Beginning MFC Programming

Prerequisites

Hi there. Welcome to the wonderful world of Windows programming with MFC. Before beginning this tutorial, I would like to assume a few things, such as the reader is comfortable with –

1. C++ - the MFC is based on C++, and so it would be natural to expect that the reader knows a little bit of C++. Although MFC doesn't use the newer features of C++ like templates and others, it is still overwhelming for the first-time user.
2. Windows programming – the MFC is a framework over the Windows API and it would again be natural to expect that the reader knows about Windows programming, at least concepts about Windows, Window classes, Windows messages, Window procedure, etc. if not the GDI and other advanced API functions. If all these sound Latin and Greek to you then please refer to my earlier tutorial on 'Beginning Windows Programming'.

Introduction

The Microsoft Foundation Class library is an application framework specifically tailored for creating applications for the Windows operating system. An application framework is "an integrated collection of object-oriented software components that offers all that's needed for a generic application". So, the framework not only provides many classes for making an application, but also defines the structure of the program itself.

MFC is arguably one of the most distinguishing features of Visual C++. The vast collection of C++ classes encapsulates most of the Win32 API and provides a powerful framework for typical (and not so typical) applications. Although that is not the only reason why one would use MFC for delivering real-world applications, it is certainly a good reason to use it. The other advantages of using MFC would be –

1. It simplifies the Windows API by grouping the API functions into logical units. In doing this, they it takes advantage of C++'s classes and inheritance to make development conceptually easier. It completely hides away the messy details of Windows programming such as registering window classes, window and dialog procedures, and message routing.
2. It also makes the transition from standard API function calls to the use of class libraries as simple as possible by designing classes that requires minimal relearning of function names for seasoned Windows programmers. It also allows mixing of traditional function calls with the use of the class library.
3. One of the design considerations of the MFC was to have the speed and efficiency of C/SDK applications. By inventing mechanisms to avoid the overhead of virtual functions for handling Windows messages, applications developed using MFC still manage to be fast and small in size.

4. Extensibility is one major factor for projects using any class library and MFC is no exception. The Windows API still continues to collect new material. When OLE, ODBC and 32-bit API was added, MFC still provided a firm foundation with it's extensible design. Each new API introduces a new learning curve but MFC eases the burden by encapsulating the details of the API in a set of classes.
5. If the above advantages have still not infused confidence in using MFC, then this surely will – Visual C++ IDE integration. The Visual C++ tools like the resource editor, the AppWizard and the ClassWizard reduce the drudgery and time of writing code for your applications. So, instead of churning out lots of mundane, but necessary code as in Windows SDK programming, the tools do the job for you!

The above statements about MFC must have surely given you the idea that MFC is big. It has to be – it encapsulates the Windows API and provides a robust application framework. I have always found that a high-level overview helps tremendously in trying to understand a complex topic. So, before getting bogged down by details of beginning to use the class library, let us first explore the class library.

"The Big Picture"

The classes in the MFC can be broadly classified into the following categories –

1. General Classes – provide things like string-handling class and collection classes.
2. Windows API Classes – provides a wrapper over the Windows API.
3. Application framework classes – which handle large pieces of the whole application such as message-pumping logic, printing as well as MFC's document/view architecture.
4. High-level abstractions – for several operating system extensions, including OLE, MAPI, and WinSock.

The following diagram gives an overview and layout of the above mentioned categories.

