Module 2: Getting Started with Visual C++ 6.0 AppWizard

Program examples compiled using Visual C++ 6.0 (MFC 6.0) compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this Tutorial are listed below:

The Windows Programming Model
Window Procedure
Messages
Common Windows Messages
Windows Messages
Command Messages
Ranges of Messages
Message Categories Details
Windows Messages and Control-Notification Messages
Message Handling and Mapping
Documents and Views Story
Using Documents
Using Views
SDI and MDI
More on View
Using the AppWizard of the Visual C++ 6.0
Drawing Inside the View Window: The Windows Graphics Device Interface
The `OnDraw()` Member Function
The Windows Device Context
Adding Draw Code to the MYMFC Program
A Preview of the Resource Editors
The Contents of `mymfc.rc` file
Running the Dialog Resource Editor
Enabling the Diagnostic Macros
Two Ways to Run a Program
AFX Functions
Precompiled Headers Story

The Windows Programming Model

Windows programs use the **event-driven programming** model illustrated in Figure 1, in which applications respond to **events** by processing **messages** sent by the operating system. An **event** could be a keystroke, a mouse click, or a command for a window to repaint itself, among other things. The entry point for a Windows program is a function named `WinMain()`, but most of the action takes place in a function known as the **window procedure**. The window procedure processes messages sent to the window. `WinMain()` creates that window and then enters a message loop, alternately retrieving messages and dispatching them to the window procedure. Messages wait in a message queue until they are retrieved. A typical Windows application performs the bulk of its processing in response to the messages it receives, and in between messages, it does little except wait for the next message to arrive.
The message loop ends when a `WM_QUIT` message is retrieved from the message queue, signaling that it's time for the application to end. This message usually appears because the user selected **Exit** from the **File** menu, clicked the close button (the small button with an **X** in the window's upper right corner), or selected **Close** from the window's system menu. When the message loop ends, `WinMain()` returns and the application terminates.

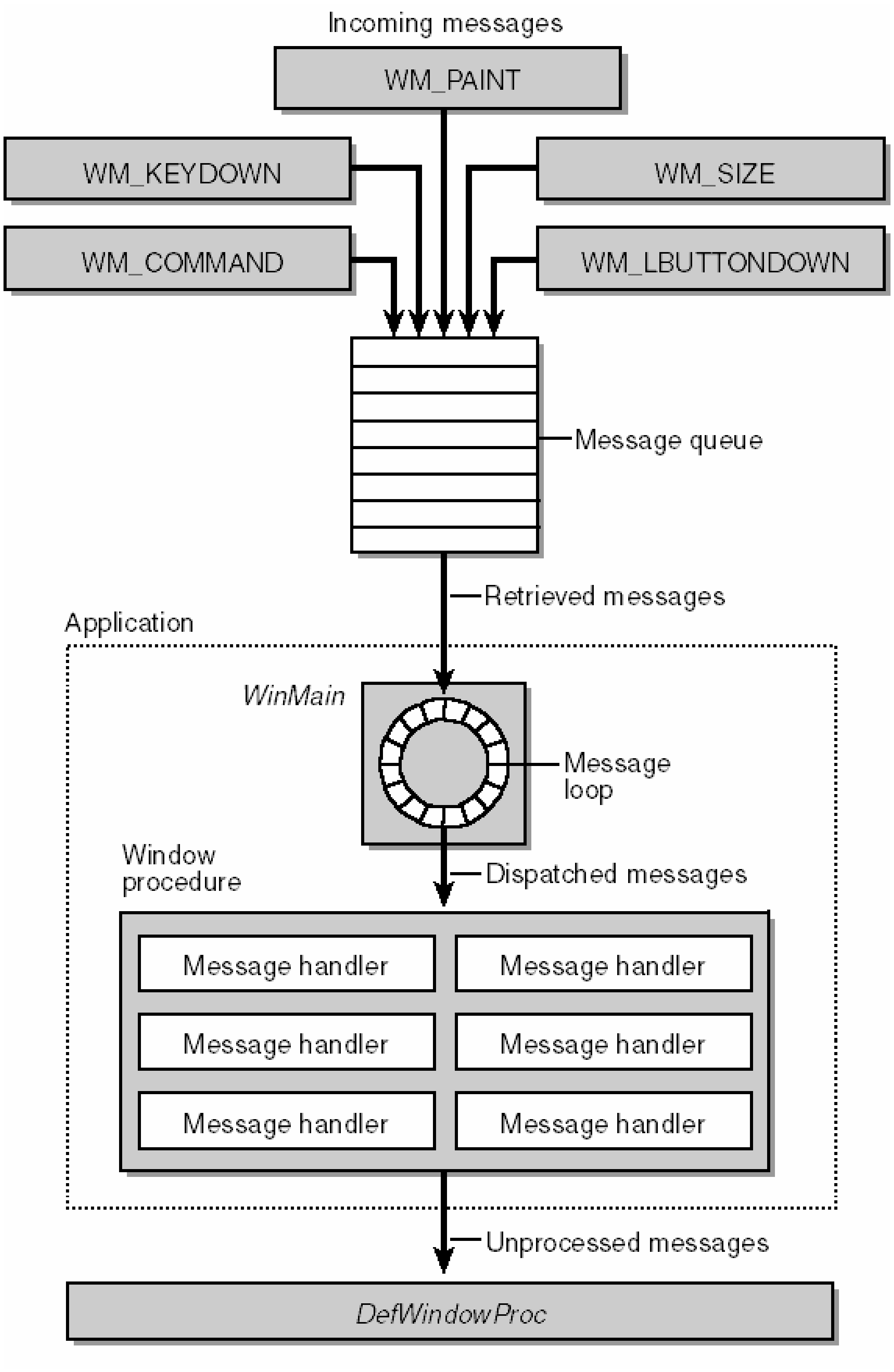


Figure 1: A simple Windows programming model.

Window Procedure

A window procedure is a **function** that receives and processes all messages sent to the window. Every window class has a window procedure, and every window created with that class uses that same window procedure to respond to messages.
The system sends a message to a window procedure by passing the **message data** as arguments to the procedure. The window procedure then performs an appropriate action for the message; it checks the **message identifier** (ID) and, while processing the message, uses the information specified by the **message parameters**.
A window procedure does not usually ignore a message. If it does not process a message, it must send the message back to the system for default processing. The window procedure does this by calling the `DefWindowProc()` function, which performs a default action and returns a message result. The window procedure must then return this value as its own message result. Most window procedures process just a few messages and pass the others on to the system by calling `DefWindowProc()`. Because a window procedure is shared by all windows belonging to the same class, it can process messages for several different windows. To identify the specific window affected by the message, a window procedure can examine the **window handle** passed with a message.
The window procedure typically calls other functions to help process the messages it receives. It can call functions local to the application, or it can call API functions provided by Windows. API functions are contained in special modules known as dynamic-link libraries, or DLLs. The Win32 API includes hundreds of functions that an application can call to perform various tasks such as creating a window, drawing a line, and performing file input and output. In C, the window procedure is typically implemented as a monolithic function containing a large switch statement with cases for individual messages. The code provided to process a particular message is known as a **message handler**.

Messages

Windows defines hundreds of different message types. Most messages have names that begin with the letters "**WM_**", as in `WM_CREATE` and `WM_PAINT`. These messages can be classified in various ways, but for the moment classification is not nearly as important as realizing the critical role messages play in the operation of an application. The following table shows 10 of the most common messages. A window receives a `WM_PAINT` message, for example, when its interior needs repainting. One way to characterize a Windows program is to think of it as a **collection of message handlers**. To a large extent, it is a program's unique way of responding to messages that gives it its personality.

Common Windows Messages

Message	Sent When
WM_CHAR	A character is input from the keyboard.
WM_COMMAND	The user selects an item from a menu, or a control sends a notification to its parent.
WM_CREATE	A window is created.
WM_DESTROY	A window is destroyed.
WM_LBUTTONDOWN	The left mouse button is pressed.
WM_LBUTTONUP	The left mouse button is released.
WM_MOUSEMOVE	The mouse pointer is moved.
WM_PAINT	A window needs repainting.
WM_QUIT	The application is about to terminate.
WM_SIZE	A window is resized.

Table 1: Windows messages example.

A message manifests itself in the form of a call to a window's window procedure. Bundled with the call are four input parameters:

1. The handle of the window to which the message is directed,
2. A message ID, and
3. Two 32-bit parameters known as `wParam` and `lParam`.

The window handle is a 32-bit value that uniquely identifies a window. Internally, the value references a data structure in which Windows stores relevant information about the window such as its size, style, and location on the screen. The message ID is a numeric value that identifies the message type: WM_CREATE, WM_PAINT, and so on. wParam and lParam contain information specific to the message type. When a WM_LBUTTONDOWN message arrives, for example, wParam holds a series of bit flags identifying the state of the Ctrl and Shift keys and of the mouse buttons. lParam holds two 16-bit values identifying the location of the mouse pointer when the click occurred. Together, these parameters provide the window procedure with all the information it needs to process the WM_LBUTTONDOWN message. A large portion of programming for the Windows environment involves message handling. Each time an event such as a keystroke or mouse click occurs, a message is sent to the application, which must then handle the event. The Microsoft Foundation Class Library offers a programming model optimized for message-based programming. In this model, "**message maps**" are used to designate which **functions** will handle various **messages** for a particular **class**. Message maps contain one or more macros that specify **which messages will be handled by which functions**. For example, a message map containing an ON_COMMAND macro might look something like this:

```
BEGIN_MESSAGE_MAP(CMyDoc, CDocument)
    ON_COMMAND(ID_MYCMD, OnMyCommand)
    // ... More entries to handle additional commands
END_MESSAGE_MAP( )
```

The ON_COMMAND macro is used to handle command messages generated by menus, buttons, and accelerator keys. Macros are available to map the following:

Windows Messages

- Control notifications.
- User-defined messages.

Command Messages

- Registered user-defined messages.
- User-interface update messages.

Ranges of Messages

- Commands.
- Update handler messages.
- Control notifications.

Although message-map macros are important, you generally won't have to use them directly. This is because the **ClassWizard** window automatically creates message-map entries in your source files when you use it to associate message-handling functions with messages. Any time you want to edit or add a message-map entry, you can use the ClassWizard. The ClassWizard does not support **message-map ranges**. You must write these message-map entries yourself.

Message Categories Details

There are three main categories of messages that you will write in Windows programming later:

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

2. **Control notifications**
   This includes WM_COMMAND notification messages from controls (buttons, list boxes, scroll bars etc.) and other child windows to their parent windows. For example, an edit control sends its parent a WM_COMMAND message containing the EN_CHANGE control-notification code when the user has taken an action that may have altered text in the edit control. The window's handler for the message responds to the notification message in some appropriate way, such as retrieving the text in the control. The framework routes control-notification messages like other WM_ messages. One exception, however, is the BN_CLICKED control-notification message sent by

buttons when the user clicks them. This message is treated specially as a command message and routed like other commands.

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

Windows Messages and Control-Notification Messages

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

Messages in category 3 – commands
Can be handled by a wider variety of objects: documents, document templates, and the application object itself in addition to windows and views. When a command directly affects some particular object, it makes sense to have that object handle the command. For example, the **Open** command on the **File** menu is logically associated with the application: the application opens a specified document upon receiving the command. So the handler for the Open command is a member function of the application class.

