constructing and distructing of GDI objects.

Ans:

GDI Objects

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

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

Stock GDI Objects

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

```
void CMyView::OnDraw(CDC* pDC)
{
CPennewPen(PS_DASHDOTDOT, 2, (COLORREF) 0);  // black pen,
                                    //  2 pixels wide
pDC->SelectObject(&newPen);
pDC->MoveTo(10, 10);
pDC->Lineto(110, 10);
```

```
pDC->SelectStockObject(BLACK_PEN);          // newPen is deselected
} // newPen destroyed on exit
```

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

Chapter-6:

The Modal Dialog and Windows Common Controls

Q: Explain the difference between modal and modeless dialog?

Q Write a short note on modal dialog? Explain the steps to create modal dialog.

Ans:

Dialog: A dialog is truly a window that receives messages, that can be moved and closed, and that can even accept drawing instructions in its client area.

There are two kinds of dialog:

1. Modal dialog,
2. Modeless dialod.

CDialog base class supports both modal and modeless dialog. With a modal dialog such as open/file dialog the user

can not work elsewhere the same application until the dialog is close. With a modeless dialog the user can work in another window in the application while the dialog remains on the screen. MS-word find and replace dialog is good example of modelss dialog.

Steps to create modal dialog:

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

Q: Write a short note on Windows Common Control?

Ans:

Windows Common Controls:

1. The Progress Indicator Control:

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

2. The Trackbar Control:

The trackbar control (class *CSliderCtrl*), sometimes called a slider, allows the user to set an "analog" value. CSliderCtrl is the base class for track bar control. If you specify a large range for this control—0 to 100 or more, for example—the trackbar's motion appears continuous. If you specify a small range, such as 0 to 5, the tracker moves in discrete increments. You can program tick marks to match the increments. In this discrete mode, you can use a trackbar to set such items as the display screen resolution, lens f-stop values, and so forth. The trackbar does not have a default range. The trackbar is easier to program than the scroll bar because you don't have to map the WM_HSCROLL or WM_VSCROLL messages in the dialog class.

3. The Spin Button Control:

The spin button control (class *CSpinButtonCtrl*) is an itsy-bitsy scroll bar that's most often used in conjunction with an edit control. CSpinButtonCtrl is the base class of spin button control. The edit control located just ahead of the spin control in the dialog tabbeing order is known as spin controls buddy. If your program uses an integer in the buddy, you can avoid C++ programming almost entirely. Just use ClassWizard to attach an integer data member to the edit control, and set the spin control's range in the *OnInitDialog* function.

4. The List Control:

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

5. The Tree Control:

You're already familiar with tree controls if you've used Microsoft Windows Explorer or Visual C++'s Workspace view. CTreeCtrl is the base class of tree control. The user can expand and collapse elements by clicking the + and - buttons or by double-clicking the elements. The icon next to each item is programmed to change when the user selects the item with a single click.

Chapter-7:

The Modeless Dialog and Windows Common Dialogs

Modeless Dialogs:

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

Creating Modeless Dialogs:

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

Modeless dialogs are more complicated. You start by invoking the *CDialog* default constructor to construct the dialog object, but then to create the dial0og window you need to call the *CDialog::Create* member function instead of *DoModal*. *Create* takes the resource ID as a parameter and returns immediately with the dialog window still on the screen. You must worry about exactly when to construct the dialog object, when to create the dialog window, when to destroy the dialog, and when to process user-entered data.

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

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

User-Defined Messages

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

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

Q: Write a short note on windows common dialog.

Ans:

The Windows Common Dialogs:

Windows provides a group of standard user interface dialogs, and these are supported by the MFC library classes. You are probably familiar with all or most of these dialogs because so many Windows-based applications, including Visual C++, already use them. All the common dialog classes are derived from a common base class, *CCommonDialog*. A list of the COMDLG32 classes is shown in the following table.

