Module 1: Microsoft Windows, Visual C++ and Microsoft Foundation Class (MFC)

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:

Introduction

After you have completed the C++ and object oriented and Standard Template Library (STL) tutorials, hopefully you got the fundamental ideas how the classes were constructed and used and how the objects instantiated, how to organize multiple files in declaration, implementation, main program parts etc. You also were introduced the principles of the object encapsulation, inheritance and polymorphism. In inheritance you have learnt that child class can inherit the properties of the parent or base classes. You also have learnt how we send messages to get tasks done instead of the traditional manner using function call. The following Table lists important terms that we have used in the C++ Tutorial.

Term	Description
class	Is a group of data and methods (functions). A class is very much like a structure type as used in ANSI-C, it is just a type used to create a variable which can be manipulated through method in a program.
object	Is an instance of a class, which is similar to an analogous of a variable, defined as an instance of a type. An object is what you actually use in a program since it contains values and can be changed.
method	Is a function contained within the class. You will find the functions used within a class often referred to as methods in programming literature. Other similar term used is 'member function'.
message	Can be considered similar to function call. In object oriented programming, we send messages instead of calling functions. In programming terms, event or action of the object normally used to describe a consequence of sending

	message.

Table 1

And the following figure shows a very simple class hierarchy used in Tenouk's C++ Tutorial.

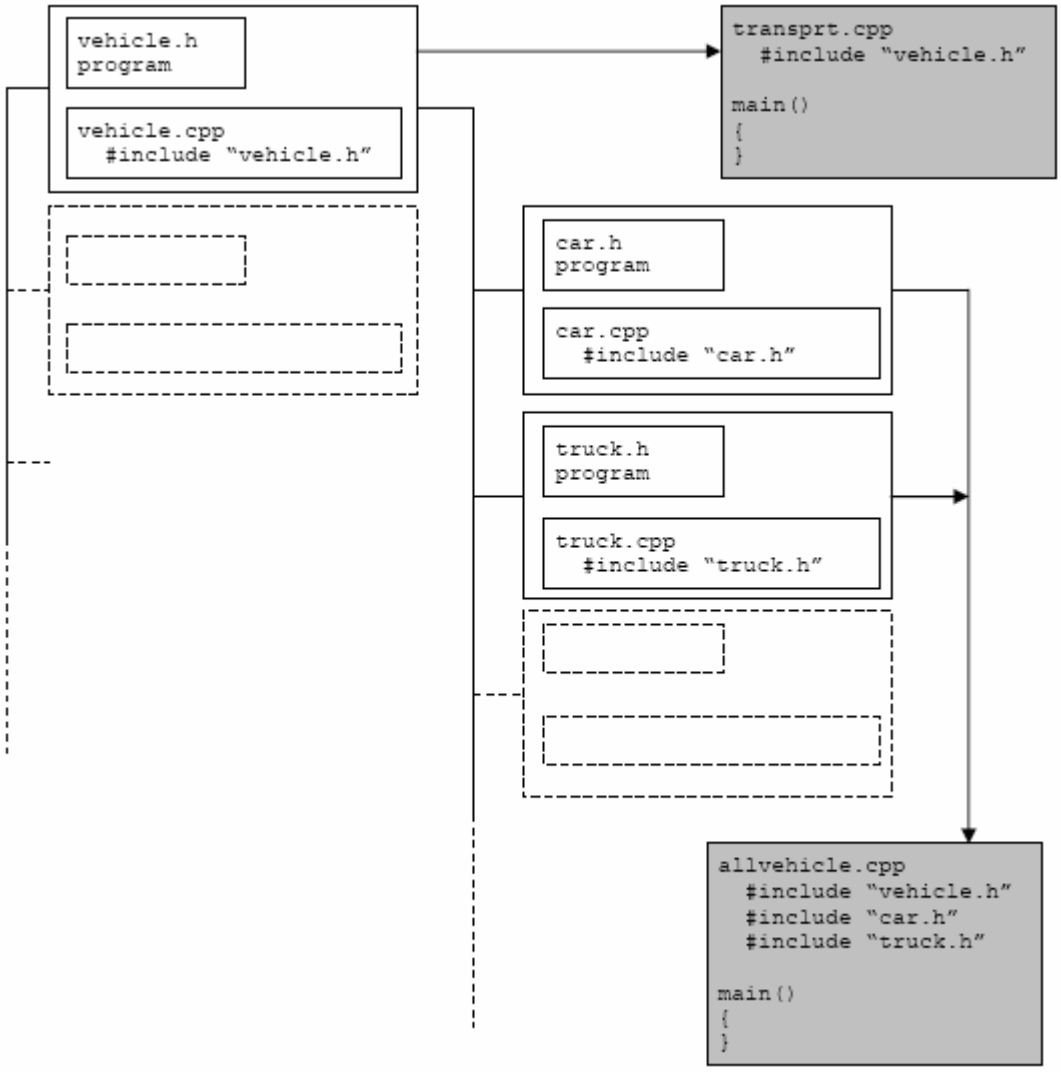


Figure 1: A very simple C++ Class hierarchy.

Though the previous Tutorials just provide the basics of the Object Oriented, but it is a very useful to grab the fundamentals in order to understand the more complex solutions such as Microsoft Foundation Class (MFC) or other object oriented programming such as Java and C#. Our next task is to explore the implementation specific of the classes in MFC. MFC contains a bunch of classes that ready to be used mainly for Windows graphic user interface programming. MFC is designed to be a great class library for creating graphically rich, sophisticated Windows applications. Before we go any further we will get some idea about the compiler, Visual C++ 6.0. Be prepared, you will find many object oriented terms, principles and many more program files in MFC programming.

The Visual C++ Components

You still can develop C-language Windows programs using only the Win32 API using Visual C++ as used in many C programming in Tenouk Tutorial. You can use many Visual C++ tools, including the resource editors, to make low-level Win32 programming easier. Visual C++ also includes the ActiveX Template Library (ATL), which you can use to develop ActiveX controls for the Internet. ATL programming is neither Win32 C-language programming nor MFC

programming. In this Tutorial we will concentrate the C++ and MFC programming using Visual C++ 6.0. Use of the MFC library programming interface doesn't cut you off from the Win32 functions. In fact, you'll almost always need some direct Win32 calls in your MFC library programs. A quick run-through of the Visual C++ components will help you get your bearings before you zero in on the application framework. Figure 2 shows an overview of the Visual C++ application build process.

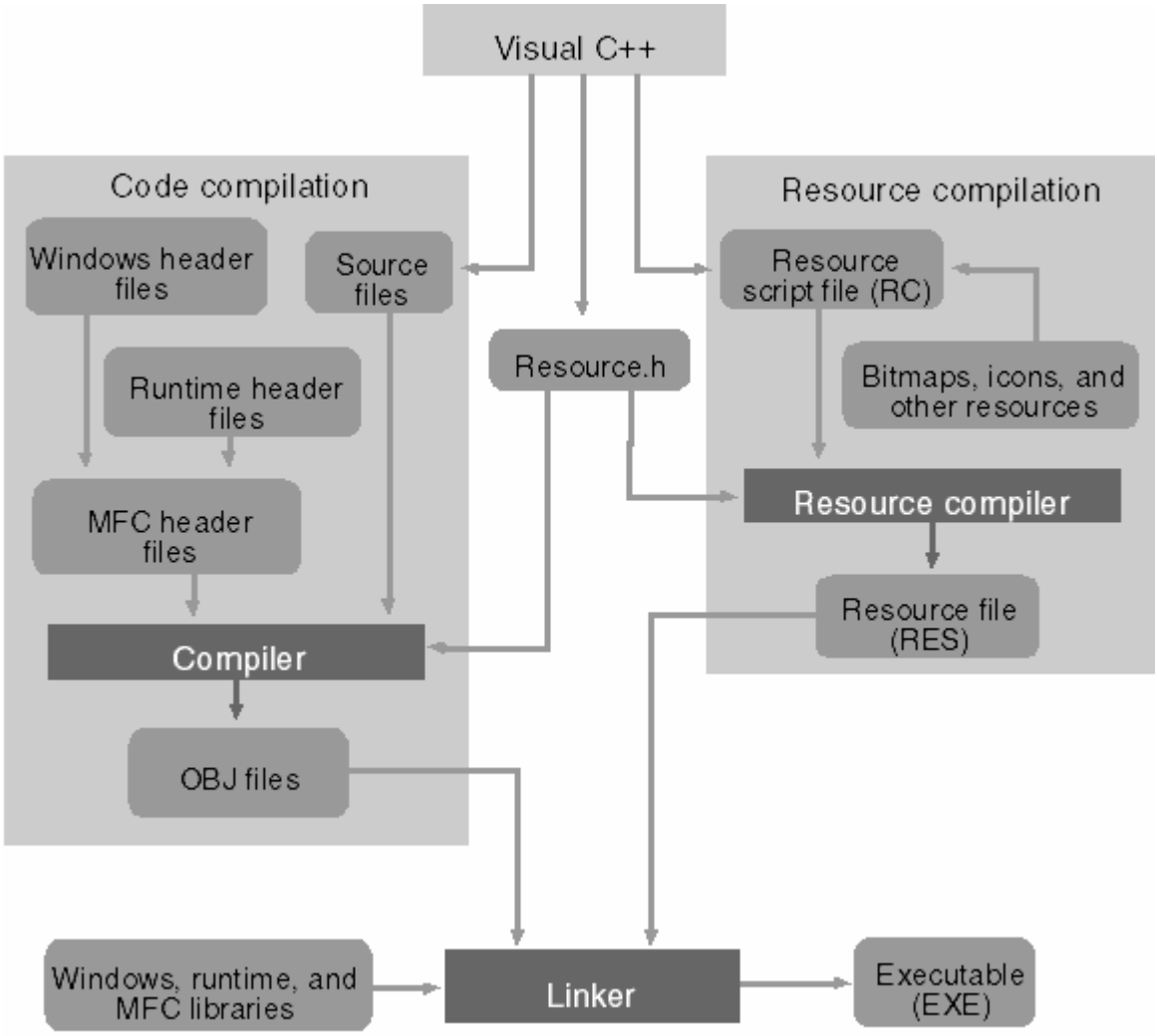


Figure 2: The Visual C++ application build process.

Microsoft Visual C++ 6.0 and the Build Process

Visual Studio 6.0 is a suite of developer tools that includes Visual C++ 6.0. The Visual C++ IDE is shared by several tools including Microsoft Visual J++. The IDE has come a long way from the original **Visual Workbench**, which was based on **QuickC** for Windows. Docking windows, configurable toolbars, plus a customizable editor that runs macros, are now part of Visual Studio. The online help system (now integrated with the MSDN Library viewer) works like a Web browser. Figure 3 shows Visual C++ 6.0 in action.