Message Handling and Mapping

In traditional programs for Windows, Windows messages are handled in a large switch statement in a window procedure. MFC instead uses message maps to map direct messages to distinct class member functions. Message maps are more efficient than virtual functions for this purpose, and they allow messages to be handled by the most appropriate C++ object such as application, document, view, and so on. You can map a single message or a range of messages, command IDs, or control IDs.
`WM_COMMAND` messages - usually generated by menus, toolbar buttons, or accelerators, also use the message-map mechanism. MFC defines a standard routing of command messages among the application, frame window, view, and Active documents in your program. You can override this routing if you need to. Message maps also supply a way to update user-interface objects (such as menus and toolbar buttons), enabling or disabling them to suit the current context.

Documents and Views Story

Most of your MFC library applications will be more complex than the previous examples. Typically, they'll contain **application** and **frame** classes plus two other classes that represent the "**document**" and the "**view**." This document-view architecture is the core of the application framework and is loosely based on the Model/View/Controller classes from the **Smalltalk** world.
In simple terms, the document-view architecture separates **data** from the user's **view** of the data. One obvious benefit is multiple views of the same data. Consider a **document** that consists of a month's worth of stock quotes stored on disk. Suppose a **table view** and a **chart view** of the data are both available. The user updates values through the table view window, and the chart view window changes because both windows display the same information but in different views.
In an MFC library application, documents and views are represented by instances of C++ classes. Figure 2 shows three objects of class `CStockDoc` corresponding to three companies: AT&T, IBM, and GM. All three documents have a table view attached, and one document also has a chart view. As you can see, there are four view objects: three objects of class `CStockTableView` and one of class `CStockChartView`.

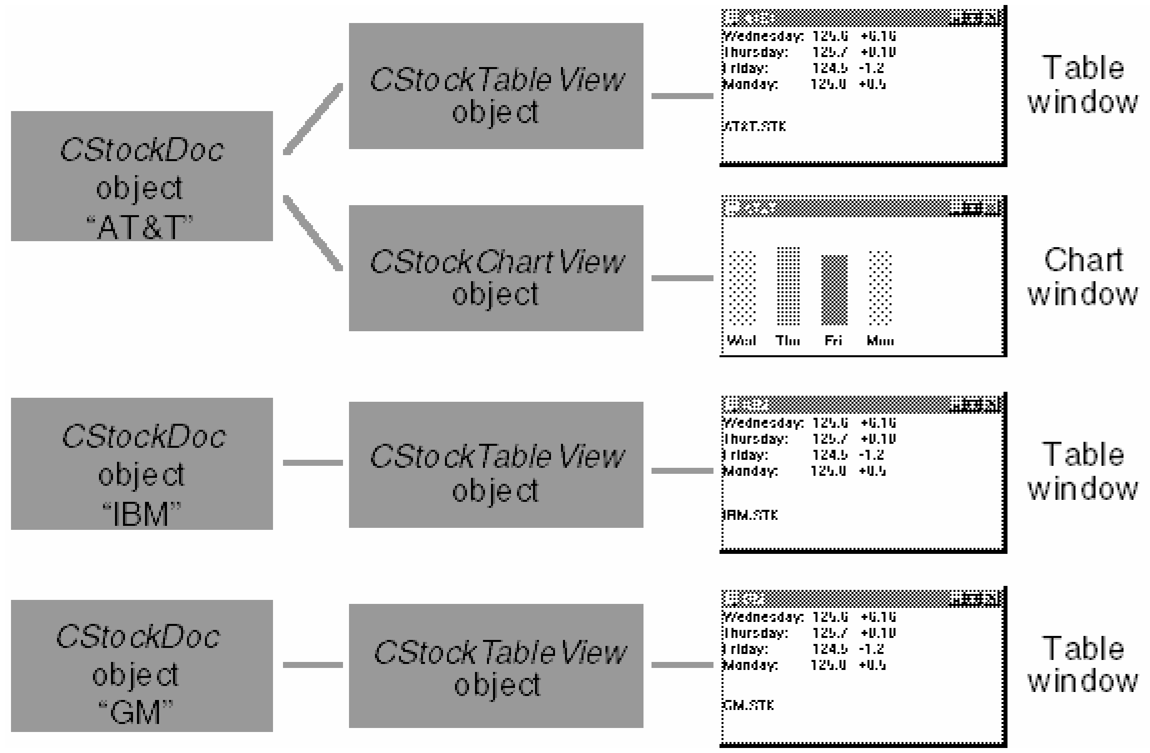


Figure 2: The document-view relationship.

Then, 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.

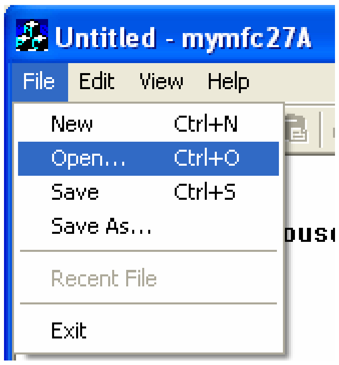


Figure 3: The document's **File Open** and application framework.

The **application framework** does most of the work of displaying the File Open and File Save dialog boxes and opening, closing, reading, and writing files. 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. 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. Don't think that a document object must be associated with a disk file that is read entirely into memory. If a "document" were really a database, for example, you could override selected document class member functions and the File Open menu item would bring up a list of databases instead of a list of files. At the heart of document/view are **four key classes**:

1. The `CDocument` (or `COleDocument`) class supports objects used to store or control your program's data and provides the basic functionality for programmer-defined document classes. A document represents the **unit of data** that the user typically opens with the Open command on the File menu and saves with the Save command on the File menu.

2. The `CView` (or one of its many derived classes) provides the basic functionality for programmer-defined view classes. A view is **attached to a document** and acts as an **intermediary between the document and the user**: the view renders an image of the document on the screen and interprets user input as operations upon the document. The view also renders the image for both printing and print preview.
3. `CFrameWnd` (or one of its variations) supports objects that provide the frame around one or more views of a document.
4. `CDocTemplate` (or `CSingleDocTemplate` or `CMultiDocTemplate`) supports an object that coordinates one or more existing documents of a given type and manages creating the correct document, view, and frame window objects for that type.

The following is another figure that shows the relationship between a document and its view.

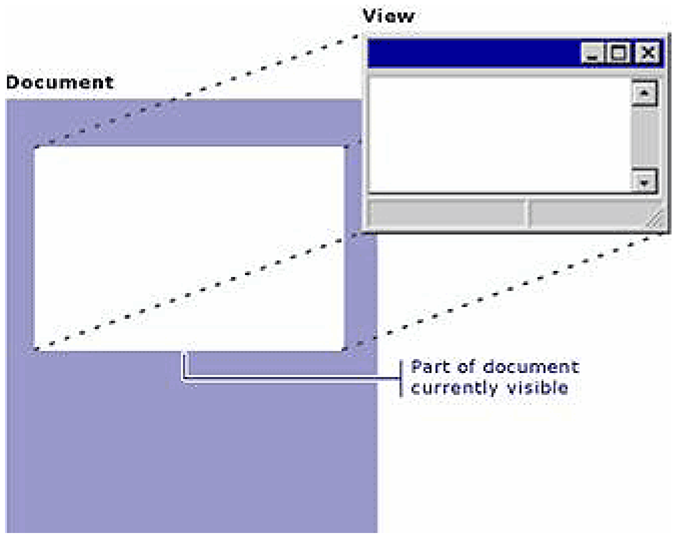


Figure 4: The document-view relationship.

The document/view implementation in the class library separates the **data** itself from its **display** and from **user operations** on the data. All changes to the data are managed through the **document class**. The view calls this interface to access and update the data. Documents, their associated views, and the frame windows that frame the views are created by a document template. The document template is responsible for creating and managing all documents of one document type.

Using Documents

Working together, documents and views:

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

The document is particularly involved in managing data. **Store your data**, normally, in document class **member variables**. The view uses these variables to access the data for display and update. The document's default serialization mechanism manages reading and writing the data to and from files. Documents can also handle commands (but not Windows messages other than `WM_COMMAND`).

Using Views

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

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

SDI and MDI

MFC supports two types of document/view applications.

1. **Single document interface** (SDI) applications support just one open document at a time.
2. **Multiple document interface** (MDI) applications permit two or more documents to be open concurrently and also support multiple views of a given document.

The WordPad applet is an SDI application; Microsoft Word is an MDI application. The framework hides many of the differences between the two user interface models so that writing an MDI application is not much different than writing an SDI application, but today developers are discouraged from using the multiple document interface because the SDI model promotes a more document-centric user interface. If the user is to edit two documents simultaneously, Microsoft would prefer that each document be displayed in a separate instance of your application.
SDI applications allow only one open document frame window at a time. MDI applications allow multiple document frame windows to be open in the same instance of an application. An MDI application has a window within which multiple MDI child windows, which are frame windows themselves, can be opened, each containing a separate document. In some applications, the child windows can be of different types, such as chart windows and spreadsheet windows. In that case, the menu bar can change as MDI child windows of different types are activated. Under Windows 95 and later, applications are commonly SDI because the operating system has adopted a "document-centered" view.

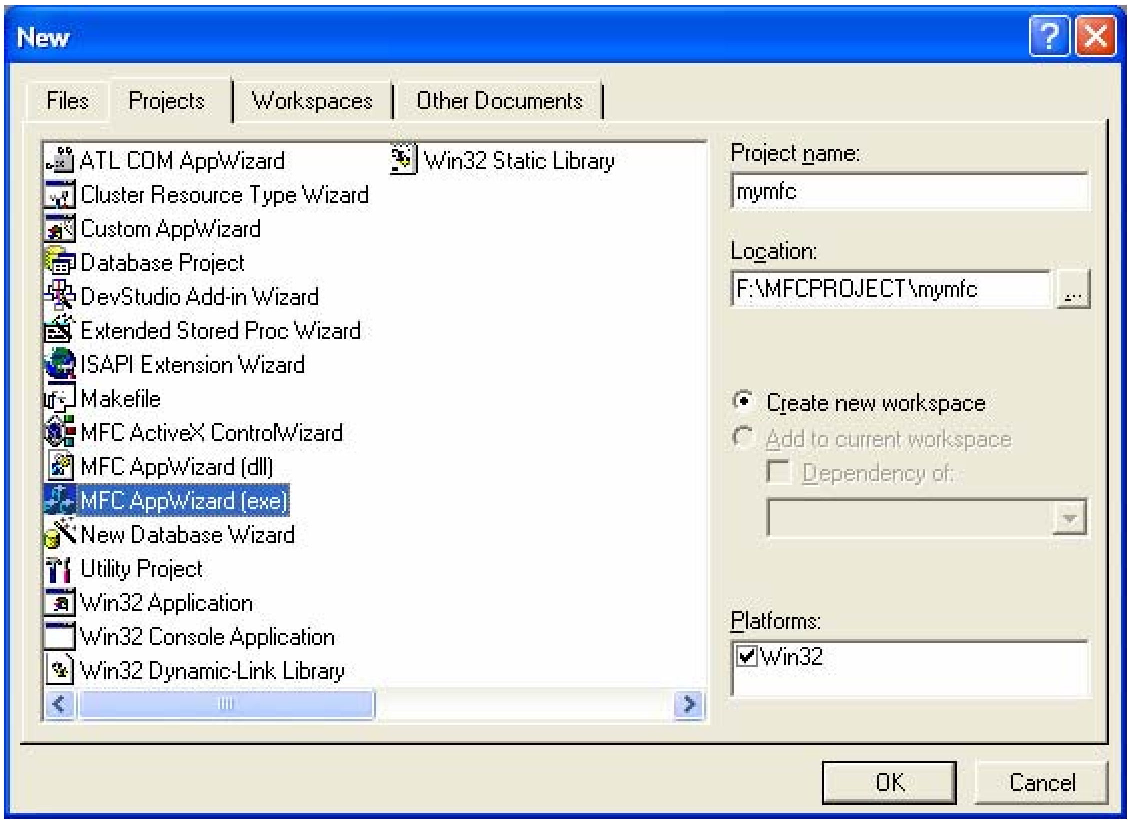
More on View

At this stage, we will concentrate on Views. From a user's 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 window. From the programmer's perspective, a view is a C++ object of a class derived from the MFC library `CView` class. Like any C++ object, the view object's behavior is determined by the **member functions** (and **data members**) of the class - both the application-specific functions in the derived class and the standard functions inherited from the base classes.
With Visual C++, you can produce interesting applications for Windows by simply adding code to the derived view class that the AppWizard code generator produces. When your program runs, the MFC library application framework constructs an object of the derived view class and displays a window that is tightly linked to the C++ view object. As is customary in C++ programming, the view class code is divided into two source modules: the **header file** (.h) and the **implementation file** (.cpp).