Class	Purpose
CColorDialog	Allows the user to select or create a color
CFileDialog	Allows the user to open or save a file
CFindReplaceDialog	Allows the user to substitute one string for another
CPageSetupDialog	Allows the user to input page measurement parameters
CFontDialog	Allows the user to select a font from a list of available fonts
CPrintDialog	Allows the user to set up the printer and print a document

Here's one characteristic that all common dialogs share: they gather information from the user, but they don't do anything with it. The file dialog can help the user select a file to open, but it really just provides your program with the pathname—your program must make the call that opens the file. Similarly, a font dialog fills in a structure that describes a font, but it doesn't create the font.

Using the *CFileDialog* Class Directly

Using the *CFileDialog* class to open a file is easy. The following code opens a file that the user has selected through the dialog:

```
CFileDialogdlg(TRUE, "bmp", "*.bmp");
if (dlg.DoModal() == IDOK) {
CFile file;
   VERIFY(file.Open(dlg.GetPathName(), CFile::modeRead));
}
```

The first constructor parameter (*TRUE*) specifies that this object is a "File Open" dialog instead of a "File Save" dialog. The default file extension is *bmp*, and **.bmp* appears first in the filename edit box. The *CFileDialog::GetPathName* function returns a *CString* object that contains the full pathname of the selected file.

Chapter-8:

Using ActiveX Controls

Q: How ActiveX Controls Are Different from Ordinary Controls? Properties and Methods.

Ans:

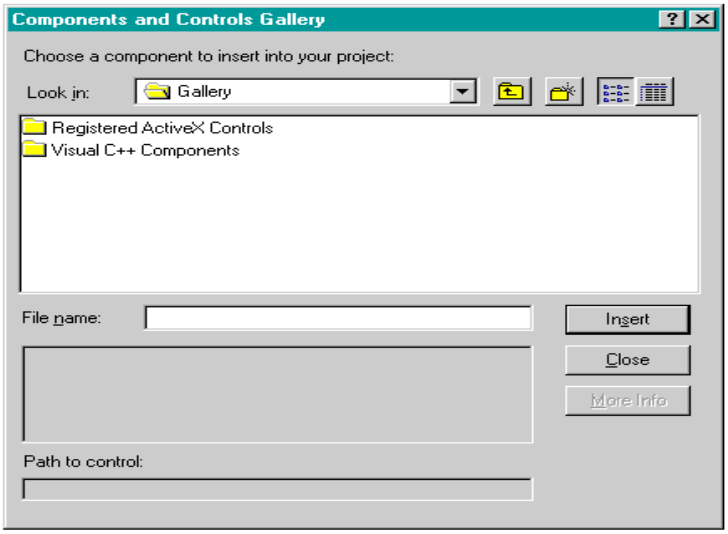
i. ActiveX control are industrial strength replacement for VBX based on Microsoft COM technology (Component object model).
ii. VBX control where written mostly in plane C while ActiveX control can be written in C++ library or with the help of on ActiveX template library (ATL).
iii. An ActiveX control is a software model that plugs into your C++ program * in the same way a window control does.
iv. The most Prominent ActiveX Controls features are properties and methods. Properties have symbolic names that are matched to integer indexes. For each property, the control designer assigns a property name, such as BackColor or GridCellEffect, and a property type, such as string, integer, or double.
v. ActiveX Controls methods are like functions. A method has a symbolic name, a set of parameters, and a return value.
vi. You call a method by calling a C++ member function of the class that represents the control.
vii. The window that contains a control is calledcontainer.
viii. An Active0X control doesn't send WM_ notification messages to its container the way ordinary controls do; instead, it "fires events."
ix. In the MFC world, ActiveX controls act just like child windows, but there's a significant layer of code between the container window and the control window.
x. A DLL is used to store one or more ActiveX controls, but the DLL often has an OCX filename extension instead of a DLL extension.

Q: What is an ActiveX control? How to install ActiveX control?