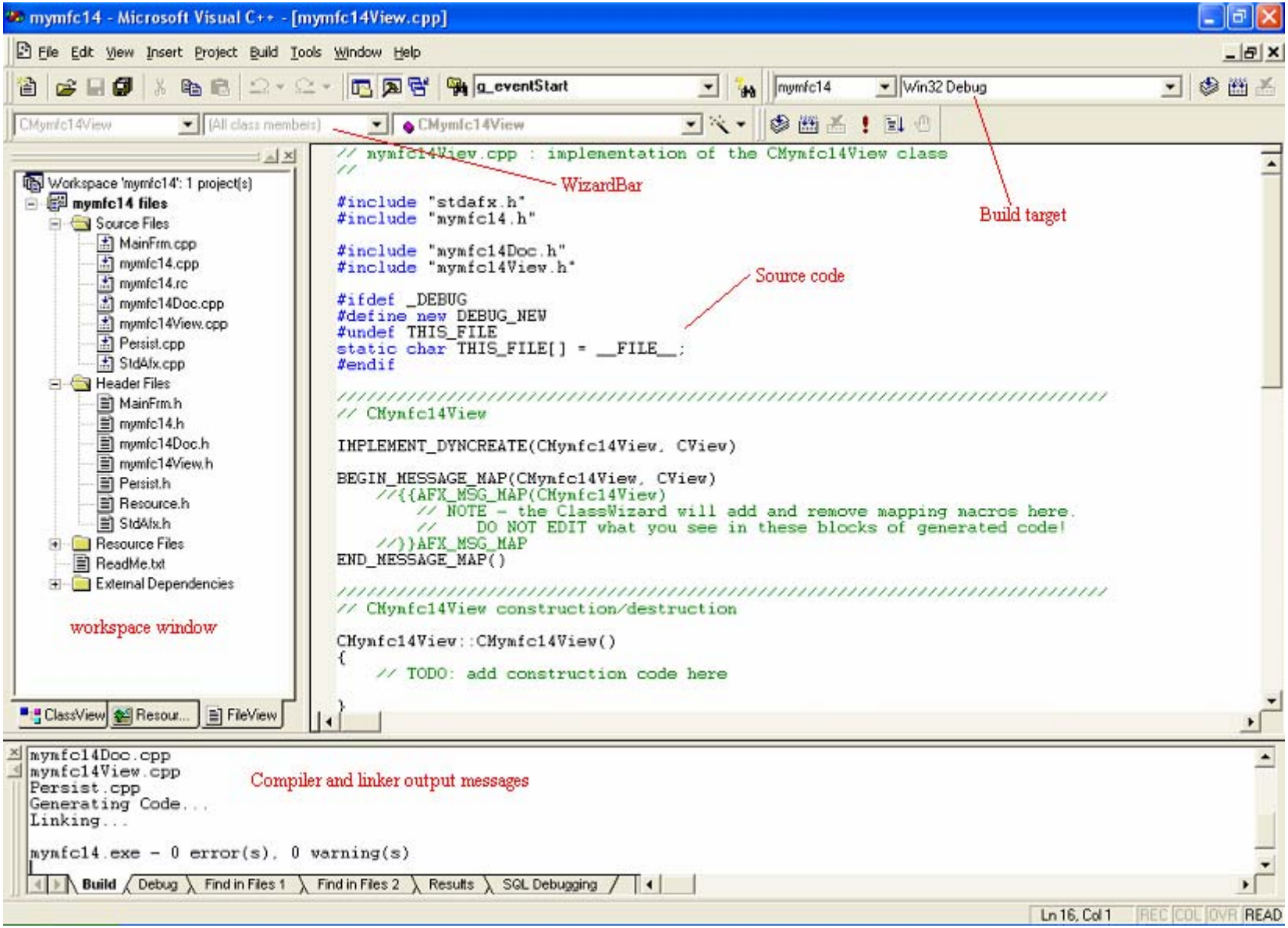

mymfc14 - Microsoft Visual C++ - [mymfc14View.cpp]
File Edit View Insert Project Build Tools Window Help
g_eventStart
mymfc14
Win32 Debug
CMymfc14View
(All class members)
CMymfc14View
WizardBar
Build target
Source code
Workspace 'mymfc14': 1 project(s)
mymfc14 files
Source Files
MainFrm.cpp
mymfc14.cpp
mymfc14.rc
mymfc14Doc.cpp
mymfc14View.cpp
Persist.cpp
StdAfx.cpp
Header Files
MainFrm.h
mymfc14.h
mymfc14Doc.h
mymfc14View.h
Persist.h
Resource.h
StdAfx.h
Resource Files
ReadMe.txt
External Dependencies
workspace window
ClassView
Resour...
FileView
// mymfc14View.cpp : implementation of the CMymfc14View class
//
#include "stdafx.h"
#include "mymfc14.h"
#include "mymfc14Doc.h"
#include "mymfc14View.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CMymfc14View
IMPLEMENT_DYNCREATE(CMymfc14View, CView)
BEGIN_MESSAGE_MAP(CMymfc14View, CView)
//{{AFX_MSG_MAP(CMymfc14View)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
// CMymfc14View construction/destruction
CMymfc14View::CMymfc14View()
{
// TODO: add construction code here
}
mymfc14Doc.cpp
mymfc14View.cpp
Persist.cpp
Generating Code...
Linking...
mymfc14.exe - 0 error(s), 0 warning(s)
Compiler and linker output messages
Build
Debug
Find in Files 1
Find in Files 2
Results
SQL Debugging
Ln 16, Col 1
REC COL OVR READ

Figure 3: Visual C++ 6.0 windows with main components displayed.

If you've used earlier versions of Visual C++ or another vendor's IDE, you already understand how Visual C++ 6.0 operates. But if you're new to IDEs, you'll need to know what a project is. A project is a collection of interrelated source files that are compiled and linked to make up an executable Windows-based program or a DLL. Source files for each project are generally stored in a separate subdirectory. A project depends on many files outside the project subdirectory too, such as include files and library files. 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. A make program 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.
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 **project file** (with a **DSP extension**) serves the same purpose. A separate text-format **workspace file** (with a **DSW extension**) has an **entry** for each project in the workspace. It's possible to have multiple projects in a workspace, but all the examples in this Tutorial have just one project per workspace. To work on an existing project, you tell Visual C++ to open the DSW file and then you can edit and build the project. The project directory also can be copied to other directory or system and can be edited, built and run as usual.

The Resource Editors: Workspace ResourceView

When you click on the **ResourceView** tab in the Visual C++ Workspace window as shown below, you can select a resource for editing by double clicking the resource such as the About dialog box shown below.

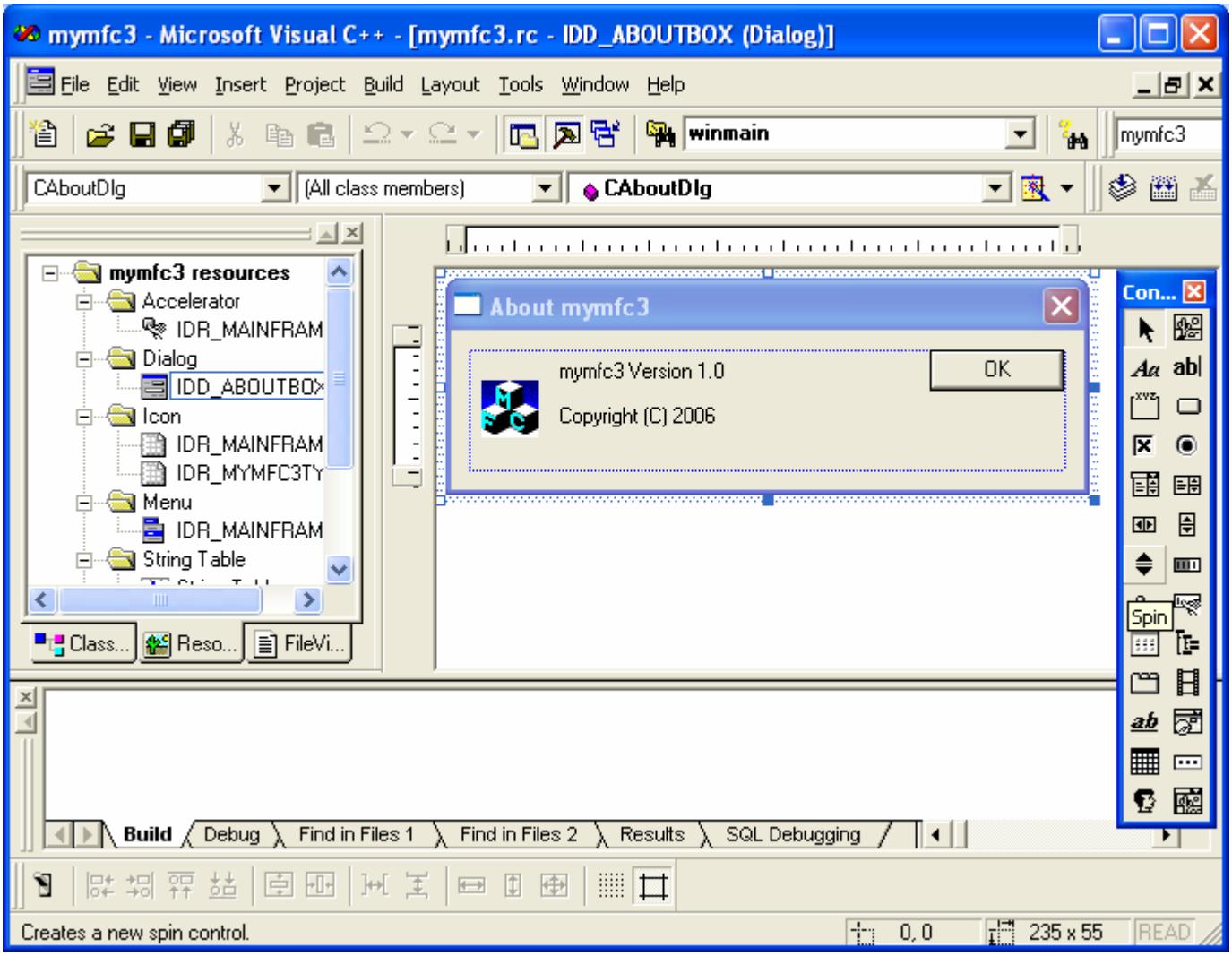


Figure 4: Visual C++ ResourceView.

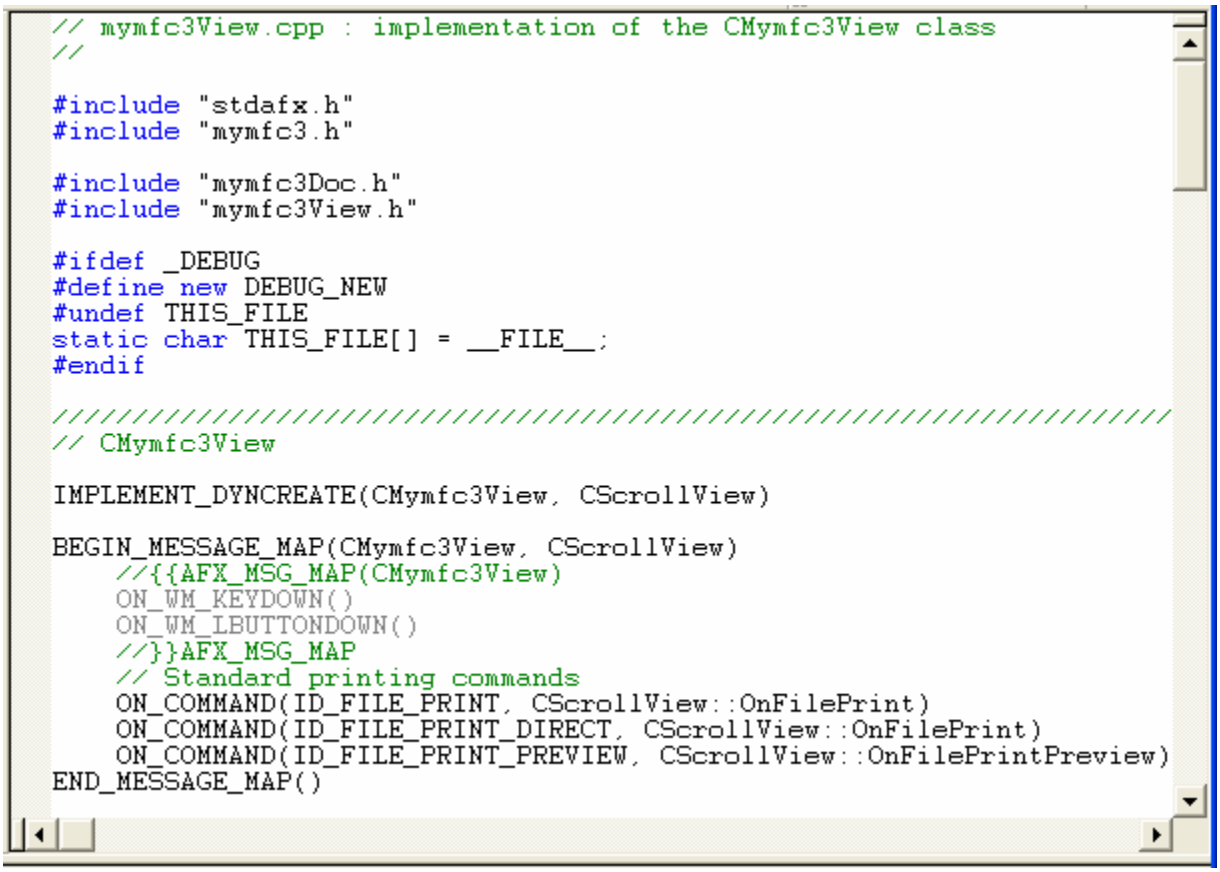
The main window hosts a resource editor appropriate for the resource type. The window can also host a WYSIWYG editor for menus and a powerful graphical editor for dialog boxes, and it includes tools for editing icons, bitmaps, and strings. The dialog editor allows you to insert ActiveX controls in addition to standard Windows controls and the new Windows common controls which have been further extended in Visual C++ 6.0. Each project usually has one text-format resource script (RC) file that describes the project's menu, dialog, string, and accelerator resources. The RC file also has `#include` statements to bring in resources from other subdirectories. These resources include project-specific items, such as bitmap (BMP) and icon (ICO) files, and resources common to all Visual C++ programs, such as error message strings. The resource editors can also process EXE and DLL files, so you can use the clipboard to "steal" resources, such as bitmaps and icons, from other Windows applications.