Using the AppWizard of the Visual C++ 6.0

The AppWizard tool generates the code for a functioning MFC library application. This working application simply brings up an empty window with a menu attached. Later you'll add code that draws inside the window. Follow the steps to build the application:

1. Run the **AppWizard** to generate SDI application source code. Choose **New** from Visual C++'s **File** menu, and then click the **Projects** tab in the resulting **New** dialog box, as shown here. Make sure that **MFC AppWizard (exe)** is highlighted, and then type **mymfc** as shown in the **Project Name** edit box, modify the **Location** of your project as needed and then click the **OK** button.



Project name: mymfc

Location: F:\MFCPROJECT\mymfc

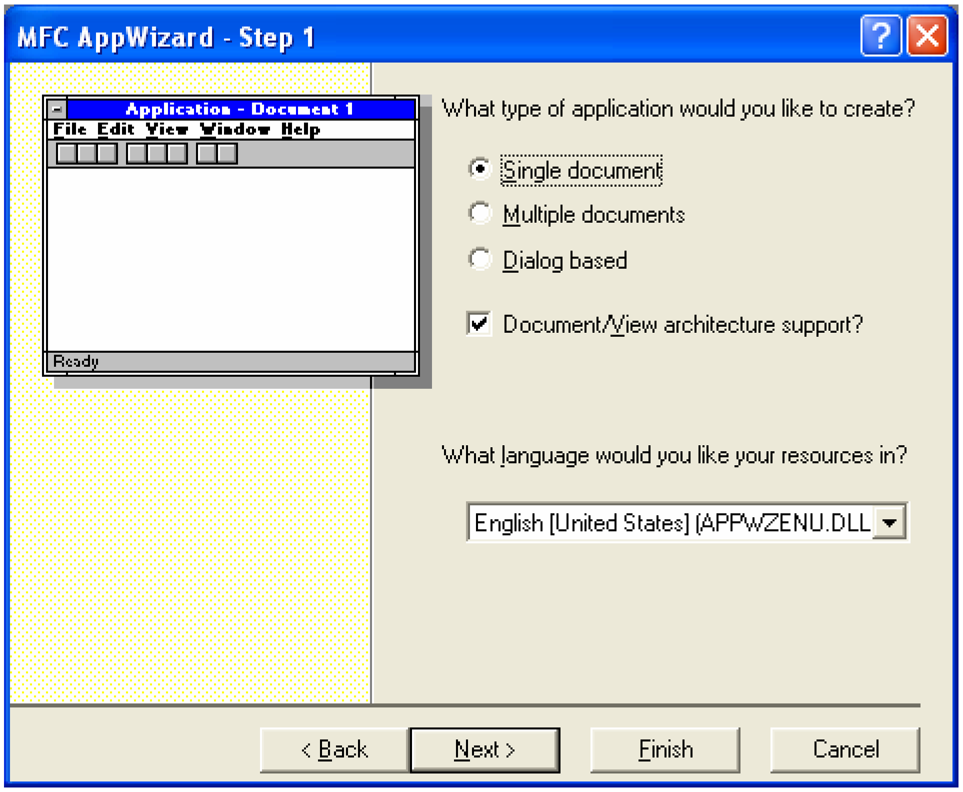
[✓ Create new workspace]

Add to current workspace

Platforms: [✓ Win32]

Figure 5: Visual C++ AppWizard **New** project dialog.

2. Now you will step through a sequence of AppWizard screens, the first of which is shown here. Just follow the steps and options selected in the Figures.



What type of application would you like to create?

[✓ Single document]

Multiple documents

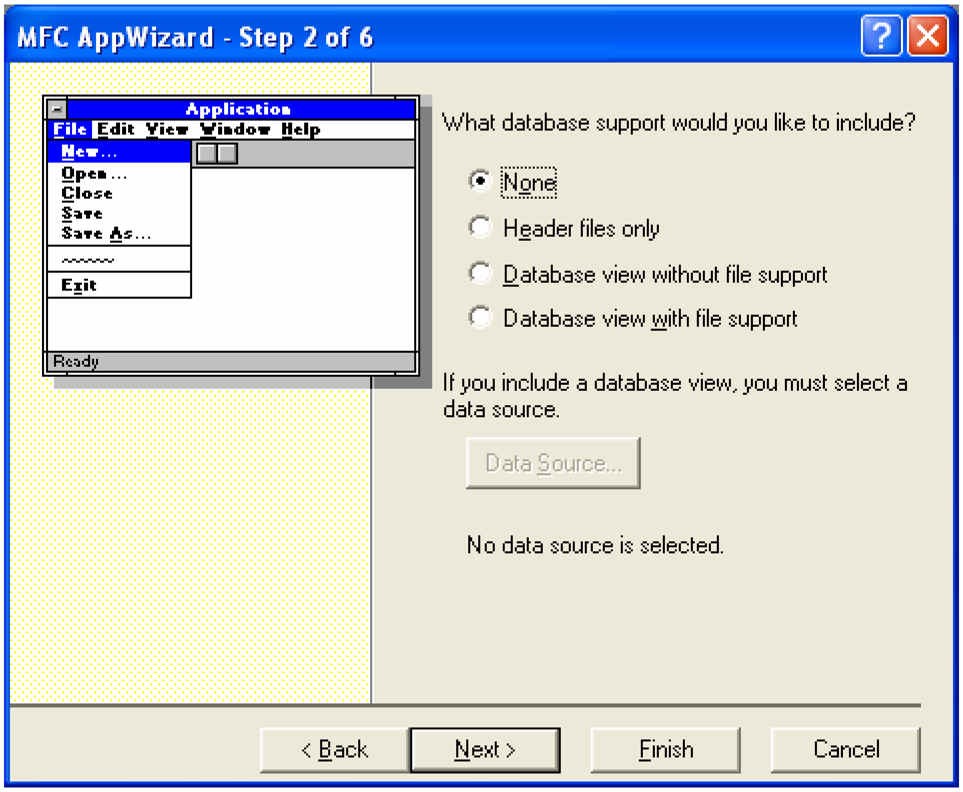
Dialog based

[✓ Document/View architecture support?]

What language would you like your resources in?

English [United States] (APPWZENU.DLL

Figure 6: Visual C++ AppWizard step 1 of 6.



What database support would you like to include?

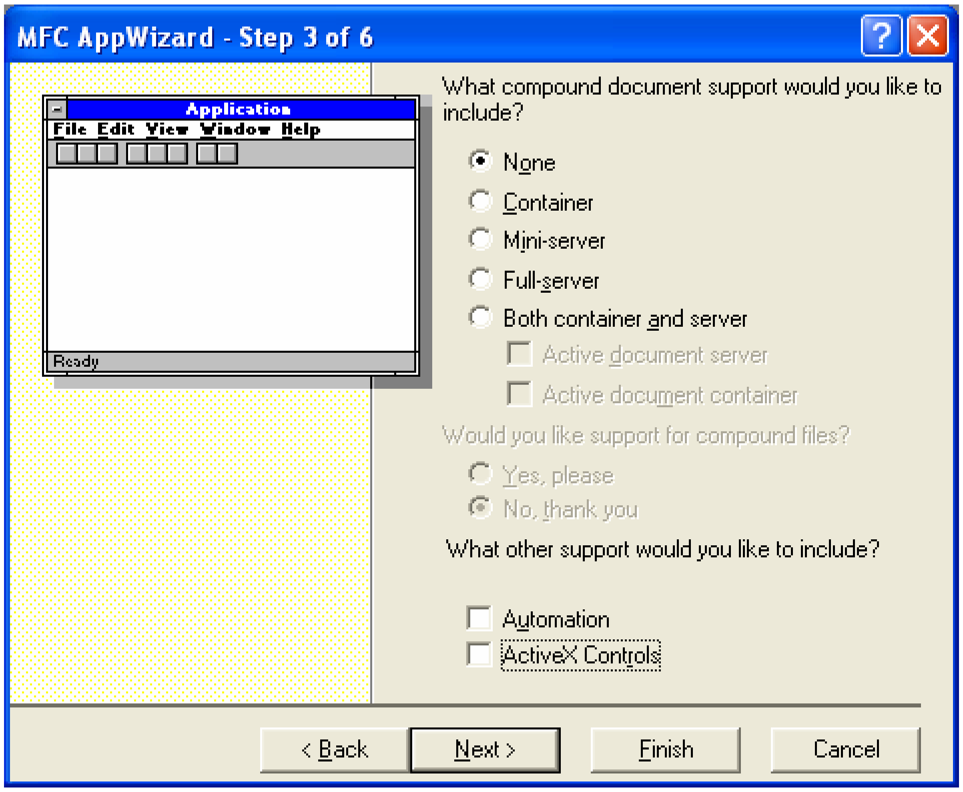
- [✓ None]
- Header files only
- Database view without file support
- Database view with file support

If you include a database view, you must select a data source.

Data Source...

No data source is selected.

Figure 7: Visual C++ AppWizard step 2 of 6.



What compound document support would you like to include?

- [✓ None]
- Container
- Mini-server
- Full-server
- Both container and server
  - Active document server
  - Active document container

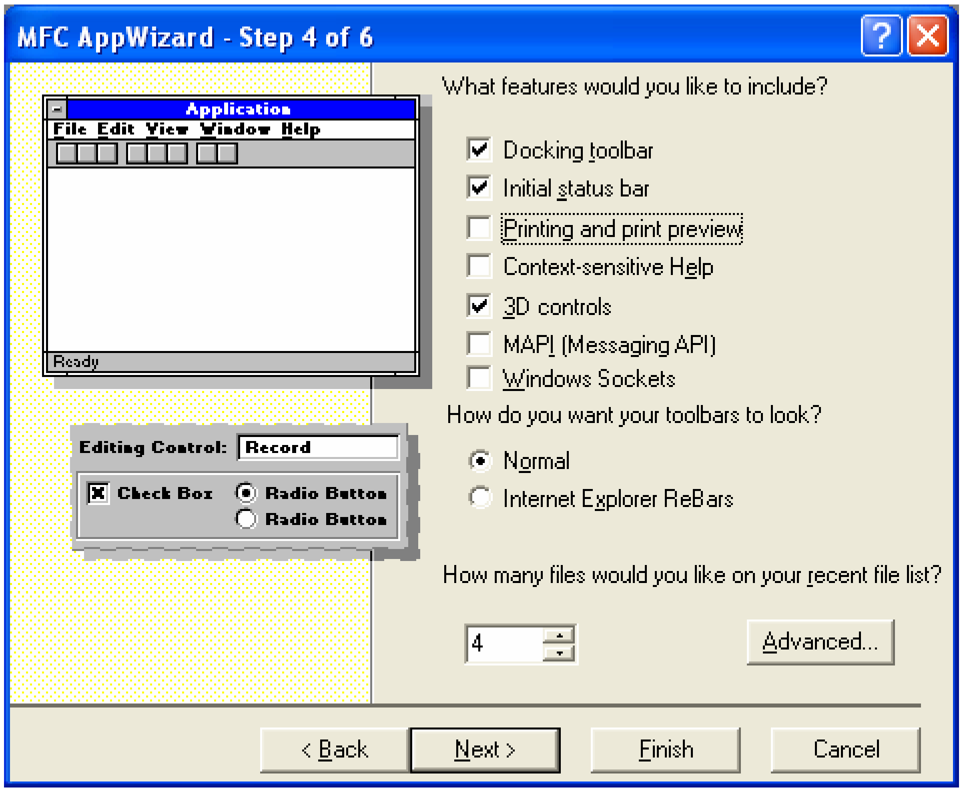
Would you like support for compound files?