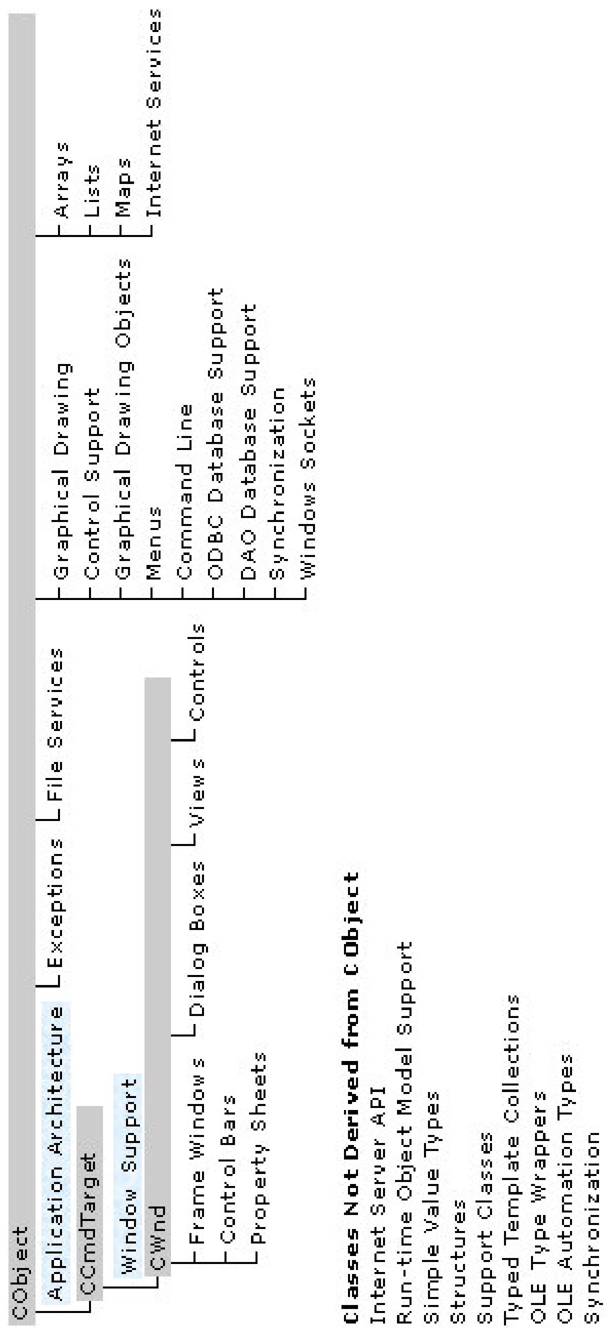
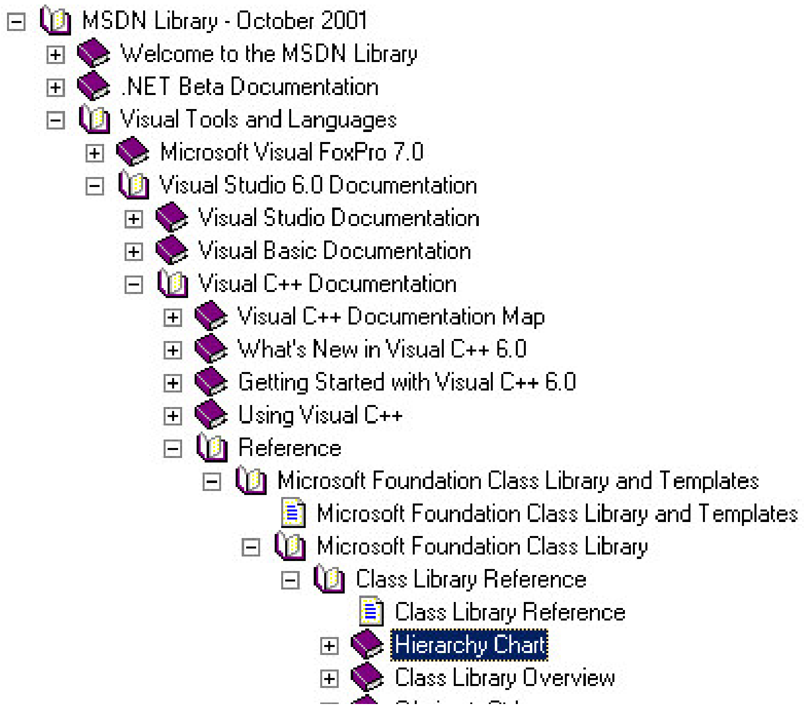