The C/C++ Compiler

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

The Source Code Editor

Visual C++ 6.0 includes a sophisticated source code editor that supports many features such as dynamic syntax coloring, auto-tabbing, keyboard bindings for a variety of popular editors (such as VI and EMACS), and pretty printing.



```
// mymfc3View.cpp : implementation of the CMymfc3View class
//

#include "stdafx.h"
#include "mymfc3.h"

#include "mymfc3Doc.h"
#include "mymfc3View.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMymfc3View

IMPLEMENT_DYNCREATE(CMymfc3View, CScrollView)

BEGIN_MESSAGE_MAP(CMymfc3View, CScrollView)
    //{{AFX_MSG_MAP(CMymfc3View)
    ON_WM_KEYDOWN()
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

Figure 5: Visual VC++ source code editor.

In Visual C++ 6.0, an exciting feature named **AutoComplete** has been added. If you have used any of the Microsoft Office products or Microsoft Visual Basic, you might already be familiar with this technology. Using the Visual C++ 6.0 AutoComplete feature, all you have to do is type the beginning of a programming statement and the editor will provide you with a list of possible completions to choose from. This feature is extremely handy when you are working with C++ objects and have forgotten an exact member function or data member name - they are all there in the list for you. You no longer have to memorize thousands of Win32 APIs or rely heavily on the online help system, thanks to this feature.

The Resource Compiler

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

The Linker

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

The Debugger

For the serious and real application developers, many times they will encounter problems in their codes. Unfortunately those problems, normally called bugs, could not be found through the code inspections or reviews. They need a

debugger to debug their programs to find those bugs. The Visual C++ debugger has been steadily improving, though it doesn't actually fix the bugs yet. The debugger works closely with Visual C++ to ensure that breakpoints are saved on disk. Toolbar buttons insert and remove breakpoints and control single-step execution. Figure 9 illustrates the Visual C++ debugger in action. Note that the Variables and Watch windows can expand an object pointer to show all data members of the derived class and base classes. If you position the cursor on a simple variable, the debugger shows you its value in a little window. To debug a program, you must build the program with the compiler and linker options set to generate debugging information. The debug session can be invoked through the **Build** menu and **Start Debug** sub menu shown below.

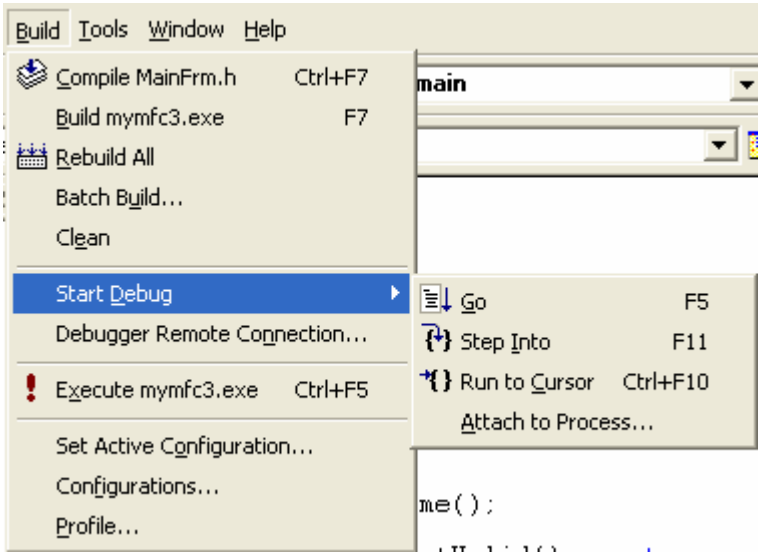


Figure 6: Visual C++ Debugging menu.

After the debug session been invoked, a new dynamic **Debu**g menu will be displayed as shown below.

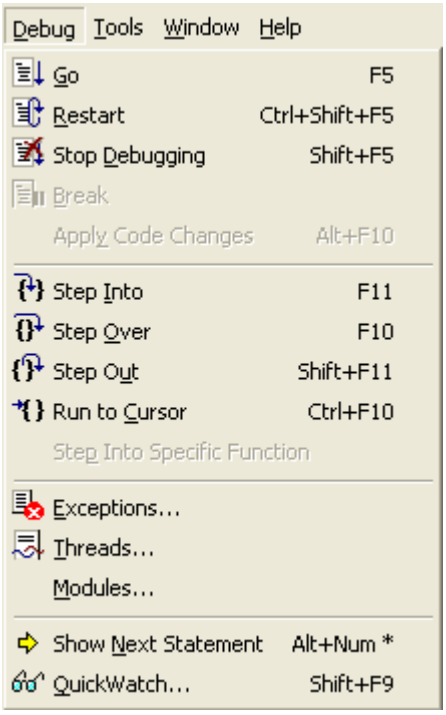


Figure 7: Visual C++ debug options.

Then you can view much more information through the **View** menu and **Debug Windows** sub menu.

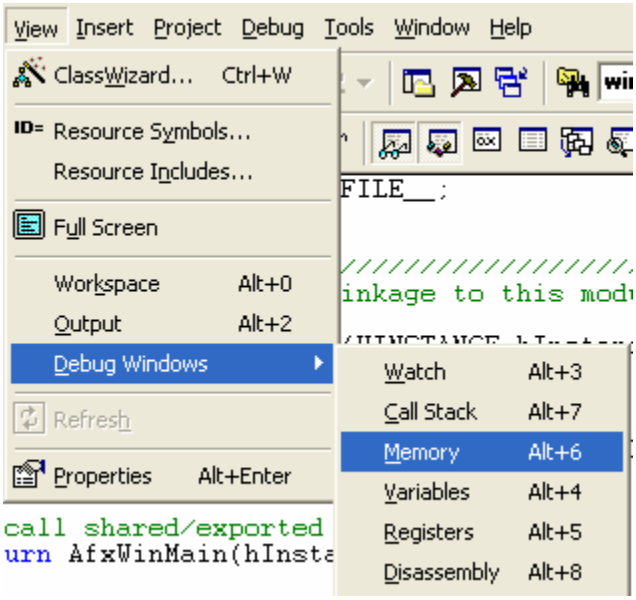


Figure 8: Visual C++ Debug window options.

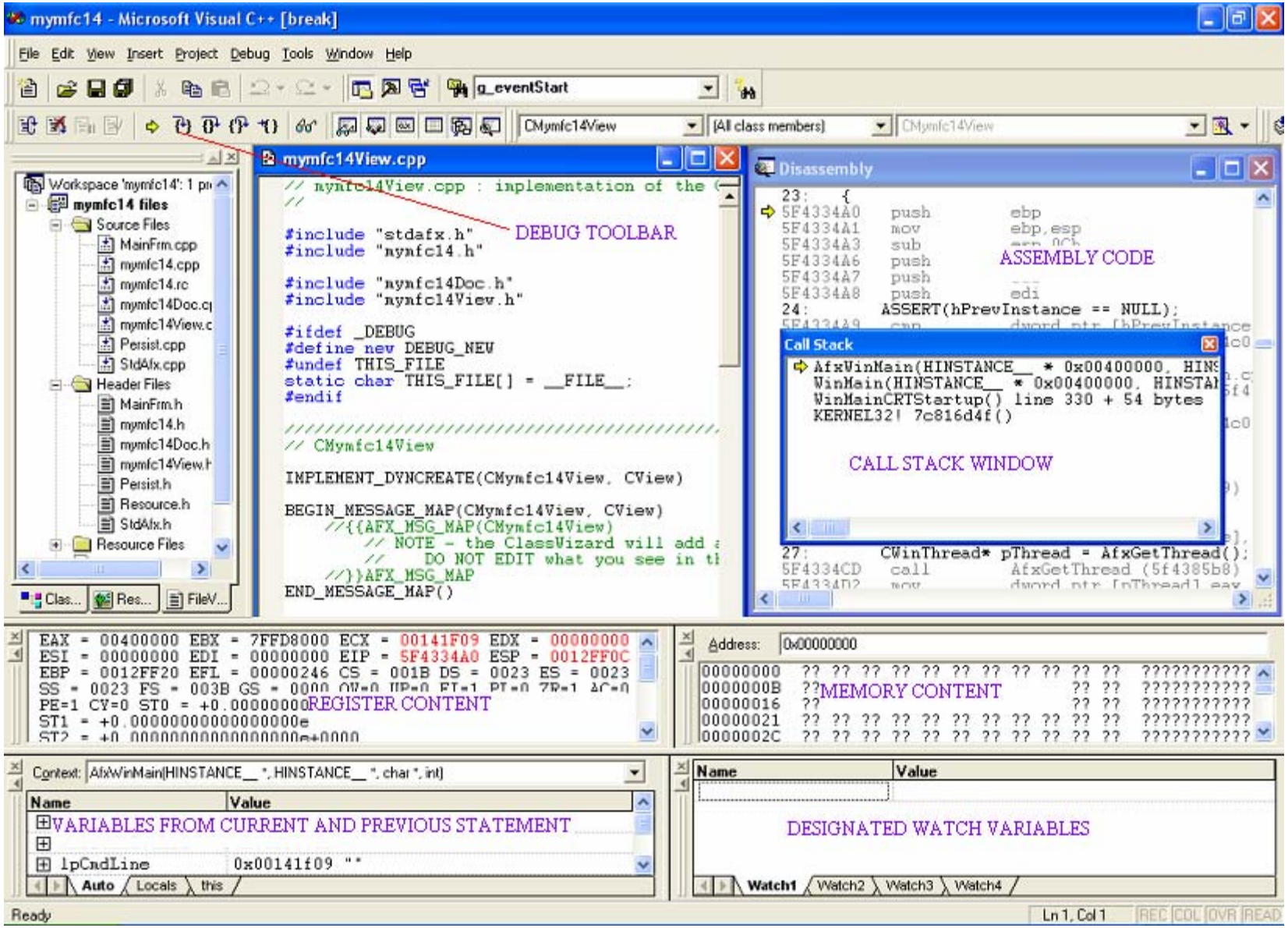

mymfc14 - Microsoft Visual C++ [break]
File Edit View Insert Project Debug Tools Window Help
mymfc14View.cpp
DEBUG TOOLBAR
Disassembly
ASSEMBLY CODE
Call Stack
CALL STACK WINDOW
REGISTER CONTENT
MEMORY CONTENT
VARIABLES FROM CURRENT AND PREVIOUS STATEMENT
DESIGNATED WATCH VARIABLES
Ready

Figure 9: The Visual C++ debugger window.

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

AppWizard

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

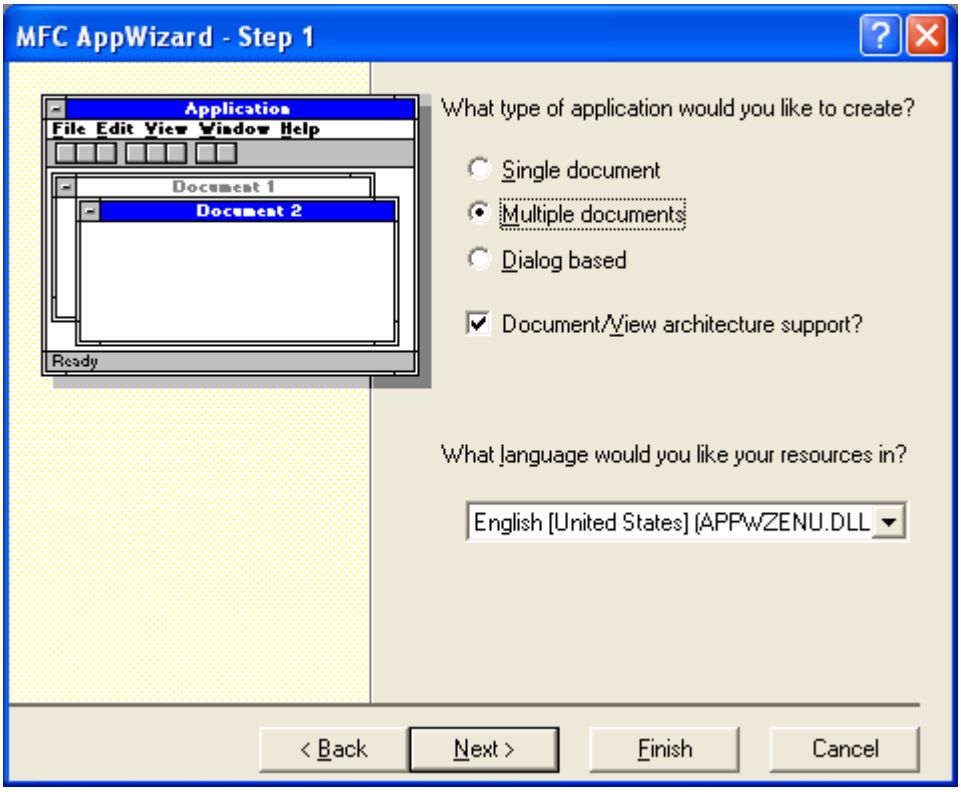


Figure 10: Visual C++ AppWizard step 1 of 6.

You'll use AppWizard extensively as you work through the examples in this Tutorial. AppWizard code is minimalist code; the functionality is inside the application framework base classes. AppWizard gets you started quickly with a new application. Advanced developers can build custom AppWizards. Microsoft Corporation has exposed its macro-based system for generating projects. If you discover that your team needs to develop multiple projects with a telecommunications interface, you can build a special wizard that automates the process.

ClassWizard

ClassWizard is a program (implemented as a DLL) that's accessible from Visual C++'s View menu. ClassWizard takes the drudgery out of maintaining Visual C++ class code. It can be access through the **View** menu and **ClassWizard…** sub menu.

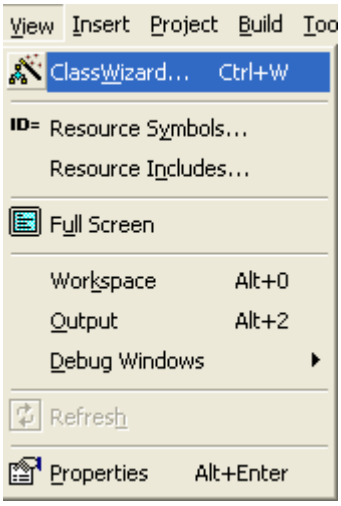


Figure 11: Invoking the ClassWizard through the **View** menu.

The following is the MFC ClassWizard dialog box.

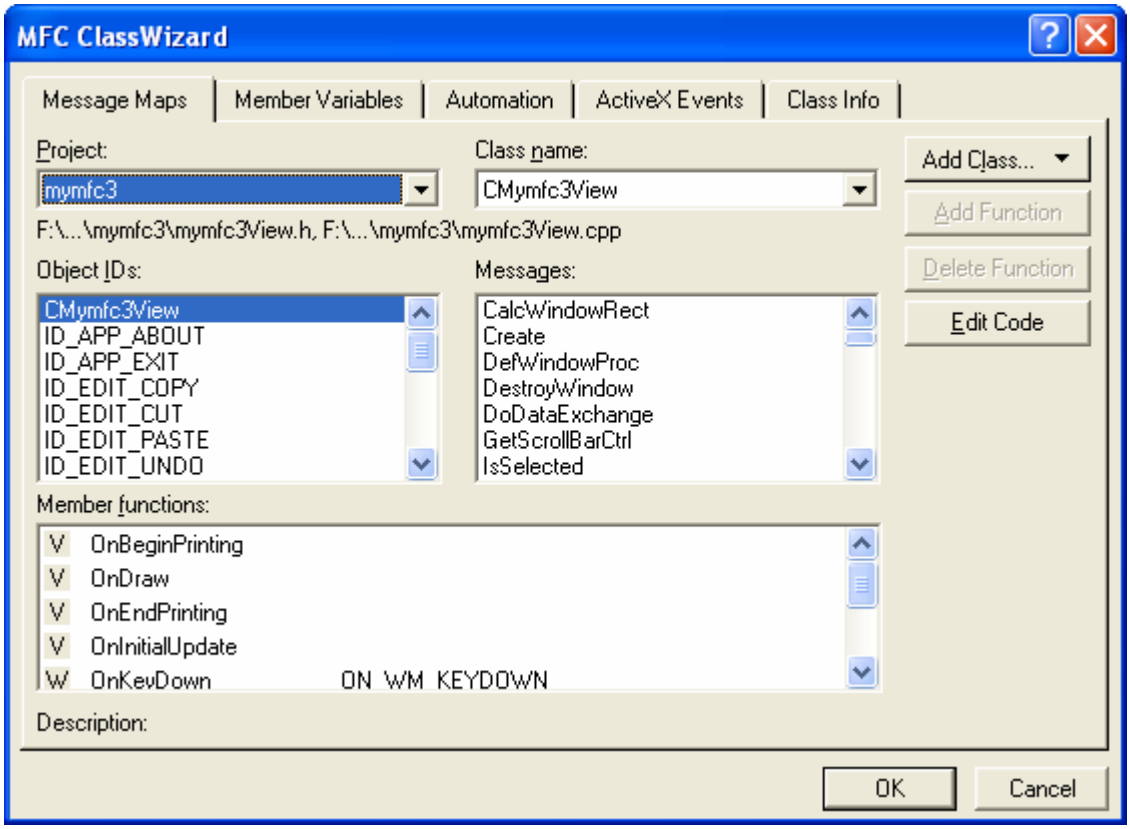


Figure 12: ClassWizard dialog.

When you need a new class, a new virtual function, or a new message-handler function (Message maps), ClassWizard writes the prototypes, the function bodies, and (if necessary) the code to link the Windows message to the function. ClassWizard can update class code that you write, so you avoid the maintenance problems common to ordinary code generators. Some ClassWizard features are available from Visual C++'s WizardBar toolbar, shown below.

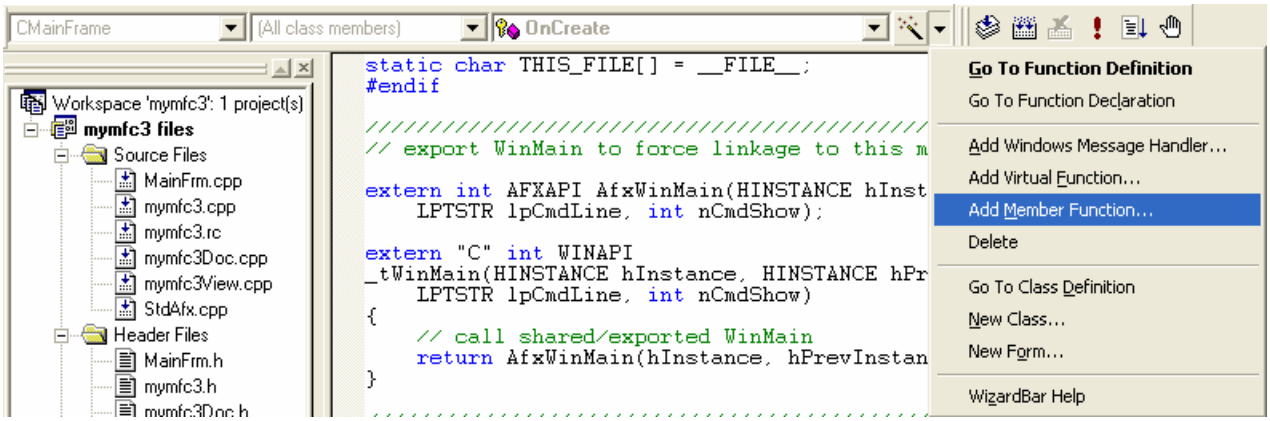


Figure 13: Visual C++'s WizardBar toolbar.

The Source Browser

If you write an application from scratch, you probably have a good mental picture of your source code files, classes, and member functions. If you take over someone else's application, you'll need some assistance. The Visual C++ Source Browser (the browser, for short) lets you examine (and edit) an application from the class or function viewpoint instead of from the file viewpoint.

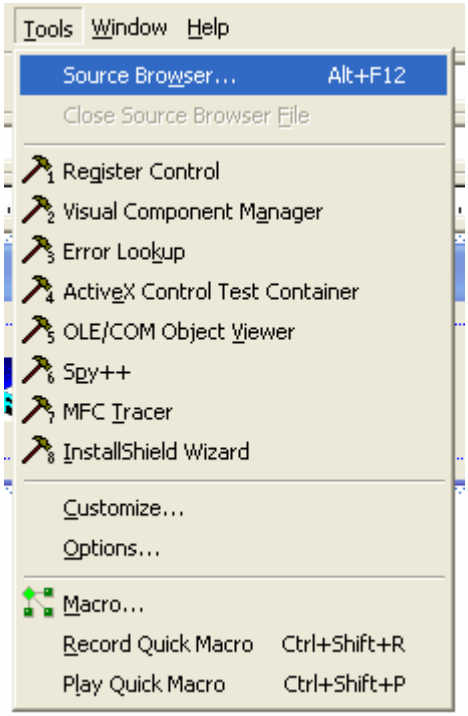


Figure 14: Visual C++ Source Browser.

It's a little like the "inspector" tools available with object-oriented libraries such as Smalltalk. The browser has the following viewing modes:

- **Definitions and References**: You select any function, variable, type, macro, or class and then see where it's defined and used in your project.
- **Call Graph/Callers Graph**: For a selected function, you'll see a graphical representation of the functions it calls or the functions that call it.