- Yes, please
- [✓ No, thank you]

What other support would you like to include?

- Automation
- ActiveX Controls

Figure 8: Visual C++ AppWizard step 3 of 6.


MFC AppWizard - Step 4 of 6

What features would you like to include?

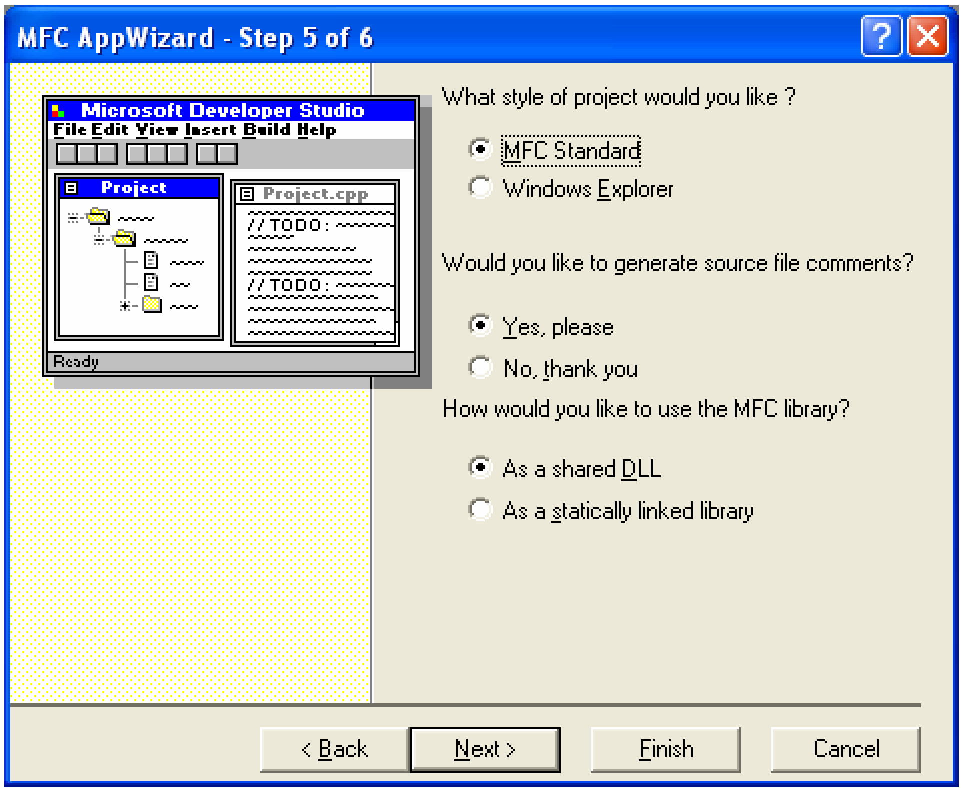
- [✓] Docking toolbar
- [✓] Initial status bar
- [ ] Printing and print preview
- [ ] Context-sensitive Help
- [✓] 3D controls
- [ ] MAPI (Messaging API)
- [ ] Windows Sockets

How do you want your toolbars to look?

[✓ Normal] Internet Explorer ReBars

How many files would you like on your recent file list? 4

Figure 9: Visual C++ AppWizard step 4 of 6.


MFC AppWizard - Step 5 of 6

What style of project would you like ?

[✓ MFC Standard] Windows Explorer

Would you like to generate source file comments?

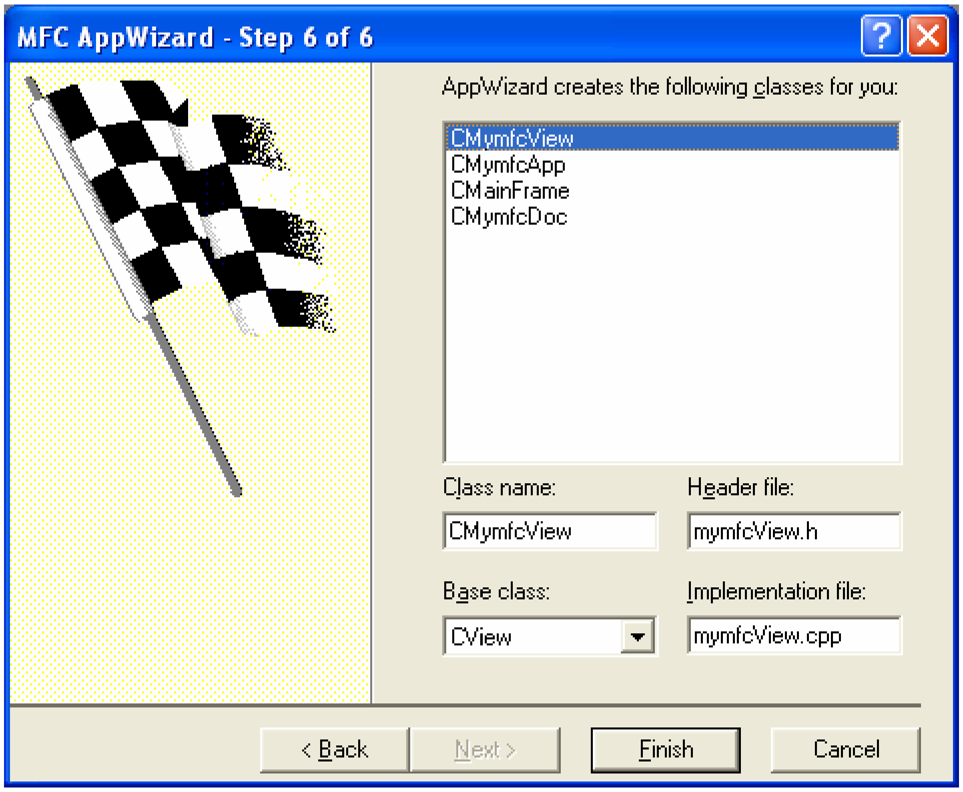
[✓ Yes, please] No, thank you

How would you like to use the MFC library?

[✓ As a shared DLL] As a statically linked library

Figure 10: Visual C++ AppWizard step 5 of 6.

Notice that the class names and source-file names have been generated based on the project name MYMFC. You could make changes to these names at this point if you wanted to. Click the **Finish** button.


MFC AppWizard - Step 6 of 6

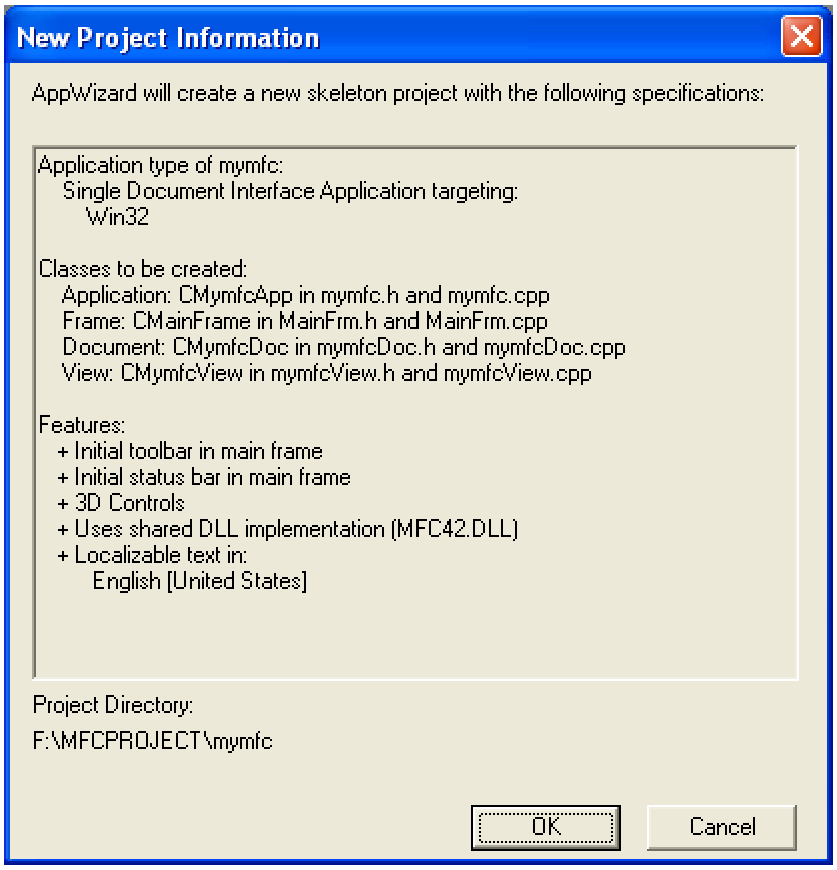
AppWizard creates the following classes for you:

CMymfcView
CMymfcApp
CMainFrame
CMymfcDoc

Class name: CMymfcView
Header file: mymfcView.h
Base class: CView
Implementation file: mymfcView.cpp

Figure 11: Visual C++ AppWizard step 6 of 6.

Just before AppWizard generates your code, it displays the **New Project Information** dialog box, shown here.



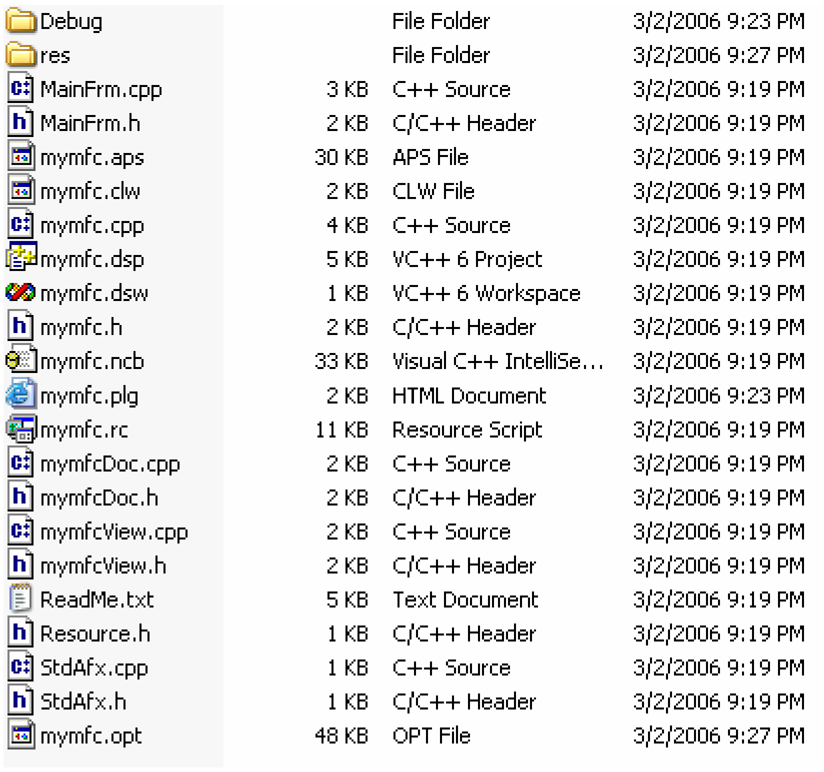
AppWizard will create a new skeleton project with the following specifications:

```
Application type of mymfc:
    Single Document Interface Application targeting:
        Win32

Classes to be created:
    Application: CMymfcApp in mymfc.h and mymfc.cpp
    Frame: CMainFrame in MainFrm.h and MainFrm.cpp
    Document: CMymfcDoc in mymfcDoc.h and mymfcDoc.cpp
    View: CMymfcView in mymfcView.h and mymfcView.cpp

Features:
   + Initial toolbar in main frame
   + Initial status bar in main frame
   + 3D Controls
   + Uses shared DLL implementation (MFC42.DLL)
   + Localizable text in:
         English [United States]
```

Project Directory:
F:\MFCPROJECT\mymfc

Figure 12: Visual C++ AppWizard project summary information.

