Module 9: A Reusable Frame Window Base Class

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:

A Reusable Frame Window Base Class

Why Reusable Base Classes Are Difficult to Write
The CPersistentFrame Class
The CFrameWnd Class and the ActivateFrame() Member Function
The PreCreateWindow() Member Function
The Windows Registry
Unicode
Using the CString Class
The Position of a Maximized Window
Control Bar Status and the Registry
Static Data Members
The Default Window Rectangle
The MYMFC14 Example

A Reusable Frame Window Base Class

C++ promises programmers the ability to produce "software Lego blocks" that can be taken "off the shelf" and fitted easily into an application. The MFC Library version 6.0 classes are a good example of this kind of reusable software. This module shows you how to build your own reusable base class by taking advantage of what the MFC library already provides.

In the process of building the reusable class, you'll learn a few more things about Microsoft Windows and the MFC library. In particular, you'll see how the application framework allows access to the Windows Registry, you'll learn more about the mechanics of the CFrameWnd class, and you'll get more exposure to static class variables and the CString class.

Why Reusable Base Classes Are Difficult to Write

In a normal application, you write code for software components that solve particular problems. It's usually a simple matter of meeting the project specification. With reusable base classes, however, you must anticipate future programming needs, both your own and those of others. You have to write a class that is general and complete yet efficient and easy to use.

This module's example showed me the difficulty in building reusable software. I started out intending to write a frame class that would "remember" its window size and position. When I got into the job, I discovered that existing Windows-based programs remember whether they have been minimized to the taskbar or whether they have been maximized to full screen. Then there was the oddball case of a window that was both minimized and maximized. After that, I had to worry about the toolbar and the status bar, plus the class had to work in a dynamic link library (DLL). In short, it was surprisingly difficult to write a frame class that would do everything that a programmer might expect.

In a production programming environment, reusable base classes might fall out of the normal software development cycle. A class written for one project might be extracted and further generalized for another project. There's always the temptation, though, to cut and paste existing classes without asking, "What can I factor out into a base class?" If you're in the software business for the long term, it's beneficial to start building your library of truly reusable components.

The CPersistentFrame Class

In this module, you'll be using a class named CPersistentFrame (the files are Persist.h and Persist.cpp) that is derived from the CFrameWnd class. This CPersistentFrame class supports a persistent SDI (Single Document Interface) frame window that remembers the following characteristics.

- Window size.
- Window position.
• Maximized status.
• Minimized status.
• Toolbar and status bar enablement and position.

When you terminate an application that's built with the CPersistentFrame class, the above information is saved on disk in the Windows Registry. When the application starts again, it reads the Registry and restores the frame to its state at the previous exit. You can use the persistent view class in any SDI application, including the examples in this book. All you have to do is substitute CPersistentFrame for CFrameWnd in your application's derived frame class files.

**The CFrameWnd Class and the ActivateFrame() Member Function**

Why choose CFrameWnd as the base class for a persistent window? Why not have a persistent view class instead? In an MFC SDI application, the main frame window is always the parent of the view window. This frame window is created first, and then the control bars and the view are created as child windows. The application framework ensures that the child windows shrink and expand appropriately as the user changes the size of the frame window. It wouldn't make sense to change the view size after the frame was created.

The key to controlling the frame's size is the CFrameWnd::ActivateFrame member function. The application framework calls this virtual function (declared in CFrameWnd) during the SDI main frame window creation process (and in response to the File New and File Open commands). The framework's job is to call the CWnd::ShowWindow function with the parameter nCmdShow. ShowWindow() makes the frame window visible along with its menu, view window, and control bars. The nCmdShow parameter determines whether the window is maximized or minimized or both.

If you override ActivateFrame in your derived frame class, you can change the value of nCmdShow before passing it to the CFrameWnd::ActivateFrame function. You can also call the CWnd::SetWindowPlacement function, which sets the size and position of the frame window, and you can set the visible status of the control bars. Because all changes are made before the frame window becomes visible, no annoying flash occurs on the screen. You must be careful not to reset the frame window's position and size after every File New or File Open command. A first-time flag data member ensures that your CPersistentFrame::ActivateFrame function operates only when the application starts.

**The PreCreateWindow() Member Function**

PreCreateWindow(), declared at the CWnd level, is another virtual function that you can override to change the characteristics of your window before it is displayed. The framework calls this function before it calls ActivateFrame(). AppWizard always generates an overridden PreCreateWindow() function in your project's view and frame window classes.