- **Derived Classes and Members/Base Classes and Members**: These are graphical class hierarchy diagrams. For a selected class, you see the derived classes or the base classes plus members. You can control the hierarchy expansion with the mouse.
- **File Outline**: For a selected file, the classes, functions, and data members appear together with the places in which they're defined and used in your project.

If you rearrange the lines in any source code file, Visual C++ regenerates the browser database when you rebuild the project. This increases the build time. In addition to the browser, Visual C++ has a **ClassView** option (shown below) that does not depend on the browser database. You get a tree view of all the classes in your project, showing member functions and data members. Double-click on an element, and you see the source code immediately. The ClassView does not show hierarchy information, whereas the browser does.

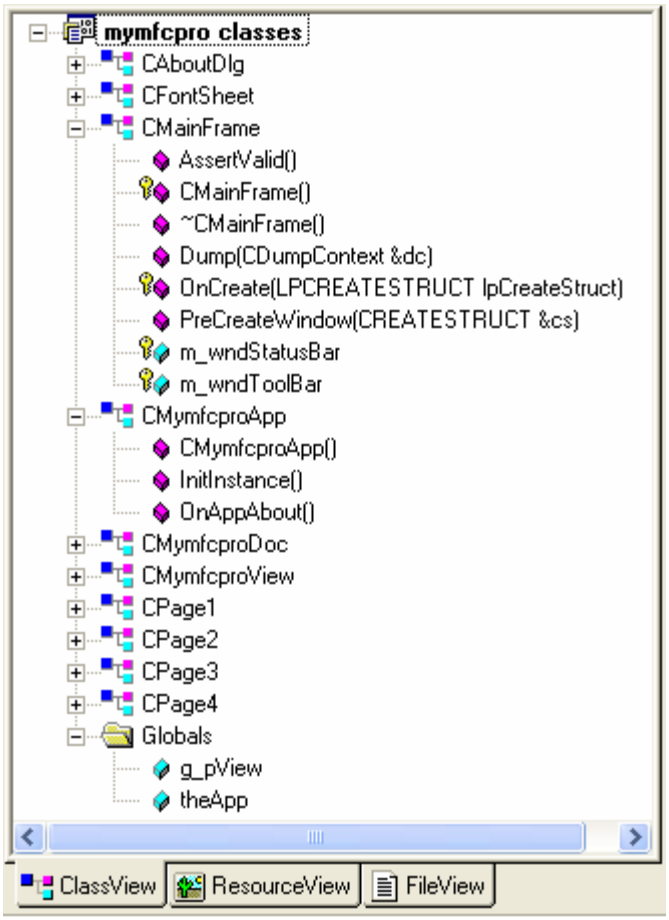


Figure 15: Visual C++ ClassView.

Windows Diagnostic Tools

Visual C++ 6.0 contains a number of useful diagnostic tools. For example, 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.

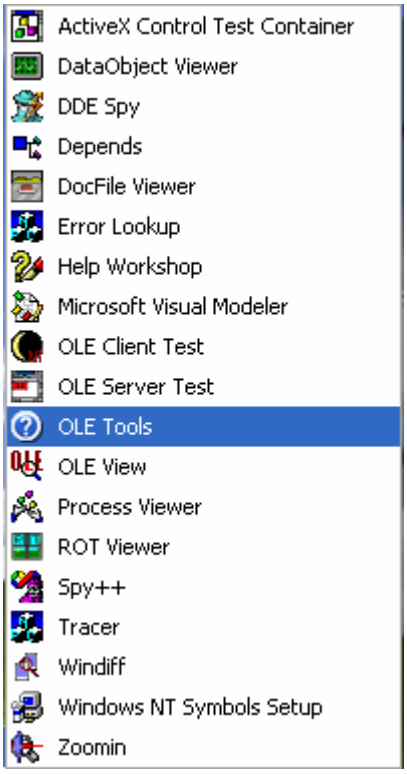


Figure 16: Visual C++ diagnostic tools.

You'll find **PVIEW** (PVIEW95 for Windows 95) useful for killing errant processes that aren't visible from the Windows 95 task list. The Windows NT Task Manager, which you can run by right-clicking the toolbar, is an alternative to PVIEW. Visual C++ also includes a whole suite of ActiveX utilities, an ActiveX control test program (now with full source code in Visual C++ 6.0), the help workshop (with compiler), a library manager, binary file viewers and editors, a source code profiler, and other utilities.

Source Code Control

During development of Visual C++ 5.0, Microsoft bought the rights to an established source code control product named **SourceSafe**.

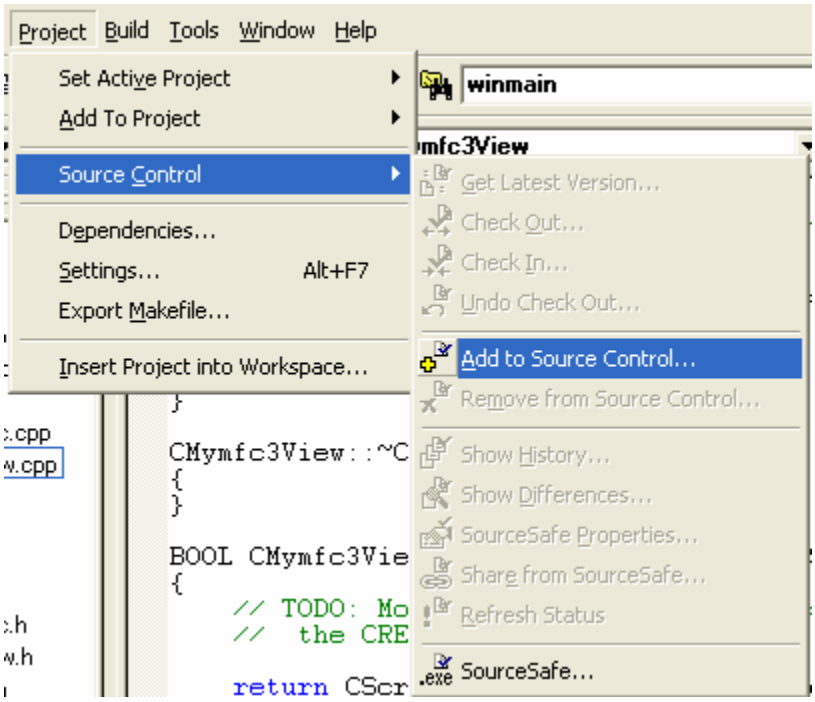


Figure 17: Visual C++ Source Code control.

This product has since been included in the Enterprise Edition of Visual C++ and Visual Studio Enterprise, and it is integrated into Visual C++ so that you can coordinate large software projects. The master copy of the project's source code is stored in a central place on the network, and programmers can check out modules for updates. These checked-out modules are usually stored on the programmer's local hard disk. After a programmer checks in modified files, other team members can synchronize their local hard disk copies to the master copy. Other source code control systems can also be integrated into Visual C++.

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 software firms.

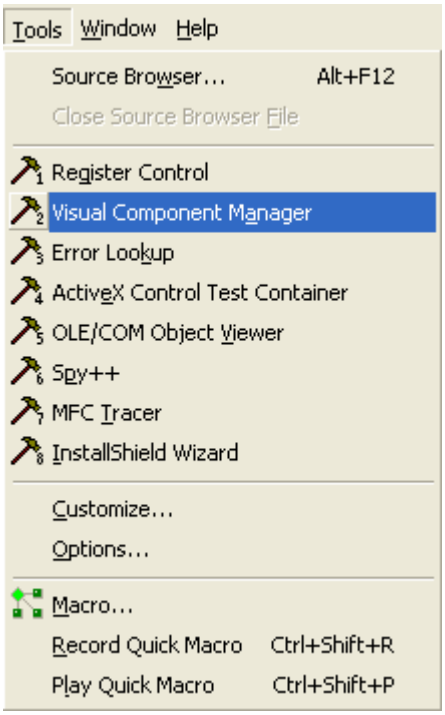


Figure 18: Visual C++ Components and Controls Gallery.

If you decide to use one of the prepackaged Visual C++ components, try it out first in a dummy project to see if it's what you really want. Otherwise, it might be difficult to remove the generated code from your regular project. All user-generated Gallery items can be imported from and exported to **OGX files**. These files are the distribution and sharing medium for Visual C++ components.

The Microsoft Active Template Library (ATL)

ATL is a tool, separate from MFC, for building ActiveX controls. You can build ActiveX controls with either MFC or ATL, but ATL controls are much smaller and quicker to load on the Internet.

The Microsoft Foundation Class Library

The Microsoft Foundation Class Library (the MFC library, for short) defines the **application framework** that you'll be learning in this Tutorial. MFC provides a variety of classes designed to serve a wide range of needs. You'll find a handy diagram of the MFC 7.0 class hierarchy here. The majority of MFC classes are derived, either directly or indirectly, from `CObject`. `CObject` provides other useful benefits to its derived classes as well. For example, it overloads the `new` and `delete` operators to provide protection against memory leaks. If you create an object from a `CObject`-derived class and fail to delete it before the application terminates, MFC will warn you by writing a message to the debug output window. The overarching importance of this most basic of MFC classes will become increasingly clear as you grow more familiar with MFC.

What's an Application Framework?

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

An Application Framework vs. a Class Library

One reason that C++ is a popular language is that it can be "extended" with class libraries. Some class libraries are delivered with C++ compilers, others are sold by third-party software firms, and still others are developed in-house. A

class library is a set of related C++ classes that can be used in an application. A mathematics class library, for example, might perform common mathematics operations, and a communications class library might support the transfer of data over a serial link. Sometimes you construct objects of the supplied classes; sometimes you derive your own classes, it all depends on the design of the particular class library.
An application framework is a superset of a class library. An ordinary library is an isolated set of classes designed to be incorporated into any program, but an application framework defines the structure of the program itself. Microsoft didn't invent the application framework concept. It appeared first in the academic world, and the first commercial version was MacApp for the Apple Macintosh. Since MFC 2.0 was introduced, other companies, including Borland, have released similar products.

An Application Framework Example

It's time to look at some code, not pseudocode but real code that actually compiles and runs with the MFC library. It's the good old "Hello, world!" application, with a few additions. It's about the minimum amount of code for a working MFC library application for Windows. You don't have to understand every line now. This is Single Document Interface (SDI) without **Document /View architecture support** application and the full steps to build this program are given in Example 1. By convention, MFC library class names begin with the letter `C`. Following is the source code for the header and implementation files for our MYAPP application (all the Visual C++ comments have been deleted). The classes `CMyApp` and `CMyFrame` are each derived from MFC library base classes. The first is the **MyApp.h** header file for the MYAPP application:

```
// MyApp.h
// 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"

CMyApp theApp; // 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. You'll learn the details of the MFC library's message mapping in Module 3. 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.

Convention

The notation used in MFC is mix of the Hungarian and CamelCase. If you already got familiar with the MFC/Windows programming, you will recognize the codes such as variables, classes, objects etc. based on the notations used. For example, MFC library class names begin with the letter C such as `CScrollView` and variables prefixed with `m_`.

Object ID Naming Conventions

There are several categories or types of IDs found in Windows applications. The MFC ID-naming convention defines different prefixes for different resource types. MFC uses the prefix "`IDR_`" to refer to a resource ID that applies to multiple resource types. For example, for a given frame window, the same "`IDR_`" value is used to refer to a menu, accelerator, string and icon resource all at once.

Object ID	Description
`IDR_`	Multiple resource types (Used for Menus, Accelerators primarily).
`IDD_`	For dialog template resources (for example, IDD_DIALOG1).
`IDC_`	For Cursor resources.
`IDI_`	For Icon resources.
`IDB_`	For Bitmap resources.
`IDS_`	For String resources.

Table 1.

Note that the `IDS_` value for a string resource is the ID passed to LoadString. The actual implementation of string table resources groups together 16 strings into one segment. For example, within a `DIALOG` resource, we follow the convention of:

Object ID	Description
`IDOK` , `IDCANCEL`	For standard push button IDs.
`IDC_`	For other dialog controls.

Table 2.

The "`IDC_`" prefix is also used for cursors. This naming conflict is not usually a problem since a typical application will have few cursors and a large number of dialog controls. Within a **Menu** resource, we follow the convention of:

Object ID	Description
`IDM_`	For menu items not using the MFC command architecture.
`ID_`	For menu item commands using the MFC command architecture.

Table 3.

Commands that follow the MFC command architecture must have an `ON_COMMAND` command handler and may have an `ON_UPDATE_COMMAND_UI` handler. If these command handlers follow the MFC command architecture, they will function correctly whether they are bound to a menu item, a toolbar button or a dialog bar button. The same `ID_` is also used for a menu prompt string displayed on the program's message bar. Most of the menu items in your application should follow the MFC command convention. All of the standard command IDs (for example, `ID_FILE_NEW`) follow this convention. MFC also uses "`IDP_`" as a specialized form of strings (that is, instead of "`IDS_`"). Strings with the "`IDP_`" prefix are "prompts," that is, strings used in message boxes. "`IDP_`" strings may contain "`%1`" and "`%2`" as place holders of strings determined by the program. "`IDP_`" strings usually have help topics, while "`IDS_`" strings do not. "`IDP_`" strings are always localized, while "`IDS_`" strings may or may not be localized. The MFC library also uses the "`IDW_`" prefix as a specialized form of control IDs (that is, instead of "`IDC_`"). These IDs are assigned to child windows such as views and splitters by the framework classes. MFC implementation IDs are prefixed with "`AFX_`".

Object ID-Numbering Convention

The following lists the valid ranges for the IDs of the specific types. Some of the limits are technical implementation limits while others are just conventions to prevent your IDs from colliding with Windows predefined IDs or MFC default implementations. We strongly recommend you do not defined IDs outside the recommended ranges. Even though the lower limit of many of these ranges is 1 (0 is not used), common convention starts practical use of IDs at 100 or 101.

Prefix	Resource type	Valid range
`IDR_`	multiple	1 → 0x6FFF
`IDD_`	dialog templates	1 → 0x6FFF
`IDC_`, `IDI_`, `IDB_`	cursors, icons, bitmaps	1 → 0x6FFF
`IDS_`, `IDP_`	general strings	1 → 0x7FFF
`ID_`	commands	0x8000 → 0xDFFF
`IDC_`	controls	8 → 0xDFFF

Table 4.

Reasons for these range limits:

- By convention, the ID value of 0 is not used.
- Windows implementation limitations restrict true resource IDs to be less than or equal to `0x7FFF`.
- MFC's internal framework implementations reserve several ranges: `0xE000` → `0xEFFF` and `0x7000` → `0x7FFF`.
- Several Windows system commands use the range of `0xF000` → `0xFFFF`.
- Control IDs of 1→7 are reserved by `IDOK`, `IDCANCEL`, and so on.
- The range of `0x8000`→`0xFFFF` for strings is reserved for menu prompts for commands.

Program Examples 1

Let go straight to the program example using MFC. This is a **Single Document Interface** (SDI) "Hello, world!" classic example. The application has only one window, an object of a class derived from `CFrameWnd`. All drawing occurs inside the frame window and all messages are handled there. This just a warm up session and we are using AppWizard that can automate most of our task in building Windows application. Follow the steps one-by-one.

Launch your Visual C++, click the **File** menu and click the **New…** sub menu. Click the **Projects** tab for the AppWizard as shown below. Select the **MFC AppWizard (exe)**, type your project name and set the project location as needed. Leave other setting as default and click the **OK** button.