CObject
Application Architecture
CCmdTarget
Window Support
CWnd
Frame Windows
Control Bars
Property Sheets
Dialog Boxes
Views
Controls
Exceptions
File Services
Graphical Drawing
Control Support
Graphical Drawing Objects
Menus
Command Line
ODBC Database Support
DAO Database Support
Synchronization
Windows Sockets
Arrays
Lists
Maps
Internet Services
Classes Not Derived from CObject
Internet Server API
Run-time Object Model Support
Simple Value Types
Structures
Support Classes
Typed Template Collections
OLE Type Wrappers
OLE Automation Types
Synchronization

Figure 1

For a more complete diagram you can refer the Hierarchy chart shown in the MSDN at


MSDN Library - October 2001
Welcome to the MSDN Library
.NET Beta Documentation
Visual Tools and Languages
Microsoft Visual FoxPro 7.0
Visual Studio 6.0 Documentation
Visual Studio Documentation
Visual Basic Documentation
Visual C++ Documentation
Visual C++ Documentation Map
What's New in Visual C++ 6.0
Getting Started with Visual C++ 6.0
Using Visual C++
Reference
Microsoft Foundation Class Library and Templates
Microsoft Foundation Class Library and Templates
Microsoft Foundation Class Library
Class Library Reference
Class Library Reference
Hierarchy Chart
Class Library Overview

As seen from Figure 1, most classes in MFC are derived from a common root – the CObject class – and so it deserves some special mention.

CObject – the Mother of (Almost) all Classes

MFC's root class is called CObject. CObject defines and implements functionality that most MFC classes need in order to work with other parts of the framework. So, as you write MFC classes on your own or with the Class Wizard, you should usually make sure that you have CObject somewhere in your class hierarchy. When you derive your class from CObject, your class automatically gains the ability to add four key MFC features –

- **Run-time class information**
  CObject's run-time class information (RTCI) feature lets the developer determine information about an object such as class name and parent at run-time. MFC maintains this information with the help of the CRuntime class. Applications rarely use the CRuntime class directly, but rely on macros such as DECLARE_DYNAMIC (to be embedded in the definition of the class) and IMPLEMENT_DYNAMIC (to be added to the implementation file). These macros add the runtime information to the class and enable the use of the IsKindOf member function.

- **Dynamic creation**
  To add dynamic creation support to your CObject derivative, you must use the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros instead of

the DECARE_DYNAMIC and IMPLEMENT_DYNAMIC macros. Once added, you can create objects based on their CRuntimeClass information through the use of the CreateObject member function.

- **Persistence (serialization)**
  Persistence (also referred to as serialization in MFC) is the ability to store object and restore their states some time later. Using persistence, it is very easy to implement file reading and writing without having to worry about the format of the file you are writing to. To support serialization in your CObject-derived classes, you must use the DECLARE_SERIAL and IMPLEMENT_SERIAL macros instead of the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros. You will also have to override the Serialize member function of CObject, which makes use of the CArchive class, to read/write your member data.

- **Run-time object diagnostics**
  All CObject-derived MFC classes have a Dump member function that you can call to print out a C++'s object state at run time. The Dump routine makes use of the CDumpContext helper class to output the debugging information. CObject also has an AssertValid member function, in which the object check's its member's validity at run-time. Any other object can call AssertValid to verify that the object is in a safe state. Both of these functions are normally overridden in the Debug builds to provide advanced debugging facilities.

"Hello World"

So, with that small introduction to the MFC library, let's dive straight in to start coding using this massive library. The rest of the tutorial will just tackle MFC's basic Windows application support.

So, let's consider a do-nothing kind of application with MFC.

```
#include <afxwin.h>

CWinApp theApp;
```

Explanation of the code:

All MFC applications include the afxwin.h header file which not only includes the windows.h header file, but also other MFC header files required for MFC applications. The next statement is just instantiating an object of the CWinApp class.