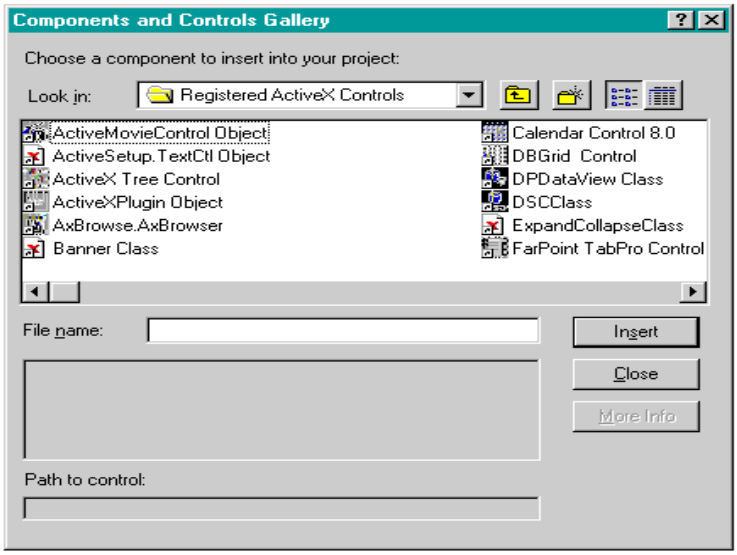
Ans:

Installing ActiveX Controls:

i. Your first step is to copy the control's DLL to your hard disk.
ii. Copy associated files such as help (HLP) or license (LIC) files to the same directory.
iii. Your next step is to register the control in the Windows Registry.
iv. After you register your ActiveX control, you must install it ineachproject that uses it. It means that ClassWizard generates a copy of a C++ class that's specific to the control, and it means that the control shows up in the dialog editor control palette for that project.
v. To install an ActiveX control in a project, choose Add To Project from the Project menu and then choose Components And Controls. Select Registered ActiveX Controls.



vi. This gets you the list of all the ActiveX controls currently registered on your system.



Chapter-9:

Internet Explorer 4 Common Controls

Q: Write a short note on IE4 common controls?

Ans:

1. The Date and Time Picker:

i. A common field on a dialog is a place for the user to enter a date and time

ii. The new date and time picker control is provided as an advanced control that prompts the user for a date or time while offering the developer a wide variety of styles and options.

iii. For example, dates can be displayed in short formats (8/14/68) or long formats (August 14, 1968). A time mode lets the user enter a time using a familiar hours/minutes/seconds AM/PM format.

iv. The control also lets you decide if you want the user to select the date via in-place editing, a pull-down calendar, or a spin button.

v. The new MFC 6.0 class *CDateTimeCtrl*provides the MFC interface to the IE4 date and time picker common control. This class provides a variety of notifications that enhance the programmability of the control.

vi. *CDateTimeCtrl* provides member functions for dealing with either *CTime* or *COleDateTime* time structures.

vii. You set the date and time in a *CDateTimeCtrl* using the *SetTime* member function. You can retrieve the date and time via the *GetTime* function.

viii. You can create custom formats using the *SetFormat* member function and change a variety of other configurations using the *CDateTimeCtrl* interface.

2. The Month Calendar:

i. Like the date and time picker control, the month calendar control lets the user choose a date. However, the month calendar control can also be used to implement a small Personal Information Manager (PIM) in your applications.

ii. You can show as many months as room provides—from one month to a year's worth of months, if you want. EX09A uses the month calendar control to show only two months.

iii. The month calendar control supports single or multiple selection and allows you to display a variety of different options such as numbered months and a circled "today's date."

iv. Notifications for the control let the developer specify which dates are in boldface. It is entirely up to the developer to decide what boldface dates might represent.

v. The MFC 6.0 class *CMonthCalCtrl*implements this control.

vi. To initialize the *CMonthCalCtrl* class, you can call the *SetToday()* member function. *CMonthCalCtrl* provides members that deal with both *CTime* and *COleDateTime,* including*SetToday()*.

3. The Internet Protocol Address Control:

i. If you write an application that uses any form of Internet or TCP/IP functionality, you might need to prompt the user for an Internet Protocol (IP) Address.

ii. This control performs an automatic validation of the entered IP address. *CIPAddressCtrl* provides MFC support for the IP address control.

iii. An IP address consists of four "fields" as shown in Figure 9-2. The fields are numbered from left to right.