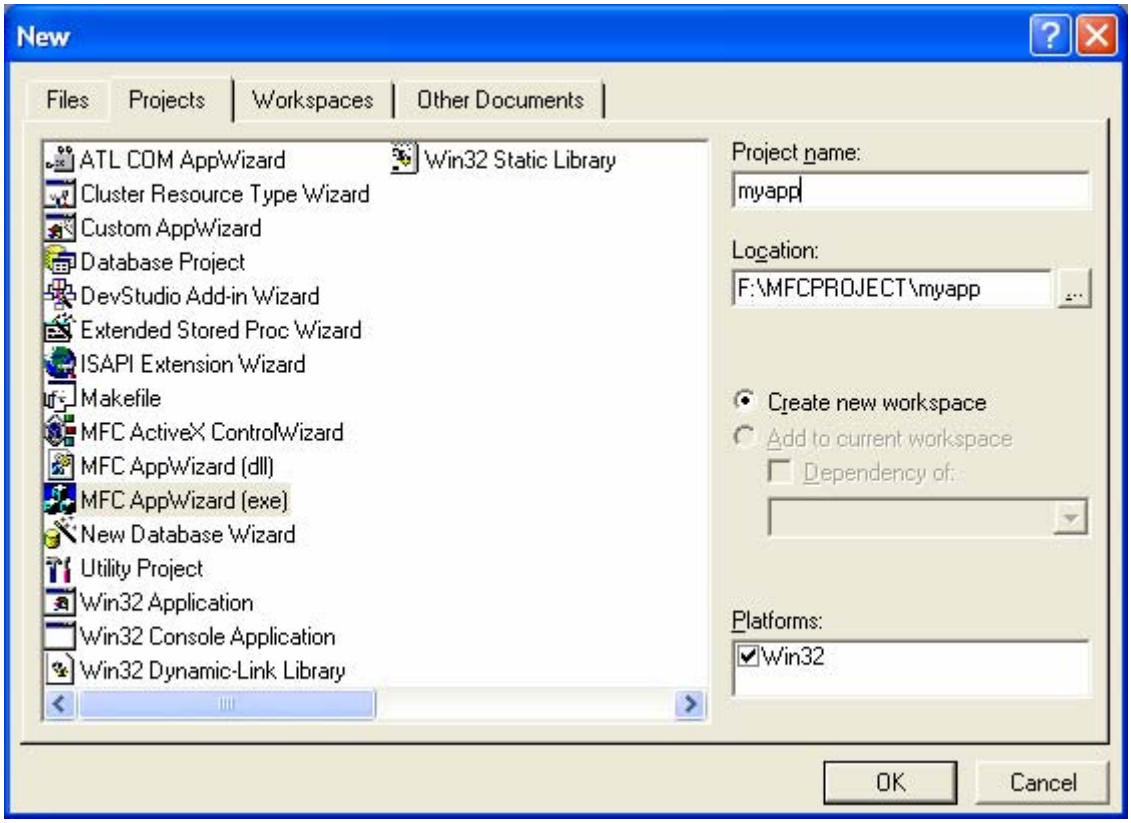


Figure 19: AppWizard dialog.

The wizard has 6 steps and you can exit at any step. Select the **Single document** radio button and uncheck the **Document/View architecture support** then click **Next** button.

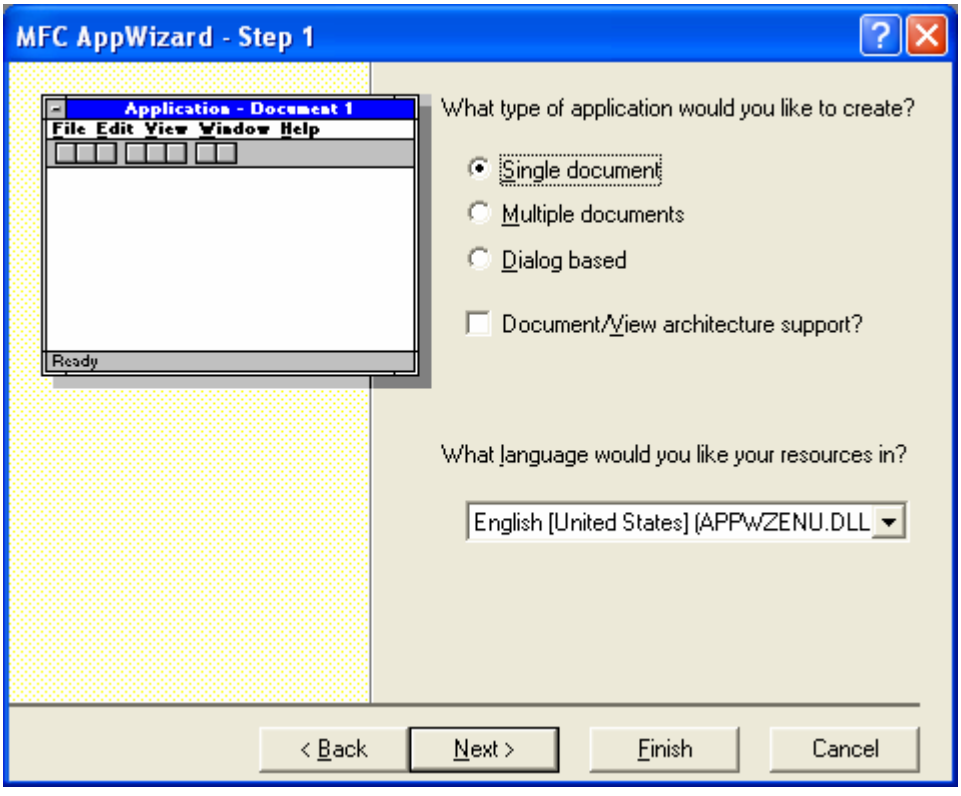


Figure 20: Visual C++ AppWizard step 1 of 6.

Select the **None** radio button for database support. Click the **Next** button. We are going to create the simplest project.

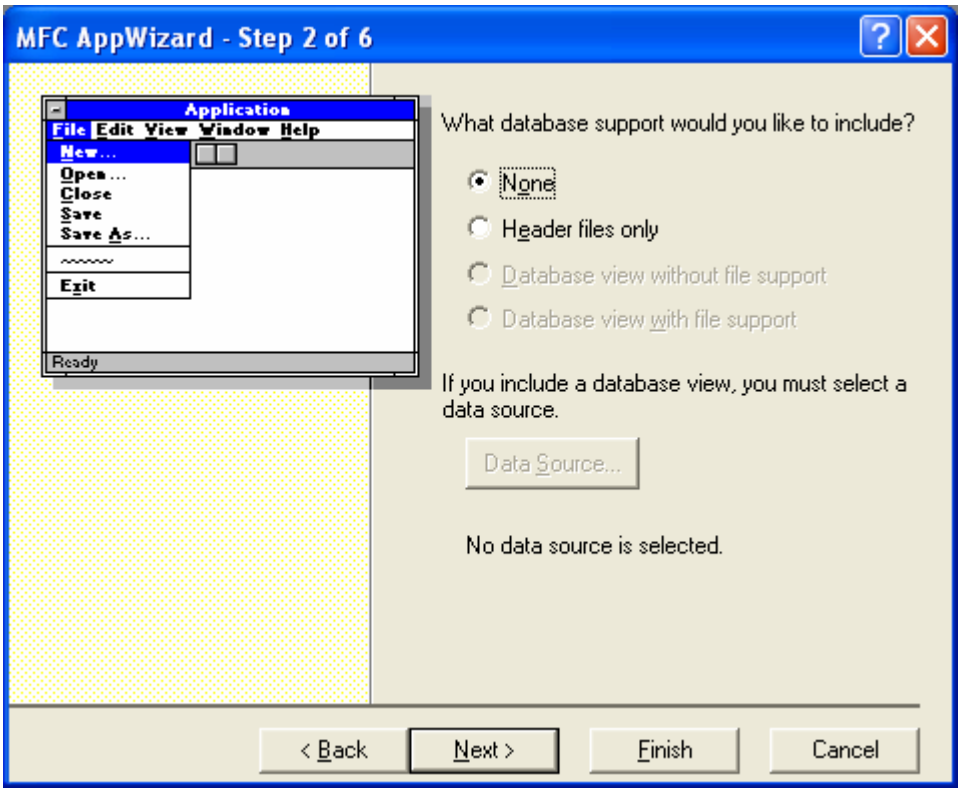


Figure 21: Visual C++ AppWizard step 2 of 6.

Uncheck the **ActiveX Controls**. Just to make our code simpler.

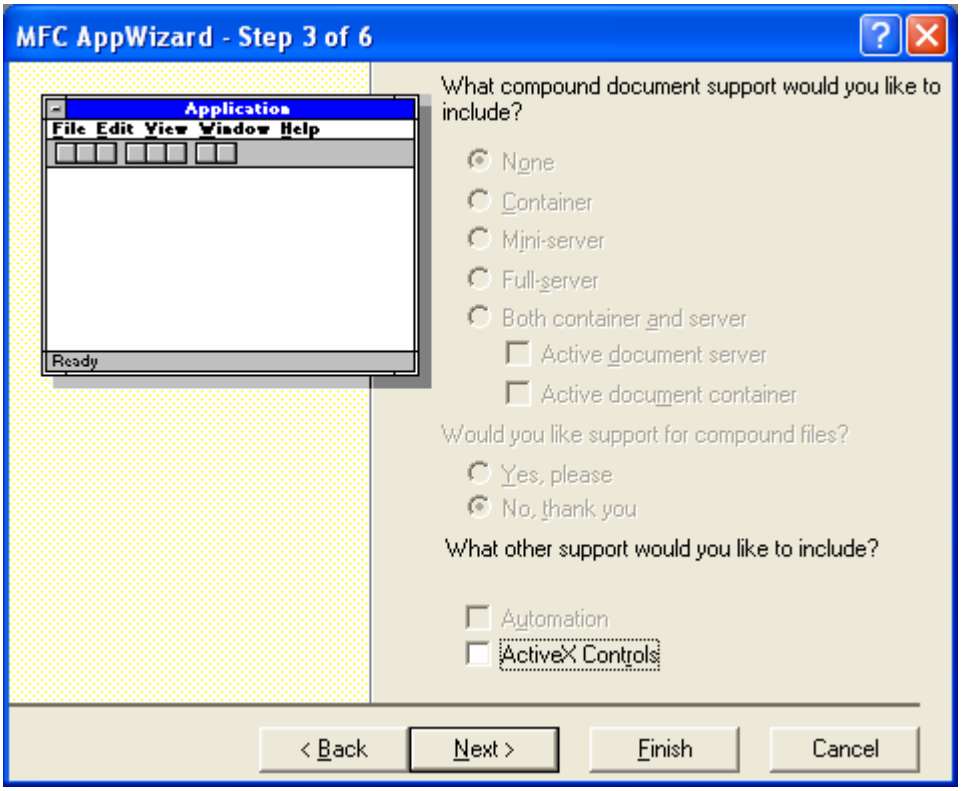


Figure 22: Visual C++ AppWizard step 3 of 6.

Accept the defaults and click the **Next** button.

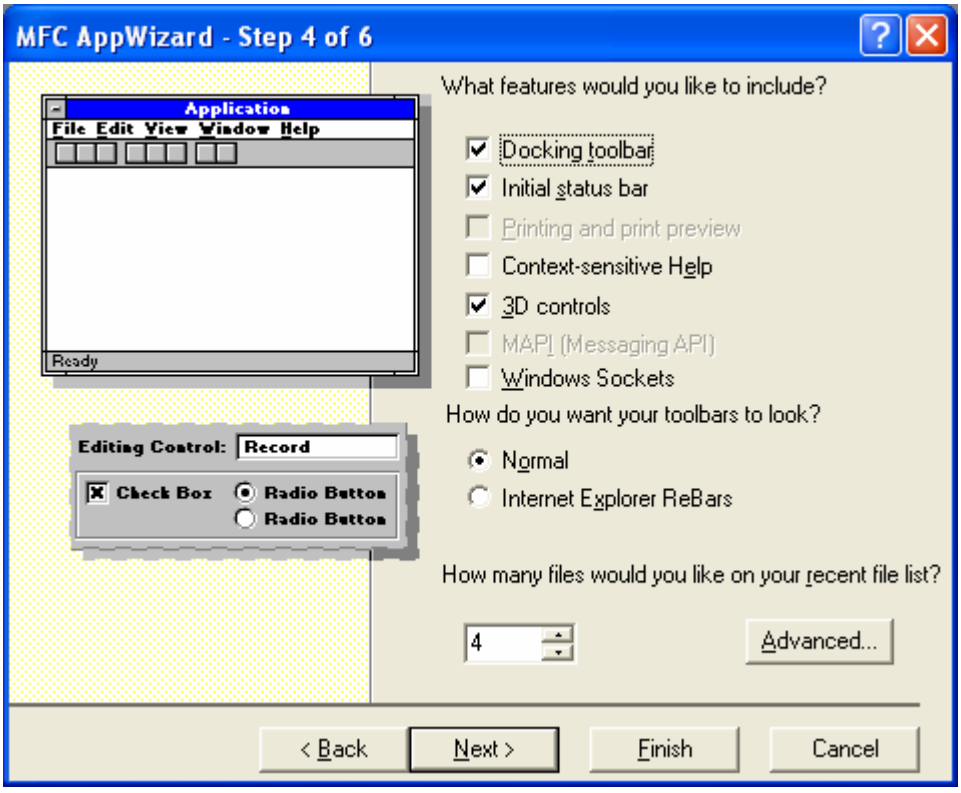


Figure 23: Visual C++ AppWizard step 4 of 6.