When you click the **OK** button, AppWizard begins to create your application's subdirectory (**mymfc** under \mfcproject\mymfc) and a series of files in that subdirectory. When AppWizard is finished, look in the application's subdirectory. Physically, the following files are of interest (for now).



Debug		File Folder	3/2/2006 9:23 PM
res		File Folder	3/2/2006 9:27 PM
MainFrm.cpp	3 KB	C++ Source	3/2/2006 9:19 PM
MainFrm.h	2 KB	C/C++ Header	3/2/2006 9:19 PM
mymfc.aps	30 KB	APS File	3/2/2006 9:19 PM
mymfc.clw	2 KB	CLW File	3/2/2006 9:19 PM
mymfc.cpp	4 KB	C++ Source	3/2/2006 9:19 PM
mymfc.dsp	5 KB	VC++ 6 Project	3/2/2006 9:19 PM
mymfc.dsw	1 KB	VC++ 6 Workspace	3/2/2006 9:19 PM
mymfc.h	2 KB	C/C++ Header	3/2/2006 9:19 PM
mymfc.ncb	33 KB	Visual C++ IntelliSe...	3/2/2006 9:19 PM
mymfc.plg	2 KB	HTML Document	3/2/2006 9:23 PM
mymfc.rc	11 KB	Resource Script	3/2/2006 9:19 PM
mymfcDoc.cpp	2 KB	C++ Source	3/2/2006 9:19 PM
mymfcDoc.h	2 KB	C/C++ Header	3/2/2006 9:19 PM
mymfcView.cpp	2 KB	C++ Source	3/2/2006 9:19 PM
mymfcView.h	2 KB	C/C++ Header	3/2/2006 9:19 PM
ReadMe.txt	5 KB	Text Document	3/2/2006 9:19 PM
Resource.h	1 KB	C/C++ Header	3/2/2006 9:19 PM
StdAfx.cpp	1 KB	C++ Source	3/2/2006 9:19 PM
StdAfx.h	1 KB	C/C++ Header	3/2/2006 9:19 PM
mymfc.opt	48 KB	OPT File	3/2/2006 9:27 PM

Figure 13: mymfc physical project's files.

Some explanation for those files.

File	Description
`mymfc.dsp`	A project file that allows Visual C++ to build your application.
`mymfc.dsw`	A workspace file that contains a single entry for mymfc.dsp.
`mymfc.rc`	An ASCII resource script file.
`mymfcView.cpp`	A view class implementation file that contains `CMymfcView` class member functions.
`mymfcView.h`	A view class header file that contains the `CMymfcView` class declaration.
`mymfc.opt`	A binary file that tells Visual C++ which files are open for this project and how the windows are arranged (This file is not created until you save the project.).
`mymfcDoc.cpp`	A document class implementation file that contains `CMymfcDoc` class member functions.
`mymfcDoc.h`	A document class header file that contains the `CMymfcDoc` class declaration.
`ReadMe.txt`	A text file that explains the purpose of the generated files.
`resource.h`	A header file that contains `#define` constant definitions.

Table 2: Important Visual C++ project files.

Open the **mymfcView.cpp** and **mymfcView.h** files and look at the source code. Together, these files define the `CMymfcView` class, which is central to the application. An object of class `CMymfcView` corresponds to the application's view window, where all the "action" takes place.

Compile and link the generated code. AppWizard, in addition to generating code, creates custom project and workspace files for your application. The project file, **mymfc.dsp**, specifies all the file dependencies together with the compile and link option flags. Because the new project becomes Visual C++'s current project, you can now build the application by choosing **Build mymfc.exe** from the **Build** menu (or by clicking the **Build** toolbar button) shown here.

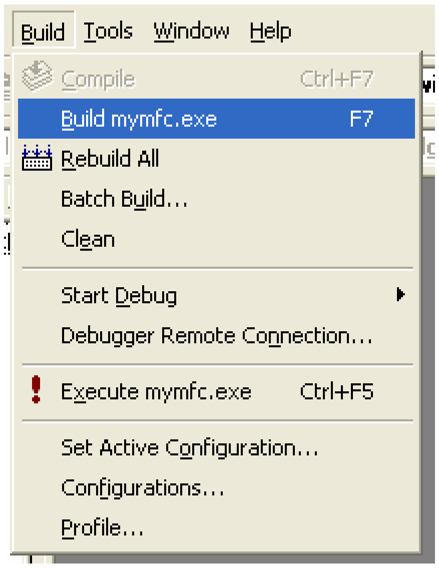
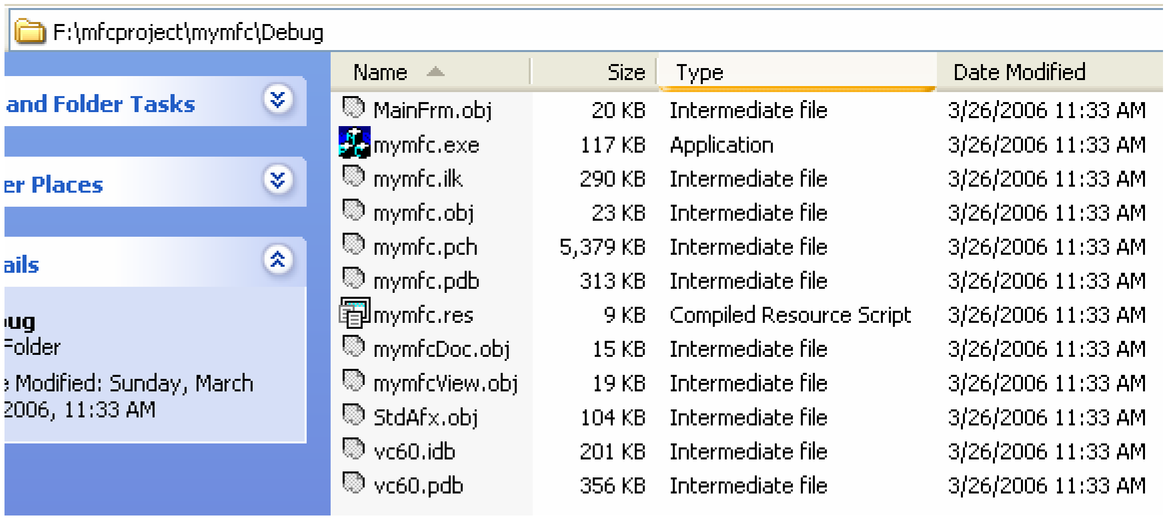


Figure 14: Building **mymfc.exe**.

If the build is successful, an executable program named **mymfc.exe** is created in a new **Debug** subdirectory underneath \mfcproject\mymfc. The OBJ files and other intermediate files are also stored in **Debug**.



Name	Size	Type	Date Modified
MainFrm.obj	20 KB	Intermediate file	3/26/2006 11:33 AM
mymfc.exe	117 KB	Application	3/26/2006 11:33 AM
mymfc.ilk	290 KB	Intermediate file	3/26/2006 11:33 AM
mymfc.obj	23 KB	Intermediate file	3/26/2006 11:33 AM
mymfc.pch	5,379 KB	Intermediate file	3/26/2006 11:33 AM
mymfc.pdb	313 KB	Intermediate file	3/26/2006 11:33 AM
mymfc.res	9 KB	Compiled Resource Script	3/26/2006 11:33 AM
mymfcDoc.obj	15 KB	Intermediate file	3/26/2006 11:33 AM
mymfcView.obj	19 KB	Intermediate file	3/26/2006 11:33 AM
StdAfx.obj	104 KB	Intermediate file	3/26/2006 11:33 AM
vc60.idb	201 KB	Intermediate file	3/26/2006 11:33 AM
vc60.pdb	356 KB	Intermediate file	3/26/2006 11:33 AM

Figure 15: Files generated after the building process under the Debug directory.

Compare the file structure on disk with the structure in the Workspace window's **FileView** page shown here. The FileView page contains a logical view of your project. The header files show up under **Header Files**, even though they are in the same subdirectory as the CPP files. The resource files are stored in the **\res** subdirectory.

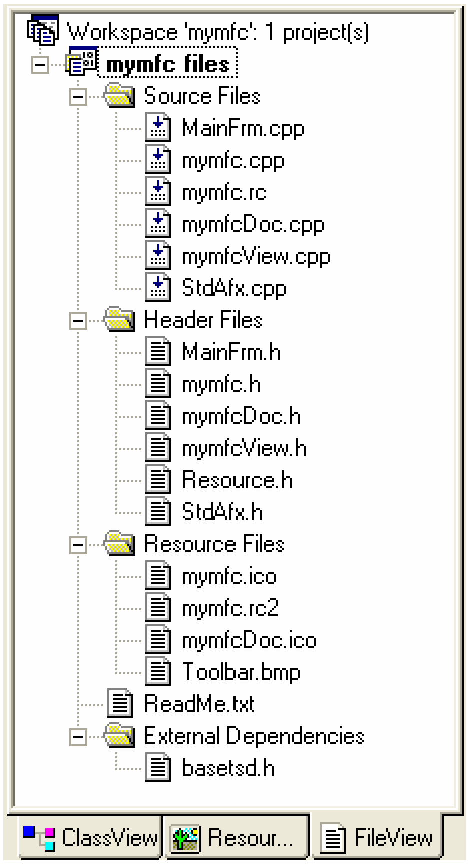


Figure 16: mymfc project FileView.

Test the resulting application. Choose **Execute mymfc.exe** from the **Build** menu. Experiment with the program. It doesn't do much because there is no coding yet. Actually, as you might guess, the program has a lot of features - you simply haven't activated them yet. Close the program window when you've finished experimenting.

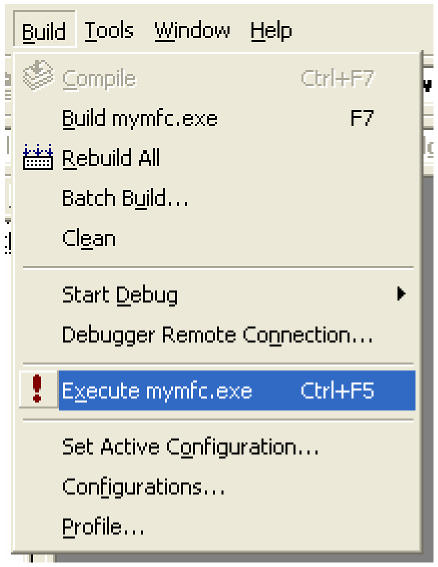


Figure 17: Executing/running the **mymfc.exe** program.