Your first reaction to the above code might be – "Huh??" You might ask if this is a valid program at all. Well, if you have asked this question, then you are on the right track. As all Windows programmers know, to be able to run under Windows, a program must supply a WinMain function as an entry point. Actually WinMain is there – it's part of the application framework; you link it in when you link the framework code. You can find MFC's WinMain in APPMODUL.CPP (yes, MFC comes with complete source code, so that you can explore if you want to). MFC's WinMain function does all the things that a good WinMain should do, including initializing the application and starting up the message loop (this is actually done by MFC's own function called AfxWinMain in WINMAIN.CPP). So, you might ask if MFC's WinMain is doing all the things, why do I need to create an instance of CWinApp. The answer to that is that WinMain takes the help of an instantiated object of CWinApp (also, called the application object) to do all the above-mentioned chores of a normal WinMain. In C++, global objects are constructed before the main (or WinMain) function is executed. So, the CWinApp constructor (in APPCORE.CPP) is called before WinMain, where it initializes its member variables. CWinApp has member functions like InitApplication, InitInstance, Run and ExitInstance, which help in running the application. The complete flow is as shown in Figure 3. We will be exploring these functions more as we move ahead.

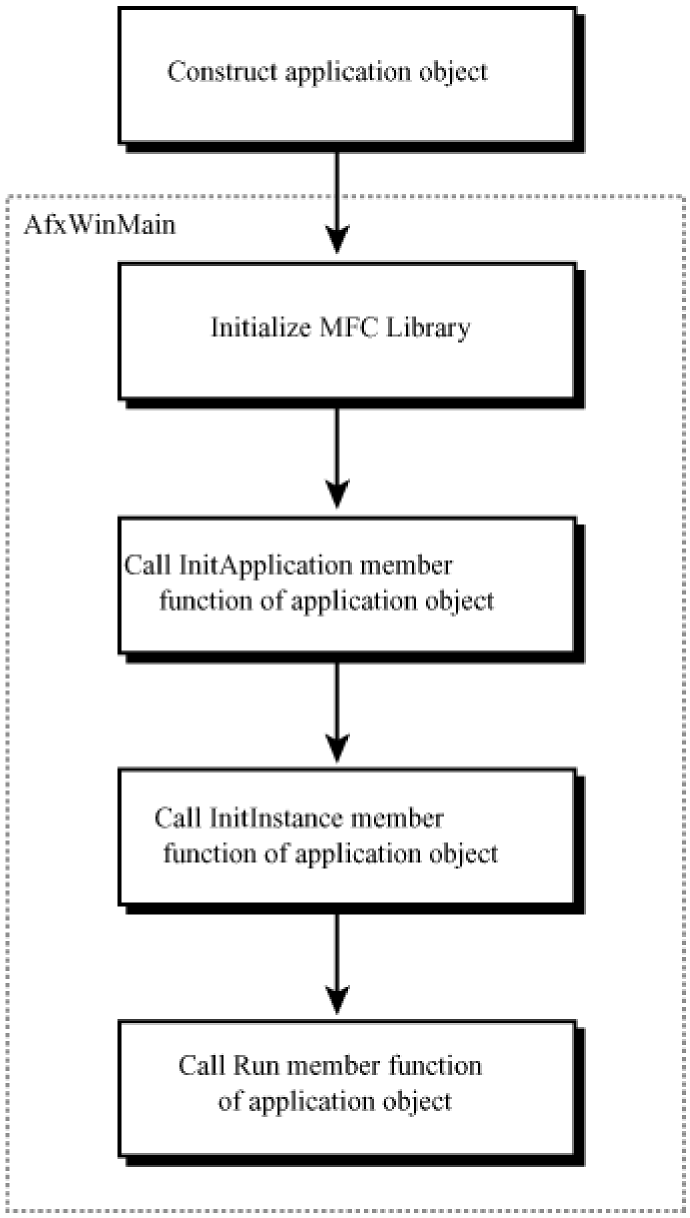
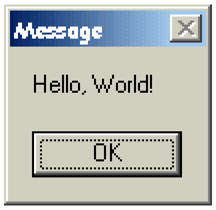

Construct application object
AfxWinMain
Initialize MFC Library
Call InitApplication member function of application object
Call InitInstance member function of application object
Call Run member function of application object

Figure 3

The above program did nothing (of course, how would you expect a 2-line program to do anything!). It was almost like an empty main function. So, let's look at another program that does something. The output would be something like this (this dialog would be familiar to readers of the earlier tutorial on Windows Programming).