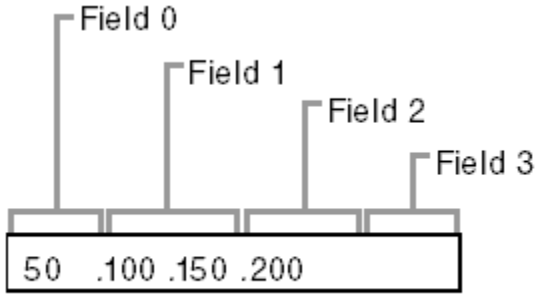


Figure 9-2. *The fields of an IP address control.*

iv. To initialize an IP address control, you call the *SetAddress* member function in your *OnInitDialog* function. *SetAddress* takes a DWORD, with each BYTE in the DWORD representing one of the fields.

v. To retrieve the various values of the four IP address fields.

4. The Extended Combo Box:

i. The extended combo box gives the developer much easier access to and control over the edit-control portion of the combo box.

ii. In addition, the extended combo box lets you attach an image list to the items in the combo box. You can display graphics in the extended combo box easily, especially when compared with the old days of using owner-drawn combo boxes.

iii. Each item in the extended combo box can be associated with three images: a selected image, an unselected image, and an overlay image. These three images can be used to provide a variety of graphical displays in the combo box, as we'll see in the EX09A sample.

iv. CComboBoxEx is a base class for extended combobox.

v. If you are already familiar with *CComboBox*, *CComboBoxEx* might cause some confusion: instead of containing strings, the extended combo box contains items of type *COMBOBOXEXITEM*, a structure that consists of the following fields:

- ***UINT mask***—A set of bit flags that specify which operations are to be performed using the structure.
- ***intiItem***—The extended combo box item number. Like the older style of combo box, the extended combo box uses zero-based indexing.
- ***LPSTR pszText***—The text of the item.
- ***intcchTextMax***—The length of the buffer available in pszText.
- ***intiImage***—Zero-based index into an associated image list.
- ***intiSelectedImage***—Index of the image in the image list to be used to represent the "selected" state.
- ***intiOverlay***—Index of the image in the image list to be used to overlay the current image.
- ***intiIndent***—Number of 10-pixel indentation spaces.
- ***LPARAM lParam***—32-bit parameter for the item.

Chapter-10:

Win32 Memory Management

- **Process:**
  A process is a program in a running mode. A process owns its own memory file handlers and other system resources.

If you launch the same program twice in a row you have two separate process running simultaneously. The important thing about aprocess is that it has its own 4GB virtual address space.

- **Program:**
  A program is an exe file that you can launch in various ways in windows.

Q: Describe win-32 virtual memory management?

Ans:

Virtual Memory Works

Various programs and data elements will be scattered throughout the 4-GB address space in 4-KB units starting on 4-KB boundaries. Each 4-KB unit, called a page, can hold either code or data. When a page is being used, it occupies physical memory, but you never see its physical memory address.

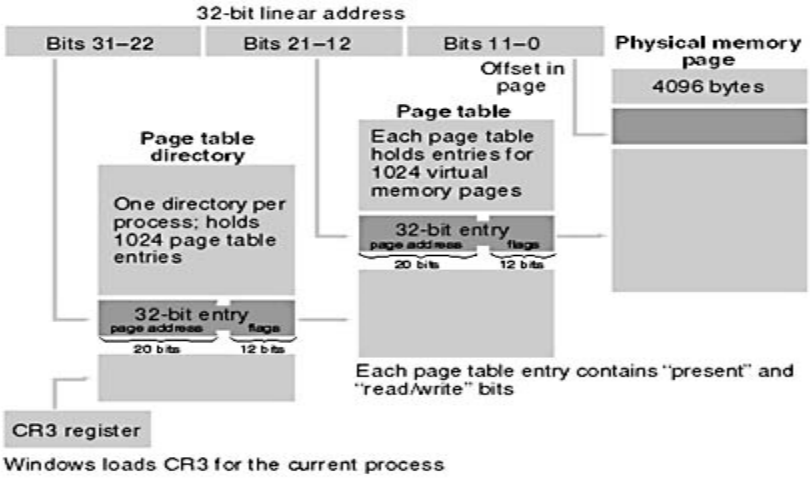


Figure 10-2. *Win32 virtual memory management (Intel).*

The Intel microprocessor chip efficiently maps a 32-bit virtual address to both a physical page and an offset within the page, using two levels of 4-KB page tables, as shown in Figure 10-2. Note that individual pages can be flagged as either read-only or read/write. Also note that each process has its own set of page tables.