Accept the defaults and click **Next** button.

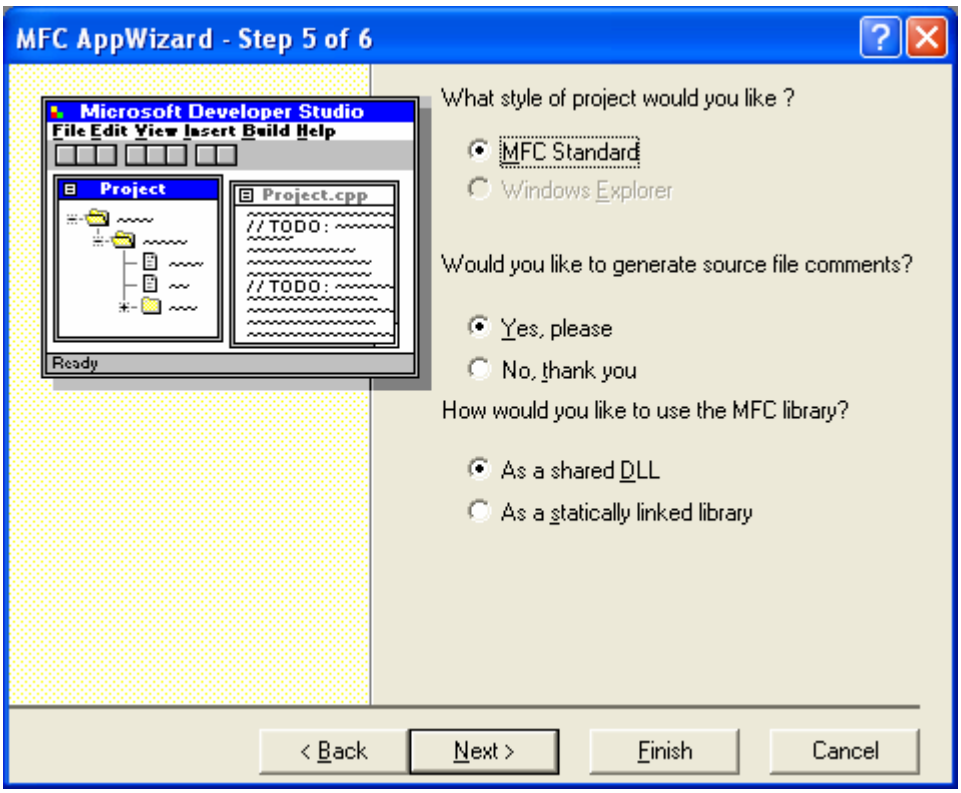


Figure 24: Visual C++ AppWizard step 5 of 6.

The following are the classes that will be generated (also the related files created) for our project. Click the **Next** button.

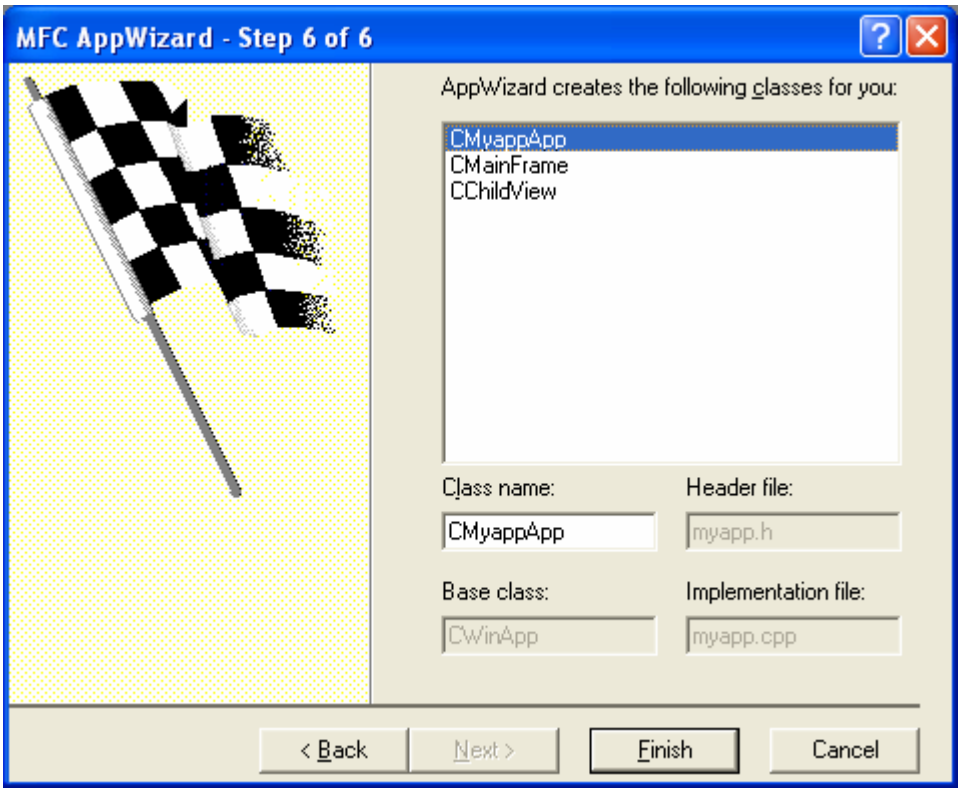


Figure 25: Visual C++ AppWizard step 6 of 6.

Finally the summary of the project settings. Click the **OK** button.

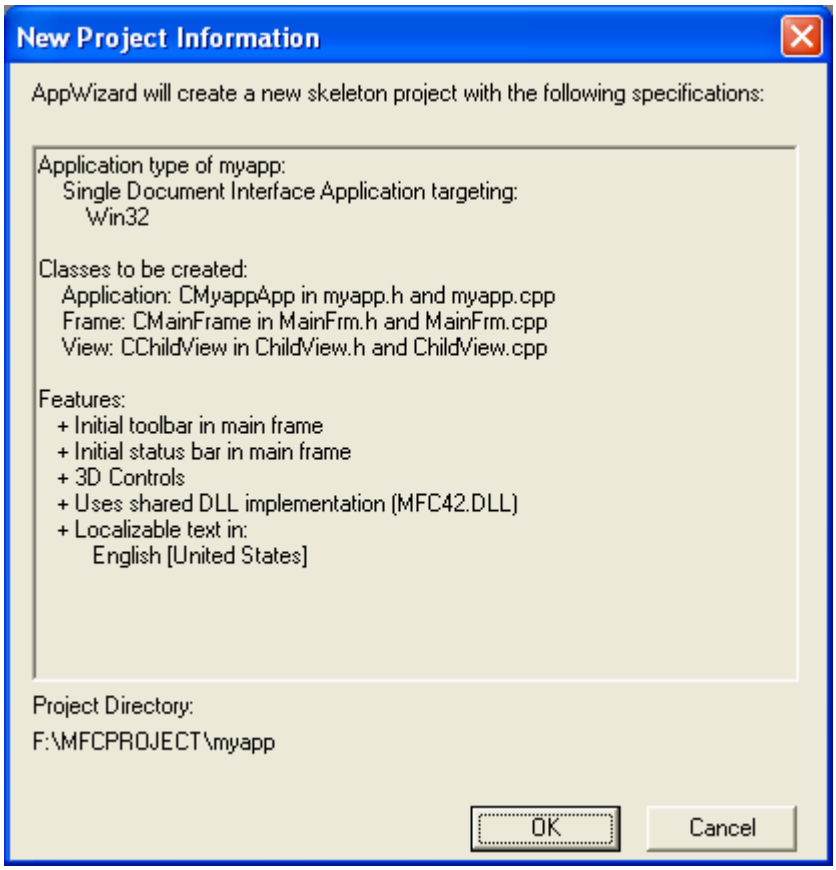


Figure 26: Visual C++ AppWizard project summary.

Expand all folders in the **FileView** and double click the **ChildView.cpp**. FileView displays all the project files that can be edited, logically. Physically, there are more projects' files.

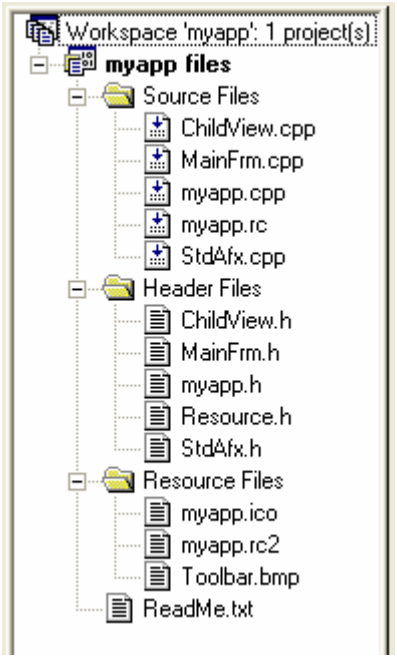


Figure 27: Visual C++ FileView.

Add code to paint in the dialog. Add the following code to the `CChildView::OnPaint` function in the **ChildView.cpp** source code file:

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

    dc.TextOut(0, 0, "Hello, world!");

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

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

    // TODO: Add your message handler code here
    dc.TextOut(0, 0, "Hello my MFC world!");

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

Listing 1.

Compile and run. You now have a complete SDI application that has no dependencies on the document-view architecture.

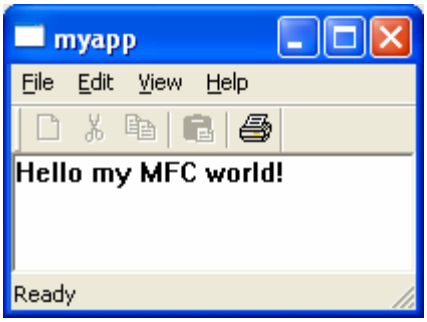


Figure 28: Visual C++ "Hello world" output.

Program Examples 2

The following is another classic example using MFC AppWizard, and Visual C++ to create, build, and run a simple MFC-based single document interface (SDI) **Scribbl**e application. We just dealing with the View part of the Document/View architecture. We will learn more detail about the Document/View architecture later.

Use MFC AppWizard to create a new project

1. In Visual C++/Visual Studio, on the **File** menu, click **New**.
2. In the **New** dialog box, click the **Projects** tab, and then do the following:
    - For the project type, select MFC AppWizard (exe).
    - For the project name, type **myscribble**.
    - Set the location for your project as needed.
    - Accept the default platform **Win32**.
    - Click **OK** to create the new project workspace. MFC AppWizard starts.
3. In MFC AppWizard Step 1, click **Single document** and **English**, leave the **Document/View architecture support** checkbox selected, and then click **Next** to go to Step 2.
4. Because this project does not need database support, accept the default **None** for database support and click **Next** to go to Step 3.
5. In Step 3, accept **None** for the **document support**, because we are not going to create a compound document that using Object Linking and Embedding (OLE), clear the **ActiveX Controls** check box, and then click **Next** to go to Step 4.
6. In Step 4, clear the **Printing and print preview** check box, accept the defaults (Docking toolbar, Initial status bar, 3D controls, and 4 files for the recent file list) and then click Next to go to Step 5.
7. In Step 5, leave the project style as **MFC Standard**, click **Yes**, **please for generating source file comments** and select **As a shared DLL** for MFC support. Click **Next** to go to Step 6.
8. In Step 6, click **Finish** to display the **New Project Information** dialog box summarizing your choices.
9. To create the application files with MFC AppWizard, click **OK**.

When MFC AppWizard is finished, you will be returned to Visual C++/Visual Studio. To see the classes that MFC AppWizard created, click the **ClassView** tab in the **Project Workspace** window.

Add message handlers that will handle several mouse clicking and dragging events. This just the skeleton, we will add the real codes later – myscribbleView.cpp file.