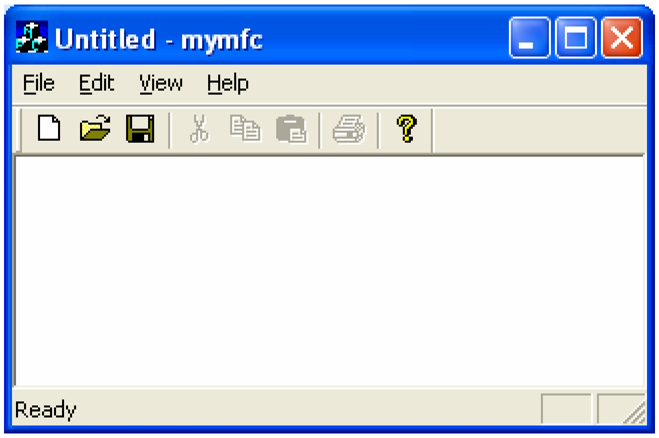


Figure 18: **mymfc** program output.

The most important `CMymfcView` base classes are `CWnd` and `CView`. `CWnd` provides `CMymfcView`'s "windowness," and `CView` provides the hooks to the rest of the application framework, particularly to the document and to the frame window.

Drawing Inside the View Window: The Windows Graphics Device Interface

Now you're ready to write code to draw inside the view window. You'll be making a few changes directly to the MYMFC source code.

The `OnDraw()` Member Function

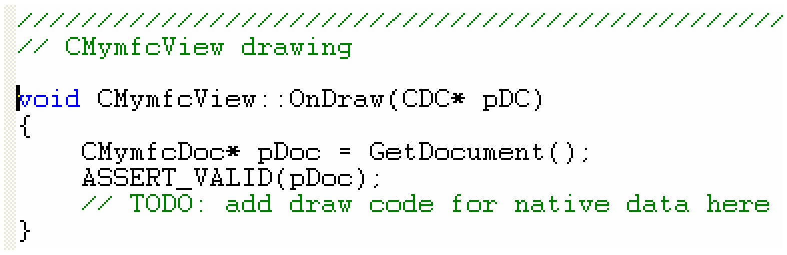
Specifically, you'll be fleshing out `OnDraw()` in mymfcView.cpp. `OnDraw()` is a virtual member function of the `CView` class that the application framework calls every time the view window needs to be repainted. A window needs to be **repainted** if the user resizes the window or reveals a previously 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. This call to `Invalidate()` triggers a later call to `OnDraw()`.
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

Windows doesn't allow direct access to the display hardware but communicates through an abstraction called a "**device context**" that is associated with the window. 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.

Adding Draw Code to the MYMFC Program

Now let's write the code to draw some text and a circle inside the view window. Be sure that the project MYMFC is open in Visual C++. You can use the Workspace window's ClassView to locate the code for the function and double-clicking the file to edit the `OnDraw()` function in mymfcView.cpp. So, find the AppWizard-generated `OnDraw()` function in mymfcView.cpp:



```
//////////////////////////////////////////////////////////
// CMymfcView drawing

void CMymfcView::OnDraw(CDC* pDC)
{
	CMymfcDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
	// TODO: add draw code for native data here
}
```

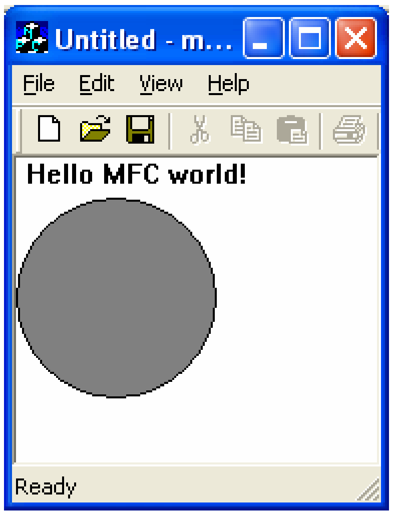
Listing 1: mymfc C++ code segment.

The following code replaces the previous code:

```
	void CMymfcView::OnDraw(CDC* pDC)
	{
	       // TODO: add draw code for native data here
	        // prints in default font and size, top left corner
	        pDC->TextOut(5, 0, "Hello MFC world!");
	        // selects a brush for the circle interior
	        pDC->SelectStockObject(GRAY_BRUSH);
	        // draws a gray circle 100 units in diameter
	        pDC->Ellipse(CRect(0, 20, 100, 120));
	}
```

You can safely remove the call to `GetDocument()` because we're not dealing with documents yet. The functions `TextOut()`, `SelectStockObject()` and `Ellipse()` are all member functions of the application framework's device context class `CDC`. The `Ellipse()` function draws a circle if the bounding rectangle's length is equal to its width. The MFC library provides a handy utility class, `CRect`, for Windows rectangles. A temporary `CRect` object serves as the bounding rectangle argument for the ellipse drawing function.

Recompile and test MYMFC. Choose **Build mymfc.exe** sub menu from **Build** menu and if there are no compile errors, test the application again. Now you have a program that visibly does something!



```
Hello MFC world!
```

Figure 19: mymfc program output.

A Preview of the Resource Editors

Now that you have a complete application program, it's a good time for a quick look at the resource editors. Although the application's resource script, mymfc.rc, is an ASCII file, modifying it with a text editor is not a good idea. That's the resource editors' job.

The Contents of `mymfc.rc`

The resource file determines much of the MYMFC application's "look and feel." The file **mymfc.rc** contains (or points to) the Windows resources listed here. You can open the rc file with text editor.

Resource	Description
Accelerator	Definitions for keys that simulate menu and toolbar selections.
Dialog	Layout and contents of dialog boxes. MYMFC has only the About dialog box.
Icon	Icons (16-by-16-pixel and 32-by-32-pixel versions), such as the application icon you see in Microsoft Windows Explorer and in the application's About dialog box. MYMFC uses the MFC logo for its application icon.
Menu	The application's top-level menu and associated pop-up menus.
String table	Strings that are not part of the C++ source code.
Toolbar	The row of buttons immediately below the menu.
Version	Program description, version number, language, and so on.

Table 3: Project resources.

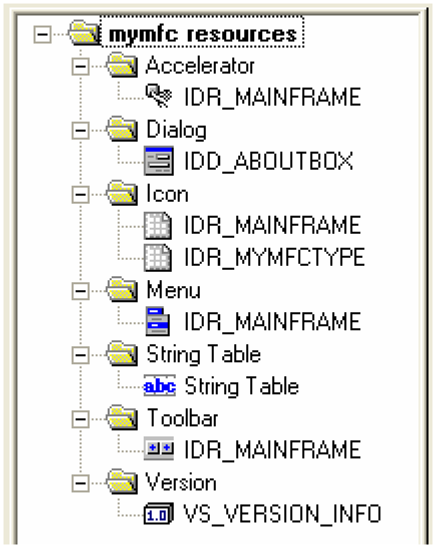
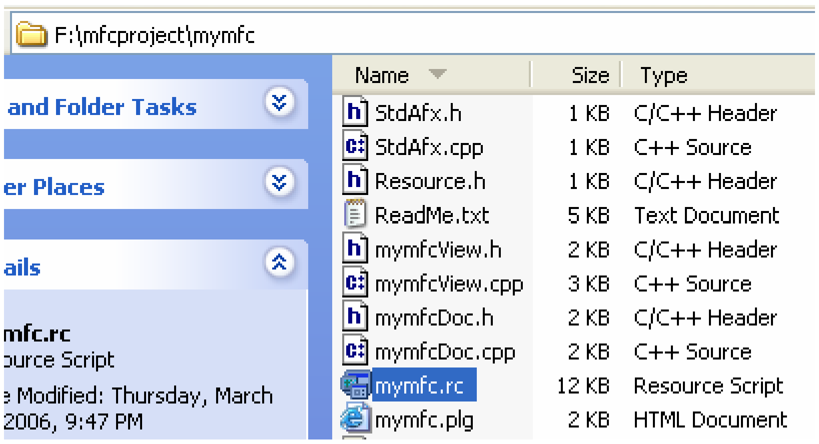


Figure 20: **mymfc** files in ResourceView.



Name	Size	Type
StdAfx.h	1 KB	C/C++ Header
StdAfx.cpp	1 KB	C++ Source
Resource.h	1 KB	C/C++ Header
ReadMe.txt	5 KB	Text Document
mymfcView.h	2 KB	C/C++ Header
mymfcView.cpp	3 KB	C++ Source
mymfcDoc.h	2 KB	C/C++ Header
mymfcDoc.cpp	2 KB	C++ Source
mymfc.rc	12 KB	Resource Script
mymfc.plg	2 KB	HTML Document

Figure 21: **mymfc** resource file (RC).

In addition to the resources listed above, mymfc.rc contains the statements:

```
#include  "afxres.h"
```

```
#include  "afxres.rc"
```

Which bring in some MFC library resources common to all applications. These resources include strings, graphical buttons, and elements needed for printing and OLE. If you're using the shared DLL version of the MFC library, the common resources are stored inside the MFC DLL. The mymfc.rc file also contains the statement:

```
#include  "resource.h"
```

This statement brings in the application's three `#define` constants, which are:

Object ID	Meaning
IDR_MAINFRAME	Identifying the menu, icon, string list, and accelerator table.
IDR_MYMFCTYPE	Identifying the default document icon, which we won't use in this program.
IDD_ABOUTBOX	Identifying the About dialog box.

Table 4.

This same resource.h file is included indirectly by the application's source code files. If you use a resource editor to add more constants (symbols), the definitions ultimately show up in **resource.h**. Be careful if you edit this file in text mode because your changes might be removed the next time you use a resource editor.

Running the Dialog Resource Editor

Open the project's RC file. Click the **ResourceView** button in the Workspace window. If you expand each item, you will see the following in the resource editor window.

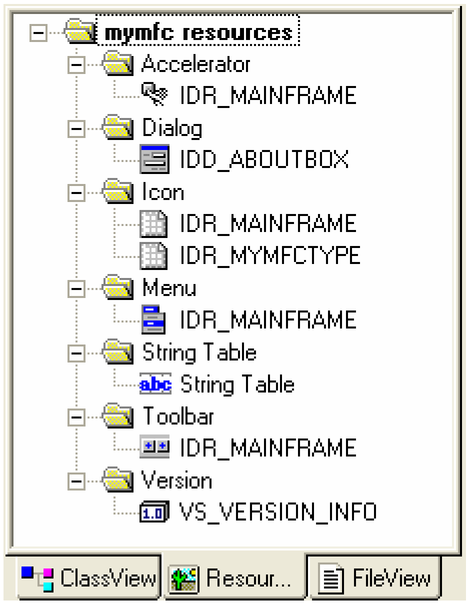


Figure 22: **mymfc**'s resources in ResourceView.

Examine the application's resources. Now take some time to explore the individual resources. When you select a resource by double-clicking on it, another window opens with tools appropriate for the selected resource. If you open a dialog resource, the control palette should appear. If it doesn't, right-click inside any toolbar, and then check **Controls**.