Hello, World!

```
#include <afxwin.h>

class CMyApp : public CWinApp
{
public:
      BOOL InitInstance()
      {
            MessageBox(NULL, "Hello, World!", "Message", MB_OK);
            return TRUE;
      }
};

CMyApp theApp;
```

Explanation of the code:

Here, we define a class inheriting from CWinApp and override the InitInstance method defined in CWinApp. The InitInstance function is used for instance-specific initializations.

As shown in Figure 3, the first thing that AfxWinMain does is that it initializes the MFC library through AfxWinInit (APPINIT.CPP). Then, it calls two functions – InitApplication and InitInstance – of the application object in that order.

The InitApplication's job is to perform any initializations that pertain to the entire application. The default implementation in CWinApp does some specific initializations for the Document/View architecture classes. Note that in 32-bit Windows, this function is kind of obsolete with Windows making no distinction between the first and the subsequent instances of the application. So, all initializations should now take place in InitInstance.

CWinApp::InitInstance serves the purpose of performing instance-specific initializations of the program. CWinApp's default implementation does nothing. However, InitInstance is virtual, so that it can be safely overridden. So, it is normally implemented to do tasks such as setting all documents for an application and showing the main window. Because, the default version of InitInstance doesn't do anything, the first program did not do anything. It's up to you to make sure that a window appears on the screen. And that is precisely what we are doing in the second program.

The last thing that WinMain does before leaving is call the CWinApp-derived object's Run function. Run starts the ball rolling with the message loop. We'll see more about this in the coming examples.

Windows is all about "Popping windows on the screen"

Even though the second program did what we wanted, i.e., print the "Hello, World!" message on the screen, we could not define the characteristics of the window that was popped in front of us. That is, we could not have done anything interesting with the message-box window or the text shown inside it. So, we need a way to display our own windows. Consider the following program -

```
#include <afxwin.h>

class CMyApp : public CWinApp
{
public:
        BOOL InitInstance()
        {
                CFrameWnd *pWnd = new CFrameWnd;
                pWnd->Create(NULL, "HELLO");

                m_pMainWnd = pWnd;
                m_pMainWnd->ShowWindow(m_nCmdShow);
                m_pMainWnd->UpdateWindow();
                return TRUE;
        }
};

CMyApp theApp;
```

Explanation of the code:

An object representing the window shown on the screen.