1. On the **View** menu, click **ClassWizard**. The MFC ClassWizard property page appears.
2. In the **Message Maps** tab, in the combo boxes provided on the property page, select the **myscribble** project, the **CMyscribbleView** class, and the **CMyscribbleView** Object ID.
3. We are going to add member functions in the **CMyscribbleView** class. In the **Messages** list box, select the `WM_LBUTTONDOWN` message and click **Add Function**.
4. Repeat Step 3 for the `WM_MOUSEMOVE` and `WM_LBUTTONUP` messages.
5. Click **OK** to close ClassWizard.

Implement `OnLButtonDown`, `OnMouseMove`, and `OnLButtonUp` handlers. This is the real codes that will implement the mouse clicking and dragging events.

1. Declaring variables in the declaration part. Open **MyscribbleView.h** (or use the ClassView).
2. In the public attributes section of the `CMyscribbleView` class declaration, define `startpt` and `endpt` variables as follows:

```
CPoint startpt, endpt;
```

Or by using the ClassView, select the `CMyscribbleView` and right click, select the **Add Member Variable**:

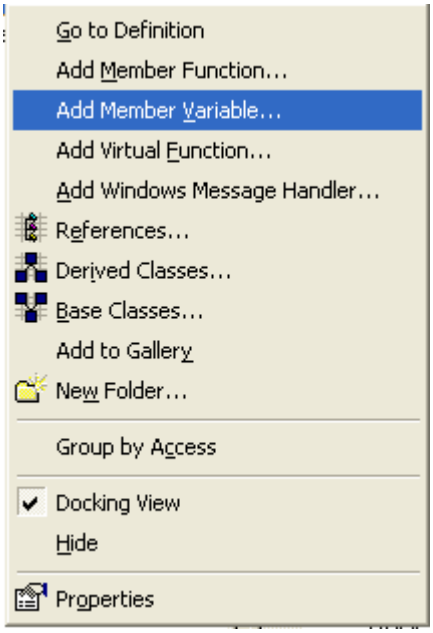


Figure 29: Adding a member variable through the ClassView.

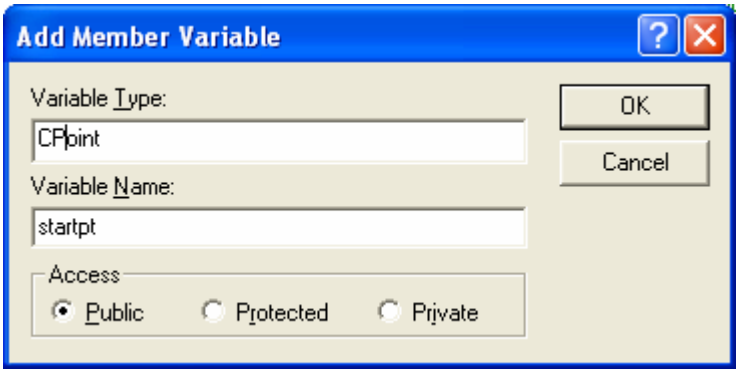


Figure 30: Add member variable dialog.

3. Save **MyscribbleView.h**.
4. Then, the implementation codes in the implementation part. Open **MyscribbleView.cpp**.
5. In the constructor, initialize `startpt` and `endpt` coordinates to `-1` as follows:

```
startpt = -1;
endpt = -1;
```

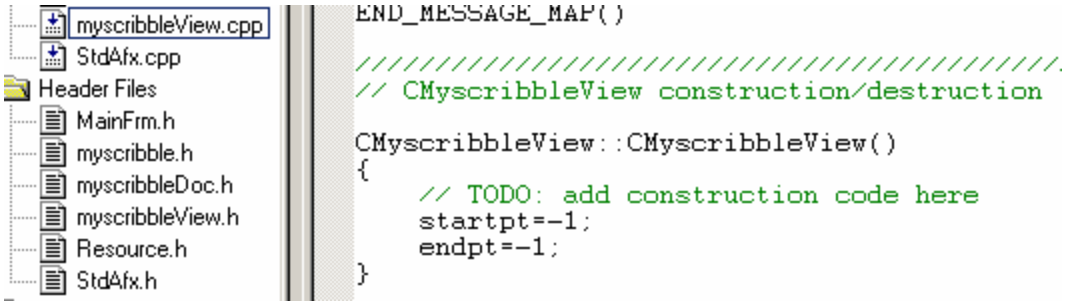


Figure 31: Scribble code segment.

6. Scroll to the `OnLButtonDown` handler.
7. In the `OnLButtonDown` handler, save the point where the mouse button is pressed as the start point:

```
startpt.x = point.x;
startpt.y = point.y;
```

8. Scroll to the `OnMouseMove` handler.
9. In the `OnMouseMove` handler, add the following code to draw a line from the previous detected point in the mouse drag to the current point:

```
CClientDC dc(this);
endpt.x = point.x;
endpt.y = point.y;
if (startpt.x != -1)
{
   dc.MoveTo(startpt.x, startpt.y);
   dc.LineTo(endpt.x, endpt.y);
   startpt.x = endpt.x;
   startpt.y = endpt.y;
}
```

10. In the `OnLButtonUp` handler, add code to reinitialize the variable `startpt` as follows:

```
   startpt = -1;
```

```
void CMyscribbleView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    startpt.x=point.x;
    startpt.y=point.y;

    CView::OnLButtonDown(nFlags, point);
}

void CMyscribbleView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC dc(this);
    endpt.x = point.x;
    endpt.y = point.y;
    if (startpt.x != -1 )
    {
        dc.MoveTo(startpt.x, startpt.y);
        dc.LineTo(endpt.x, endpt.y);
        startpt.x = endpt.x;
        startpt.y = endpt.y;
    }

    CView::OnMouseMove(nFlags, point);
}

void CMyscribbleView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    startpt = -1;

    CView::OnLButtonUp(nFlags, point);
}
```

Listing 3: Scribbles' C++ code segment.

Build and run the project

1. On the **Build** menu, click **Build Myscribble.exe**. Visual Studio displays the status of the build process as it builds your project.
2. After the build is complete, on the **Build** menu, click **Execute Myscribble.exe**. The Scribble application starts. Try writing something using your mouse.

Figure 32: Visual C++'s Scribble program output.

.Net Framework

Since its introduction in 2002, people's attention has focused on the many new features that have formed part of Microsoft .NET, such as the major changes to Microsoft Visual Basic (Visual Basic .Net), the introduction of C#, the new ASP.NET and ADO.NET models, and the increased use of XML. So, from application framework we are introduced the next level, .**Net framework** (currently version 2.0). C++ developers need not feel left out, however, because a lot of the new features in Microsoft Visual C++ .NET make C++ a first-class member of the .NET family of programming languages. This new functionality is called the **Managed Extensions for C++**, and as well as providing C++ programmers with access to all the functionality in the .NET class libraries, it also lets you interoperate with existing C++ code, COM objects, and the Win32 API.

Managed vs. Unmanaged Code

Code and data that live in the .NET world are called **managed** because locations and lifetimes are managed by the common language runtime (CLR). Code and data that exist outside of .NET are called **unmanaged**, because there is no central mechanism for managing their lifetimes.

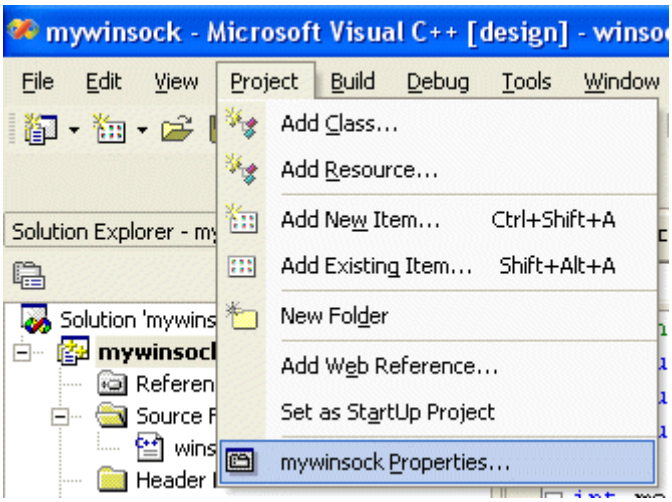


Figure 33: Enabling/disabling the **/clr** for managed/unmanaged code in Visual C++ .Net through the project properties.

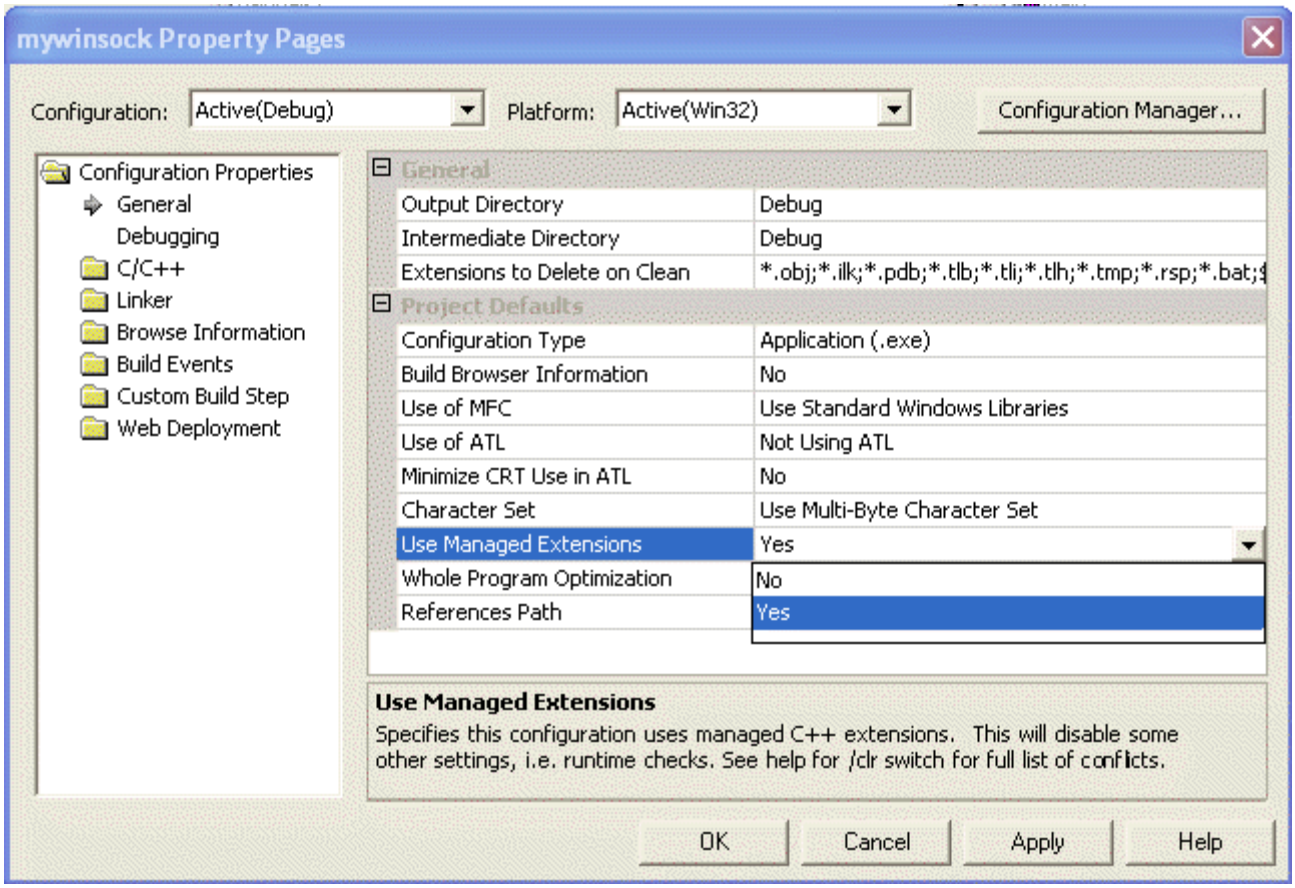


Figure 34: Enabling/disabling the **/clr** in Visual C++ .Net.

Sometimes you have to mix the two, calling existing unmanaged code from within .NET. The .NET Framework is the new library of classes that you use to build Windows applications. It is large, quite complex, and far-reaching in its scope. For MFC programming we do not use the **/clr** (can be enabled in the Visual C++ .Net: **Project** menu → ***your_project_name*** **Properties → General → Use Managed Extensions → Yes/No**, as shown in the above Figures), that is unmanaged. The managed code will be used in the .Net programming and don't confused that .Net framework is not compiler, it is framework. All program examples used in this Tutorial series are unmanaged and we are using Visual C++ 6.0. The managed code used in Visual C++ .Net provided that the **/clr** is used.

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.