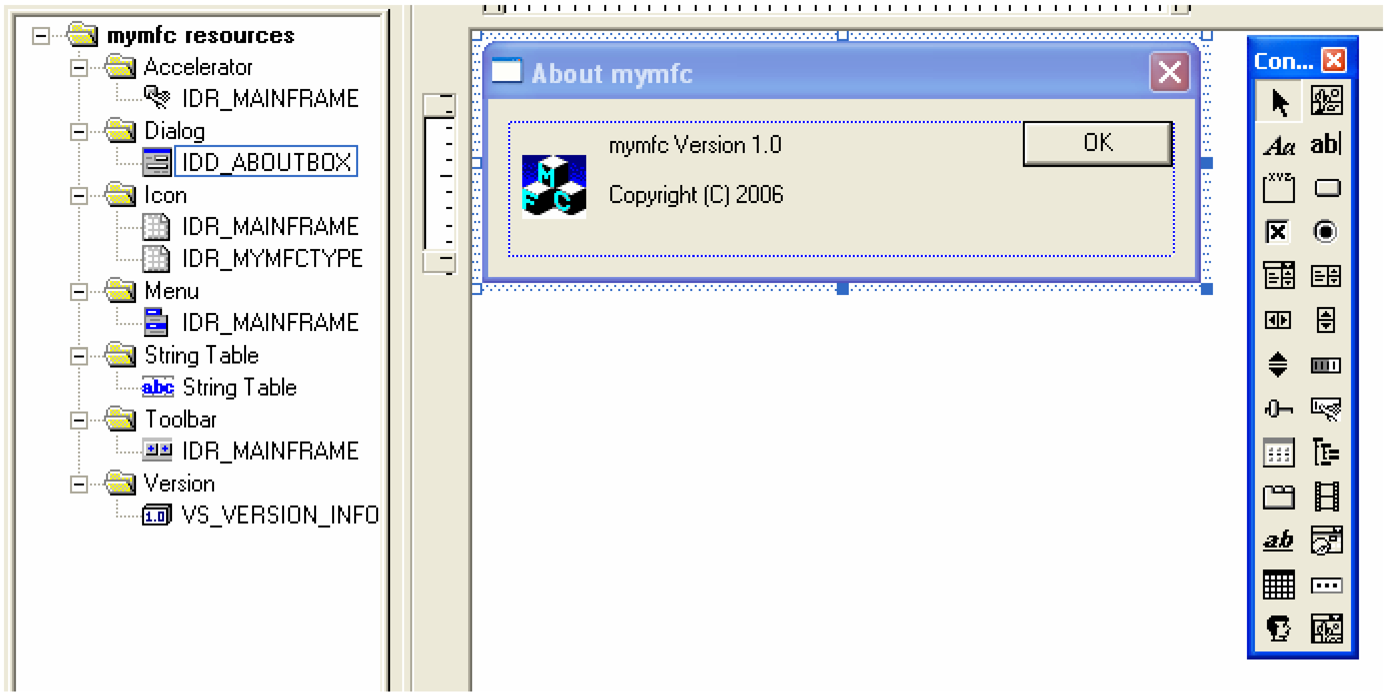
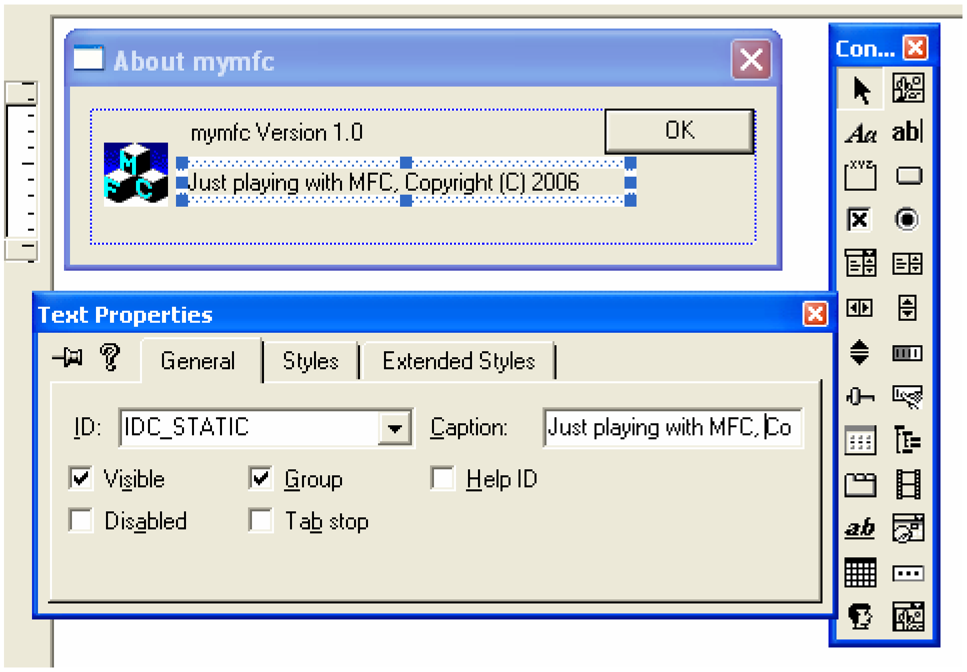


Figure 23: `IDD_ABOUTBOX` dialog in resource editor.

Modify the `IDD_ABOUTBOX` dialog box by making some changes to the **About mymfc** dialog box, shown here.



ID: IDC_STATIC
Caption: Just playing with MFC, Co
[✓] Visible
[✓] Group
[ ] Help ID
[ ] Disabled
[ ] Tab stop

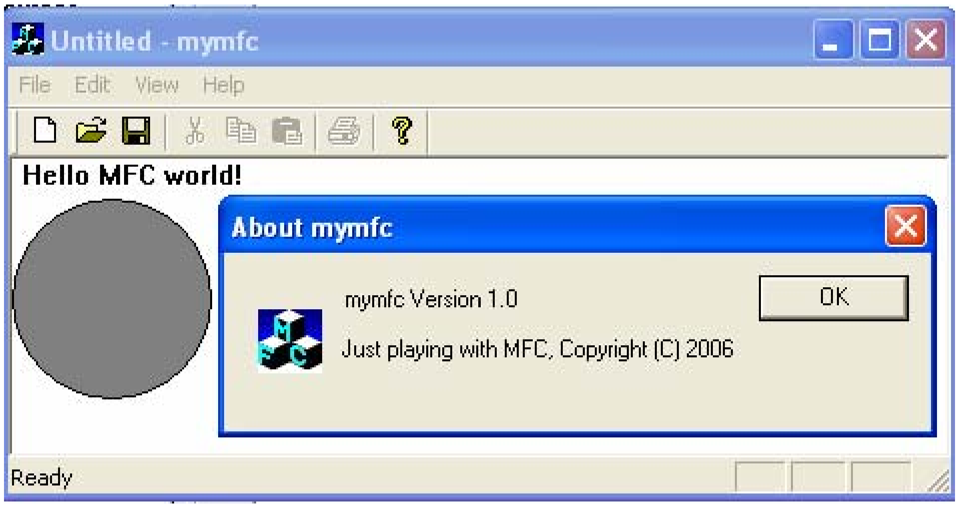
Figure 24: Changing the **About** dialog properties.

You can change the size of the window by dragging the right and bottom borders, move the OK button, change the text, and so forth. Simply click on an element to select it, and then right-click to change its properties.

Rebuild the project with the modified resource file. In Visual C++, choose **Build mymfc.exe** from the **Build** menu. Notice that no actual C++ recompilation is necessary. Visual C++ saves the edited resource file, and then the **Resource**

Compiler (`rc.exe`) processes **mymfc.rc** to produce a compiled version, **mymfc.res**, which is fed to the linker. The linker runs quickly because it can link the project incrementally.

Test the new version of the application. Run the MYMFC program again, and then choose **About** from the application's **Help** menu to confirm that your dialog box was changed as expected.



mymfc Version 1.0

Just playing with MFC, Copyright (C) 2006

Figure 25: **mymfc** program output.

Win32 Debug Target vs. Win32 Release Target

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.

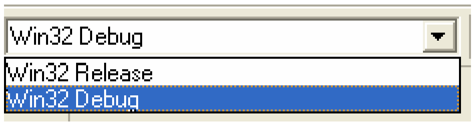


Figure 26: Release vs Debug target.

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)

Table 5.

You develop your application in **Debug mode**, and then you rebuild in **Release mode** prior to delivery. The Release build EXE will be smaller and faster because of the optimization and assuming that you have fixed all the bugs. You select the configuration from the build target window in the **Build** toolbar or **Build** minibar.

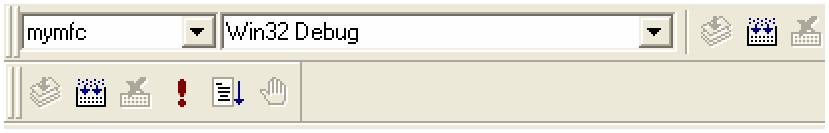
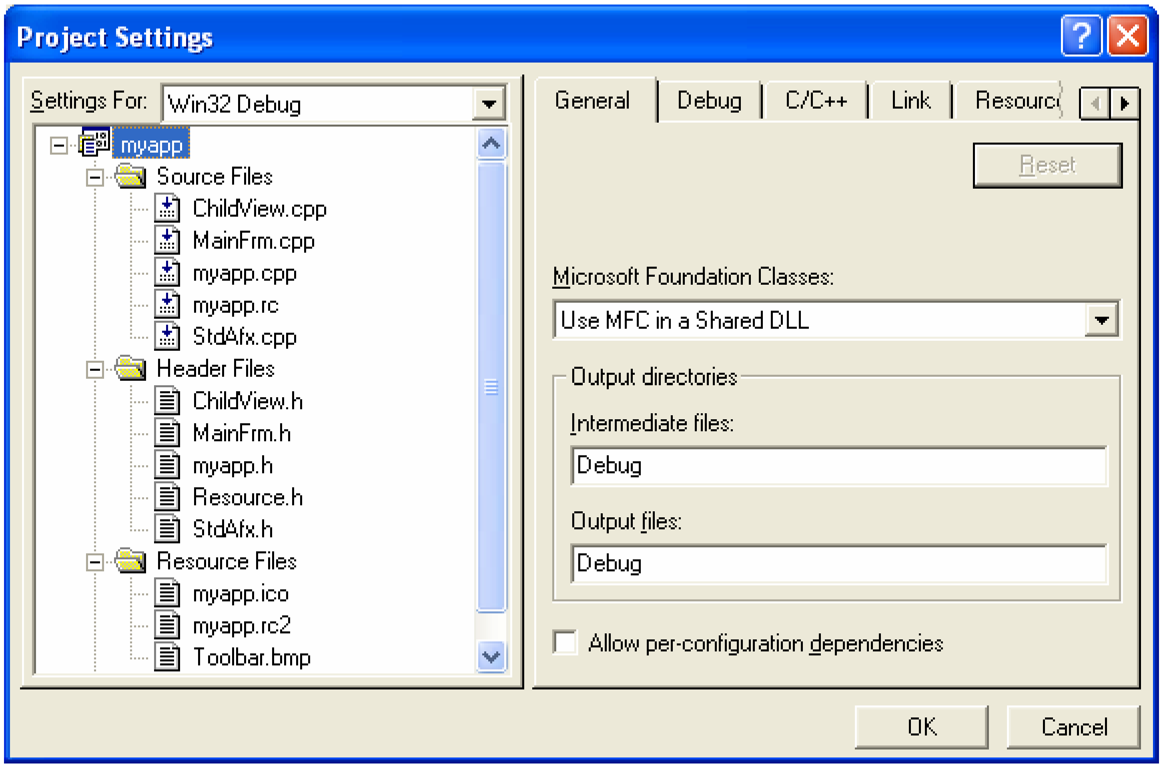


Figure 27: Build toolbar and minibar.

By default, the Debug output files and intermediate files are stored in the project's Debug subdirectory; the Release files are stored in the Release subdirectory. You can change these directories (and other project settings as well) on the **General** tab in the **Project Settings** dialog box.