The chip's CR3 register holds a pointer to the directory page, so when Windows switches from one process to another, it simply updates CR3. So now our process is down from 4 GB to maybe 5 MB—a definite improvement. But if we're running several programs, along with Windows itself, we'll still run out of RAM. If you look at Figure 10-2 again, you'll notice that the page table entry has a "present" bit that indicates whether the 4-KB page is currently in RAM. If we try to access a page that's not in RAM, an interrupt fires and Windows analyzes the situation by checking its internal tables. If the memory reference was bogus, we'll get the dreaded "page fault" message and the program will exit.

Otherwise, Windows reads the page from a disk file into RAM and updates the page table by loading the physical address and setting the present bit. This is the essence of Win32 virtual memory. At that point the page is copied; as a result, each process has its own private copy stored at the same virtual address.

- **The *VirtualAlloc* Function—Committed and Reserved Memory**

i. If your program needs dynamic memory, sooner or later the Win32 *VirtualAlloc* function will be called.
ii. When memory is reserved, a contiguous virtual address range is set aside. If, for example, you know that your program is going to use a single 5-MB memory block (known as a region) but you don't need to use it all right away, you call *VirtualAlloc* with a *MEM_RESERVE* allocation type parameter and a 5-MB size parameter.
iii. When you get more serious about needing memory, you call *VirtualAlloc* again to commit the reserved memory, using a *MEM_COMMIT* allocation type parameter.
iv. You call the *VirtualFree* function to "decommit" committed memory.

- **The Windows Heap and the *GlobalAlloc* Function Family**

A heap is a memory pool for a specific process. When your program needs a block of memory, it calls a heap allocation function, and it calls a companion function to free the memory.

The *HeapAlloc* function allocates memory in a Windows heap, and *HeapFree* releases it. You might never need to call *HeapAlloc* yourself, but it will be called for you by the *GlobalAlloc* function that's left over from Win16. In the ideal 32-bit world, you wouldn't have to use *GlobalAlloc*, but in this real world, we're stuck with a lot of code ported from Win16 that uses "memory handle" (*HGLOBAL*) parameters instead of 32-bit memory addresses.

GlobalAlloc uses the default Windows heap. It does two different things, depending on its attribute parameter. If you specify *GMEM_FIXED*, *GlobalAlloc* simply calls *HeapAlloc* and returns the address cast as a 32-bit *HGLOBAL* value. If you specify *GMEM_MOVEABLE*, the returned *HGLOBAL* value is a pointer to a handletable entry in your process. That entry contains a pointer to the actual memory, which is allocated with *HeapAlloc*.

Chapter-12:

Bitmaps

Windows bitmaps are arrays of bits mapped to display pixels. There are two kinds of Windows bitmaps: GDIbitmaps and DIBs.

Q: Explain GDI Bitmap and Device Independent Bitmap?

Ans:

1. **GDI Bitmaps:**

i. GDI bitmap objects are represented by the Microsoft Foundation Class (MFC) Library version 6.0 CBitmap class.
ii. The GDI bitmap object has an associated Windows data structure, maintained inside the Windows GDI module, that is device-dependent.
iii. Your program can get a copy of the bitmap data, but the bit arrangement depends on the display hardware.
iv. GDI bitmaps can be freely transferred among programs on a single computer, but because of their device dependency, transferring bitmaps by disk or modem doesn't make sense.

2. **DIB Bitmaps:**

i. DIBs offer many programming advantages over GDI bitmaps. Because a DIB carries its own color information, color palette management is easier.
ii. DIBs also make it easy to control gray shades when printing.
iii. Any computer running Windows can process DIBs, which are usually stored in BMP disk files or as a resource in your program's EXE or DLL file.
iv. The wallpaper background on your monitor is read from a BMP file when you start Windows.
v. The primary storage format for Microsoft Paint is the BMP file, and Visual C++ uses BMP files for toolbar buttons and other images.

vi. Other graphic interchange formats are available, such as TIFF, GIF, and JPEG, but only the DIB format is directly supported by the Win32 API.