The window created by the above program looks something similar to this –

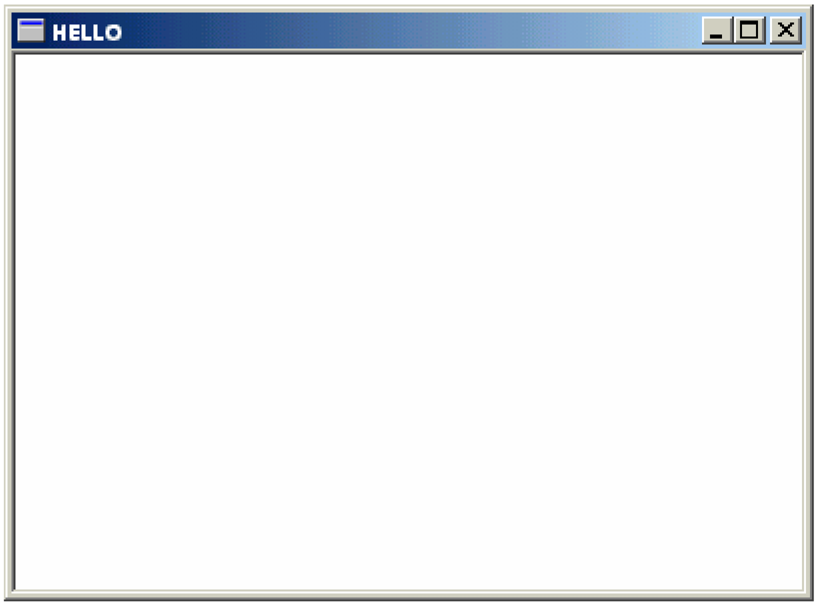


Figure 4

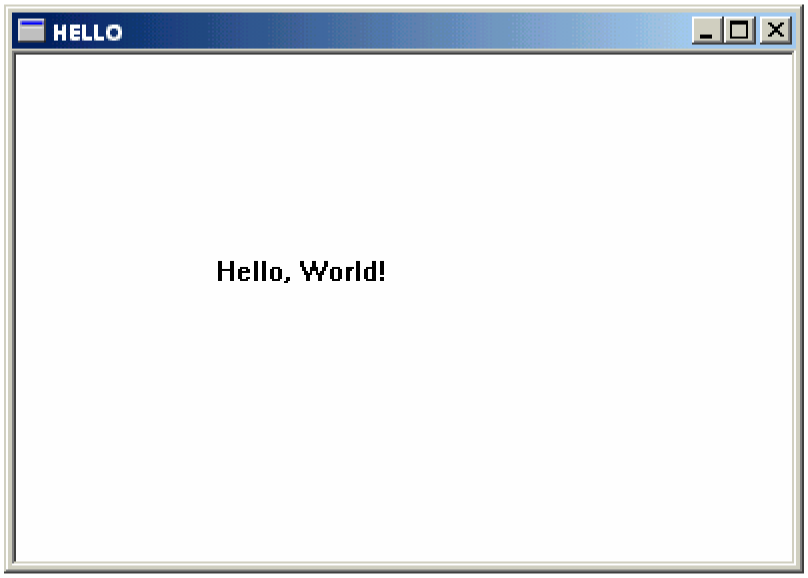
The above program makes use of another MFC class CFrameWnd which is derived from CWnd – the mother of (almost) all Window support classes (AFXWIN.H). CWnd provides two areas of functionality: (1) wrapping the regular Windows API functions (like Create and ShowWindow) and (2) providing higher-level MFC-related functionality, like default message handling.

…

But, we wanted to display the string "Hello, World!" within the window, right? This is where one of the most interesting parts of MFC comes in and that is handling messages.

"Windows shalt give messages and thou shalt handle them"

Every Window class within a Windows application requires a message-handling procedure. It is through the message handler that a window derives it's specific behavior. Whenever Windows detects an event that is pertinent to a specific window, it generates a message and calls the window's message handler with information about that event.



```
#include <afxwin.h>

class CMyWindow : public CFrameWnd
{
public:
	void OnPaint()
	{
		CPaintDC dc(this);
		dc.TextOut(100, 100, "Hello, World!", 13);
	}
	DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CMyWindow, CFrameWnd)
	ON_WM_PAINT()
END_MESSAGE_MAP()
```

```
class CMyApp : public CWinApp
{
public:
	BOOL InitInstance()
	{
		CMyWindow *pWnd = new CMyWindow;
		pWnd->Create(NULL, "HELLO");

		m_pMainWnd = pWnd;
		m_pMainWnd->ShowWindow(m_nCmdShow);
		m_pMainWnd->UpdateWindow();
		return TRUE;
	}
};

CMyApp theApp;
```

And MFC programming lived happily ever after…

This was just an introduction to the Windows support in MFC, something like a drop in the ocean. But, I hope it has given an incentive to you to explore and experiment further in the wonderful world of MFC.

- Anil K. Patro