Settings For: Win32 Debug

```
myapp
  Source Files
    ChildView.cpp
    MainFrm.cpp
    myapp.cpp
    myapp.rc
    StdAfx.cpp
  Header Files
    ChildView.h
    MainFrm.h
    myapp.h
    Resource.h
    StdAfx.h
  Resource Files
    myapp.ico
    myapp.rc2
    Toolbar.bmp
```

Microsoft Foundation Classes: Use MFC in a Shared DLL

Output directories

Intermediate files: Debug

Output files: Debug

[ ] Allow per-configuration dependencies

Figure 28: Visual C++ project settings.

You can create your own custom configurations if you need to by choosing **Configurations** from Visual C++'s **Build** menu.

Enabling the Diagnostic Macros

The application framework TRACE macros are particularly useful for monitoring program activity. They require that tracing be enabled, which is the default setting. If you're not seeing TRACE output from your program, first make sure that you are running the debug target from the debugger and then run the TRACER utility. If you check the `Enable Tracing` checkbox, TRACER will insert the statement:

```
        TraceEnabled = 1
```

in the `[Diagnostics]` section of a file named **Afx.ini**. (No, it's not stored in the Registry.) You can also use TRACER to enable other MFC diagnostic outputs, including message, OLE, database, and Internet information.

AFX Functions

Not all of the functions that MFC offers are members of classes. MFC provides an API of sorts all its own in the form of global functions whose names begin with **Afx**. Class member functions can be called only in the context of the objects to which they belong, but AFX functions are available anytime and anywhere.
The following table lists some of the more commonly used AFX functions. `AfxBeginThread()` simplifies the process of creating threads of execution. `AfxMessageBox()` is the global equivalent of the Windows `MessageBox()` function and, unlike `CWnd::MessageBox`, can be called just as easily from a document class as from a window class. `AfxGetApp()` and `AfxGetMainWnd()` return pointers to the application object and the

application's main window and are useful when you want to access a function or data member of those objects but don't have a pointer readily available. `AfxGetInstanceHandle()` is handy when you need an instance handle to pass to a Windows API function. Even MFC programs call API functions every now and then! The following Table list the commonly use Afx functions.

Function Name	**Description**
`AfxAbort()`	Unconditionally terminates an application; usually called when an unrecoverable error occurs.
`AfxBeginThread()`	Creates a new thread and begins executing it.
`AfxEndThread()`	Terminates the thread that is currently executing.
`AfxMessageBox ()`	Displays a Windows message box.
`AfxGetApp()`	Returns a pointer to the application object.
`AfxGetAppName()`	Returns the name of the application.
`AfxGetMainWnd()`	Returns a pointer to the application's main window.
`AfxGetInstanceHandle()`	Returns a handle identifying the current application instance.
`AfxRegisterWndClass()`	Registers a custom `WNDCLASS` for an MFC application.

Table 6

Precompiled Headers Story

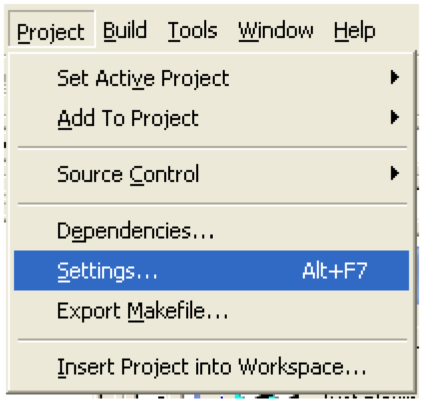
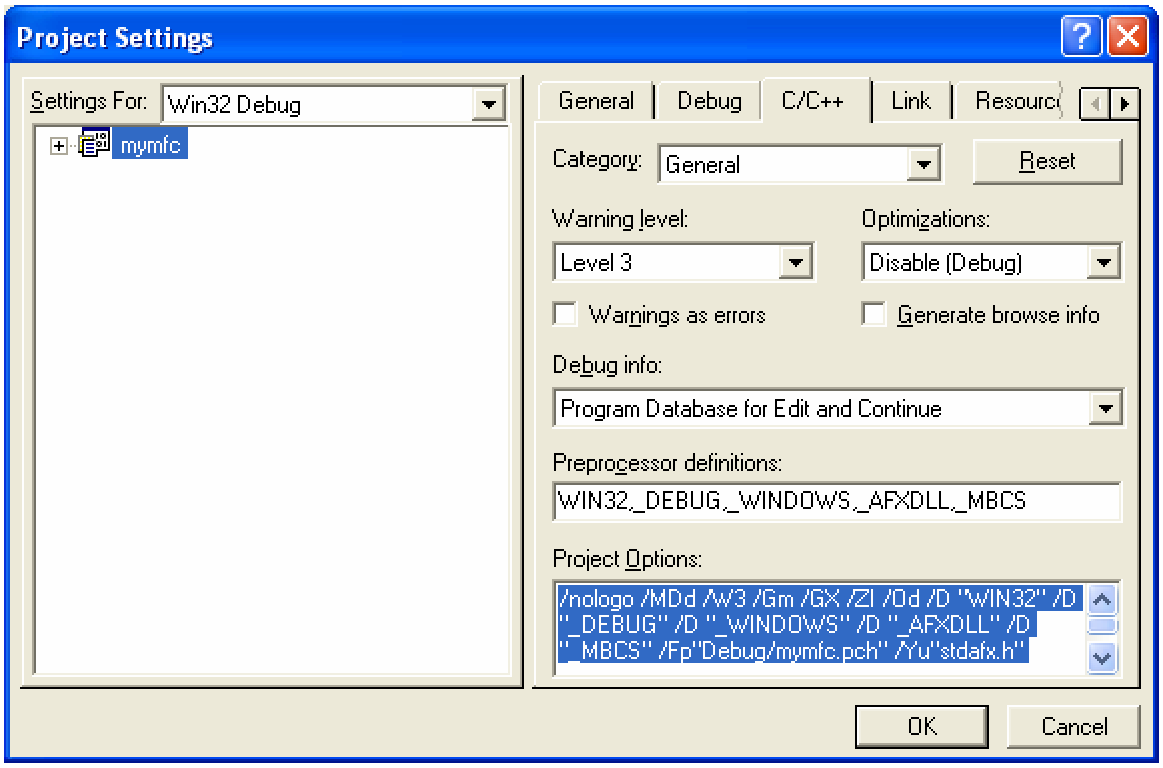


Figure 29: Invoking the Visual C++ project settings dialog.



Settings For: Win32 Debug

mymfc

Category: General

Warning level: Level 3

Optimizations: Disable (Debug)

Warnings as errors

Generate browse info

Debug info: Program Database for Edit and Continue

Preprocessor definitions: WIN32,_DEBUG,_WINDOWS,_AFXDLL,_MBCS

Project Options:

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D
"_DEBUG" /D "_WINDOWS" /D "_AFXDLL" /D
"_MBCS" /Fp"Debug/mymfc.pch" /Yu"stdafx.h"
```

Figure 30: Visual C++ project settings dialog.

When AppWizard generates a project, it generates switch settings and files for precompiled headers. You must understand how the make system processes precompiled headers in order to manage your projects effectively. Visual C++ has two precompiled header "systems:" automatic and manual. Automatic precompiled headers, activated with the **/Yx** compiler switch, store compiler output in a "database" file. Manual precompiled headers are activated by the **/Yc** and **/Yu** switch settings and are central to all AppWizard-generated projects. Precompiled headers represent compiler "snapshots" taken at a particular line of source code. In MFC library programs, the snapshot is generally taken immediately after the following statement:

```
#include  "StdAfx.h"
```

The file `StdAfx.h` contains `#include` statements for the MFC library header files. The file's contents depend on the options that you select when you run AppWizard, but the file always contains these statements:

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

If you're using compound documents, `StdAfx.h` also contains the statement:

```
#include <afxole.h>
```

And if you're using Automation or ActiveX Controls, it contains:

```
#include <afxdisp.h>
```

If you're using Internet Explorer 4 Common Controls, StdAfx.h contains the statement:

```
#include <afxdtctl.h>
```

Occasionally you will need other header files - for example, the header for template-based collection classes that is accessed by the statement:

```
#include <afxtempl.h>
```

The source file StdAfx.cpp contains only the statement:

```
#include  "StdAfx.h"
```

And is used to generate the precompiled header file in the project directory. The MFC library headers included by **StdAfx.h** never change, but they do take a long time to compile. The compiler switch /Yc, used only with StdAfx.cpp, causes creation of the precompiled header (PCH) file. The switch /Yu, used with all the other source code files, causes use of an existing PCH file. The switch /Fp specifies the PCH filename that would otherwise default to the project name (with the PCH extension) in the target's output files subdirectory. Figure 31 illustrates the whole process.
AppWizard sets the /Yc and /Yu switches for you, but you can make changes if you need to. It's possible to define compiler switch settings for individual source files. On the C/C++ tab in the Project Settings dialog box, if you select only StdAfx.cpp, you'll see the /Yc setting. This overrides the /Yu setting that is defined for the target.
Be aware that PCH files are big - 5 MB is typical. If you're not careful, you'll fill up your hard disk. You can keep things under control by periodically cleaning out your projects' Debug directories, or you can use the /Fp compiler option to reroute PCH files to a common directory.

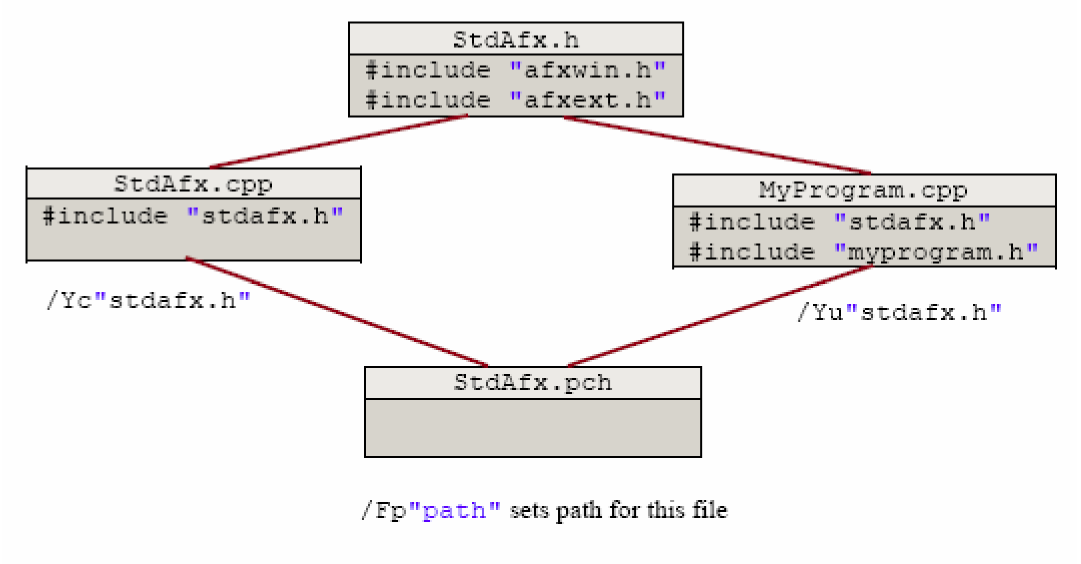


Figure 31: The Visual C++ precompiled header process.

Two Ways to Run a Program

Visual C++ lets you run your program directly (by pressing Ctrl-F5) or through the debugger (by pressing F5). Running your program directly is much faster because Visual C++ doesn't have to load the debugger first. If you know you don't want to see diagnostic messages or use breakpoints, start your program by pressing Ctrl-F5 or use the "exclamation point" button on the **Build** toolbar.

Further reading and digging:

1. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
2. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
3. MSDN Library
4. Windows data type.
5. Win32 programming Tutorial.
6. The best of C/C++, MFC, Windows and other related books.
7. Unicode and Multibyte character set: Story and program examples.