- **Color Bitmaps and Monochrome Bitmaps:**

Many color bitmaps are 16-color. A standard VGA board has four contiguous color planes, with 1 corresponding bit from each plane combining to represent a pixel. The 4-bit color values are set when the bitmap is created. With a standard VGA board, bitmap colors are limited to the standard 16 colors. Windows does not use dithered colors in bitmaps.

A monochrome bitmap has only one plane. Each pixel is represented by a single bit that is either off (0) or on (1). The *CDC::SetTextColor* function sets the "off" display color, and *SetBkColor* sets the "on" color. You can specify these pure colors individually with the Windows *RGB* macro.

- **DIB Access Functions:**

Windows supplies some important DIB access functions. None of these functions is wrapped by MFC, so you'll need to refer to the online Win32 documentation for details.

- ***SetDIBitsToDevice***—This function displays a DIB directly on the display or printer. No scaling occurs; one bitmap bit corresponds to one display pixel or one printer dot. This scaling restriction limits the function's usefulness. The function doesn't work like *BitBlt* because *BitBlt* uses logical coordinates.
- ***StretchDIBits***—This function displays a DIB directly on the display or printer in a manner similar to that of *StretchBlt*.
- ***GetDIBits***—This function constructs a DIB from a GDI bitmap, using memory that you allocate. You have some control over the format of the DIB because you can specify the number of color bits per pixel and the compression. If you are using compression, you have to call *GetDIBits* twice—once to calculate the memory needed and again to generate the DIB data.
- ***CreateDIBitmap***—This function creates a GDI bitmap from a DIB. As for all these DIB functions, you must supply a device context pointer as a parameter. A display device context will do; you don't need a memory device context.
- ***CreateDIBSection***—This Win32 function creates a special kind of DIB known as a DIBsection. It then returns a GDI bitmap handle. This function gives you the best features of DIBs and GDI bitmaps. You have direct access to the DIB's memory, and with the bitmap handle and a memory device context, you can call GDI functions to draw into the DIB.

Chapter-12:
Windows Message Processing and Multithreaded Programming.
Windows Message Processing

To understand threads, you must first understand how 32-bit Windows processes messages. The best starting point is a single-threaded program that shows the importance of the message translation and dispatch process. You'll improve that program by adding a second thread, which you'll control with a global variable and a simple message.

- **Thread:**

i. A thread of execution is the smallest sequence of program instruction that can be managed by operating system scheduler.
ii. This implementation of thread and process differ from one OS to another. But in most case a thread is contain inside a process.

Q: How a Single-Threaded Program Processes Messages? Explain with example.

Ans:

All the programs so far in this book have been single-threaded, which means that your code has only one path of execution. With ClassWizard's help, you've written handler functions for various Windows messages and you've written *OnDraw* code that is called in response to the WM_PAINT

message. It might seem as though Windows magically calls your handler when the message floats in, but it doesn't work that way. Deep inside the MFC code (which is linked to your program) are instructions that look something like this:

```
MSG message;
while (::GetMessage(&message, NULL, 0, 0)) {
   ::TranslateMessage(&message);
   ::DispatchMessage(&message);
}
```

i. Windows determines which messages belong to your program, and the *GetMessage* function returns when a message needs to be processed. If no messages are posted, your program is suspended and other programs can run.
ii. When a message eventually arrives, your program "wakes up."
iii. The *TranslateMessage* function translates WM_KEYDOWN messages into WM_CHAR messages containing ASCII characters, and the *DispatchMessage* function passes control (via the window class) to the MFC message pump, which calls your function via the message map.
iv. When your handler is finished, it returns to the MFC code, which eventually causes *DispatchMessage* to return.

- **Yielding Control:**

What would happen if one of your handler functions was a pig and chewed up 10 seconds of CPU time? Back in the 16-bit days, that would have hung up the whole computer for the duration. Only cursor tracking and a few other interrupt-based tasks would have run. With Win32, multitasking got a whole lot better. Other applications can run because of preemptive multitasking—Windows simply interrupts your pig function when it needs to. However, even in Win32, your program would be locked out for 10 seconds. It couldn't process any messages because *DispatchMessage* doesn't return until the pig returns.

There is a way around this problem, however, which works with both Win16 and Win32. You simply train your pig function to be polite and yield control once in a while by inserting the following instructions inside the pig's main loop:

```
MSG message;
if (::PeekMessage(&message, NULL, 0, 0, PM_REMOVE)) {
   ::TranslateMessage(&message);
   ::DispatchMessage(&message);
}
```

The *PeekMessage* function works like *GetMessage*, except that it returns immediately even if no message has arrived for your program. In that case, the pig keeps on chewing. If there is a message, however, the pig pauses, the handler is called, and the pig starts up again after the handler exits.

- **Timers:**

A Windows timer is a useful programming element that sometimes makes multithreaded programming unnecessary. If you need to read a communication buffer, for example, you can set up a timer to retrieve the accumulated characters every 100 milliseconds. You can also use a timer to control animation because the timer is independent of CPU clock speed.

Timers are easy to use. You simply call the *CWnd* member function *SetTimer* with an interval parameter, and then you provide, with the help of ClassWizard, a message handler function for the resulting WM_TIMER messages. Once you start the timer with a specified interval in milliseconds, WM_TIMER messages will be sent continuously to your window until you call *CWnd::KillTimer* or until the timer's window is destroyed. If you want to, you can use multiple timers, each identified by

an integer. Because Windows isn't a real-time operating system, the interval between timer events becomes imprecise if you specify an interval much less than 100 milliseconds.
Like any other Windows messages, timer messages can be blocked by other handler functions in your program. Fortunately, timer messages don't stack up. Windows won't put a timer message in the queue if a message for that timer is already present.

Q: What do you mean by multithreaded programming?
Ans:
Multithreaded Programming

Windows offers two kinds of threads, workerthreads and userinterface threads. The Microsoft Foundation Class (MFC) Library supports both. A user interface thread has windows, and therefore it has its own message loop. A worker thread doesn't have windows, so it doesn't need to process messages. Worker threads are easier to program and are generally more useful.

Writing the Worker Thread Function and Starting the Thread

To start the thread (with function name *ComputeThreadProc*), your program makes the following call:

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

The compute thread code looks like this:

```
UINT ComputeThreadProc(LPVOID pParam)
{
   // Do thread processing
   return 0;
}
```

The *AfxBeginThread* function returns immediately; the return value is a pointer to the newly created thread object. You can use that pointer to suspend and resume the thread (*CWinThread::SuspendThread* and *ResumeThread*), but the thread object has no member function to terminate the thread. The second parameter is the 32-bit value that gets passed to the global function, and the third parameter is the thread's priority code. Once the worker thread starts, both threads run independently. Windows divides the time between the two threads (and among the threads that belong to other processes) according to their priority. If the main thread is waiting for a message, the compute thread can still run.

How the Main Thread Talks to a Worker Thread

The main thread (your application program) can communicate with the subsidiary worker thread in many different ways. One option that will not work, however, is a Windows message; the worker thread doesn't have a message loop. The simplest means of communication is a global variable because all the threads in the process have access to all the globals. Suppose the worker thread increments and tests a global integer as it computes and then exits when the value reaches 100. The main thread could force the worker thread to terminate by setting the global variable to 100 or higher.
The code below looks as though it should work, and when you test it, it probably will:

```
UINT ComputeThreadProc(LPVOID pParam)
{
g_nCount = 0;
   while (g_nCount++ < 100) {
      // Do some computation here
   }
   return 0;
}
```

How the Worker Thread Talks to the Main Thread

A Windows message is the preferred way for a worker thread to communicate with the main thread because the main thread always has a message loop. This implies, however, that the main thread has a window (visible or invisible) and that the worker thread has a handle to that window.

How does the worker thread get the handle? That's what the 32-bit thread function parameter is for. You pass the handle in the *AfxBeginThread* call. Why not pass the C++ window pointer instead? Doing so would be dangerous because you can't depend on the continued existence of the object and you're not allowed to share objects of MFC classes among threads. (This rule does not apply to objects derived directly from *CObject* or to simple classes such as *CRect* and *CString*.)