This function has a CREATESTRUCT structure as a parameter, and two of the data members in this structure are style and dwExStyle. You can change these data members before passing the structure on to the base class PreCreateWindow() function. The style flag determines whether the window has a border, scroll bars, a minimize box, and so on. The dwExStyle flag controls other characteristics, such as always-on-top status. The CREATESTRUCT member lpszClass is also useful to change the window's background brush, cursor, or icon. It makes no sense to change the brush or cursor in a frame window because the view window covers the client area. If you want an ugly red view window with a special cursor, for example, you can override your view's PreCreateWindow() function like this:

```cpp
BOOL CMyView::PreCreateWindow(CREATESTRUCT& cs)
{
    if (!CView::PreCreateWindow(cs)) {
        return FALSE;
    }

cs.lpszClass = AfxRegisterWndClass(CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW,
                                   AfxGetApp()->LoadCursor(IDC_MYCURSOR),
                                   ::CreateSolidBrush(RGB(255, 0, 0)));
    if (cs.lpszClass != NULL) {
        return TRUE;
    }
    else {
        return FALSE;
    }
}
```
If you override the `PreCreateWindow()` function in your persistent frame class, windows of all derived classes will share the characteristics you programmed in the base class. Of course, derived classes can have their own overridden `PreCreateWindow()` functions, but then you'll have to be careful about the interaction between the base class and derived class functions.

**The Windows Registry**

If you've used Win16-based applications, you've probably seen INI files. You can still use INI files in Win32-based applications, but Microsoft recommends that you use the **Windows Registry** instead. The Registry is a set of system files, managed by Windows, in which Windows and individual applications can store and access permanent information. The Registry is organized as a kind of hierarchical database in which string and integer data is accessed by a multipart key.

For example, a text processing application, TEXTPROC, might need to store the most recent font and point size in the Registry. Suppose that the program name forms the root of the key (a simplification) and that the application maintains two hierarchy levels below the name. The structure looks something like this:

```
TEXTPROC
    Text formatting
        Font = Times Roman
        Points = 10
```

**Unicode**

European languages use characters that can be encoded in 8 bits, even characters with diacritics. Most Asian languages require 16 bits for their characters. Many programs use the double-byte character set (DBCS) standard: some characters use 8 bits and others 16 bits, depending on the value of the first 8 bits. DBCS is being replaced by Unicode, in which all characters are 16-bit "wide" characters. No specific Unicode character ranges are set aside for individual languages: if a character is used in both the Chinese and the Japanese languages, for example, that character appears only once in the Unicode character set.

When you look at MFC source code and the code that AppWizard generates, you'll see the types `TCHAR`, `LPTSTR`, and `LPCTSTR` and you'll see literal strings like `_T("string")`. You are looking at Unicode macros. If you build your project without defining `_UNICODE`, the compiler generates code for ordinary 8-bit ANSI characters (`CHAR`) and pointers to 8-bit character arrays (`LPSTR`, `LPCSTR`). If you do define `_UNICODE`, the compiler generates code for 16-bit Unicode characters (`WCHAR`), pointers (`LPWSTR`, `LPCWSTR`), and literals (L"wide string"). The `_UNICODE` preprocessor symbol also determines which Windows functions your program calls. Many Win32 functions have two versions. When your program calls `CreateWindowEx()`, for example, the compiler generates code to call either `CreateWindowExA()` (with ANSI parameters) or `CreateWindowExW()` (with Unicode parameters). In Microsoft Windows NT, which uses Unicode internally, `CreateWindowExW()` passes all parameters straight through, but `CreateWindowExA()` converts ANSI string and character parameters to Unicode. In Microsoft Windows 95, which uses ANSI internally, `CreateWindowExW()` is a stub that returns an error and `CreateWindowExA()` passes the parameters straight through. If you want to create a Unicode application, you should target it for Windows NT and use the macros throughout. You can write Unicode applications for Windows 95, but you'll do extra work to call the "A" versions of the Win32 functions. **Component Object Model** (COM) calls (except Data Access Object - DAO) always use wide characters. Although Win32 functions are available for converting between ANSI and Unicode, if you're using the CString class you can rely on a wide character constructor and the `AllocSysString()` member function to do the conversions. For simplicity, this book's example programs use ANSI only. The code AppWizard generated uses Unicode macros, but the code I wrote uses 8-bit literal strings and the `char`, `char*`, and `const char*` types. The MFC library provides four `CWinApp` member functions, holdovers from the days of INI files, for accessing the Registry. Starting with Visual C++ version 5.0, AppWizard generates a call to `CWinApp::SetRegistryKey` in your application's `InitInstance()` function as shown here:

```cpp
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```
If you remove this call, your application will not use the Registry but will create and use an INI file in the Windows directory. The SetRegistryKey() function's string parameter establishes the top of the hierarchy, and the following Registry functions define the bottom two levels: called **heading name** and **entry name**:

- GetProfileInt()
- WriteProfileInt()
- GetProfileString()
- WriteProfileString()

These functions treat Registry data as either CString objects or **unsigned integers**. If you need floating-point values as entries, you must use the string functions and do the conversion yourself. All the functions take a heading name and an entry name as parameters. In the example shown above, the heading name is **Text Formatting** and the entry names are **Font** and **Points**.

To use the Registry access functions, you need a pointer to the application object. The global function AfxGetApp() does the job. With the previous sample Registry, the Font and Points entries were set with the following code:

```
AfxGetApp()->WriteProfileString("Text formatting", "Font", "Times Roman");
AfxGetApp()->WriteProfileInt("Text formatting", "Points", 10);
```

You'll see a real Registry example in MYMFC14, and you'll learn to use the **Windows Regedit** program to examine and edit the Registry. The application framework stores a list of most recently used files in the Registry under the heading **Recent File List**.

### Using the CString Class

The MFC CString class is a significant de facto extension to the C++ language. As the Microsoft Foundation Classes and **Templates** section of the online help points out, the CString class has many useful operators and member functions, but perhaps its most important feature is its dynamic memory allocation. You never have to worry about the size of a CString object. The statements here represent typical uses of CString objects:

```
CString strFirstName("Elvis");
CString strLastName("Presley");
CString strTruth = strFirstName + " " + strLastName; // concatenation
strTruth += " is alive";
ASSERT(strTruth == "Elvis Presley is alive");
ASSERT(strTruth.Left(5) == strFirstName);
ASSERT(strTruth[2] == 'v'); // subscript operator
```

In a perfect world, C++ programs would use all CString objects and never use ordinary zero-terminated character arrays. Unfortunately, many runtime library functions still use character arrays, so programs must always mix and match their string representations. Fortunately, the CString class provides a const char* operator that converts a CString object to a character pointer. Many of the MFC library functions have const char* parameters. Take the global AfxMessageBox() function, for example. Here is one of the function's prototypes:

```
int AFXAPI AfxMessageBox(LPCTSTR lpszText, UINT nType = MB_OK, UINT nIDHelp = 0);
```

**Note:** LPCTSTR is not a pointer to a CString object but rather is a Unicode-enabled replacement for const char*. You can call AfxMessageBox() this way:

```
char szMessageText[] = "Unknown error";
AfxMessageBox(szMessageText);
```

Or you can call it this way:

```
CString strMessageText("Unknown ;error");
AfxMessageBox(strMessageText);
```

Now suppose you want to generate a formatted string. CString::Format does the job, as shown here:
Suppose you want direct write access to the characters in a CString object. If you write code like this:

```cpp
CString strTest("test");
strncpy(strTest, "T", 1);
```

You'll get a compile error because the first parameter of `strncpy()` is declared `char*`, not `const char*`. The `CString::GetBuffer` function "locks down" the buffer with a specified size and returns a `char*`. You must call the `ReleaseBuffer()` member function later to make the string dynamic again. The correct way to capitalize the T is shown here.

```cpp
CString strTest("test");
strncpy(strTest.GetBuffer(5), "T", 1);
strTest.ReleaseBuffer();
ASSERT(strTest == "Test");
```

The `const char*` operator takes care of converting a `CString` object to a constant character pointer, but what about conversion in the other direction? It so happens that the `CString` class has a constructor that converts a constant character pointer to a `CString` object, and it has a set of overloaded operators for these pointers. That's why statements such as the following work:

```cpp
strTruth += " is alive";
```

The special constructor works with functions that take a `CString` reference parameter, such as `CDC::TextOut`. In the following statement, a temporary `CString` object is created on the calling program's stack and then the object's address is passed to `TextOut()`:

```cpp
pDC->TextOut(50, 50, "Hello, MFC world!");
```

It's more efficient to use the other overloaded version of `CDC::TextOut` if you're willing to count the characters:

```cpp
pDC->TextOut(50, 50, "Hello, MFC world!", 17);
```

If you're writing a function that takes a string parameter, you've got some design choices. Here are some programming rules.

- If the function doesn't change the contents of the string and you're willing to use C runtime functions such as `strcpy()`, use a `const char*` parameter.
- If the function doesn't change the contents of the string but you want to use `CString` member functions inside the function, use a `const CString&` parameter.
- If the function changes the contents of the string, use a `CString&` parameter.

### The Position of a Maximized Window

As a Windows user, you know that you can maximize a window from the system menu or by clicking a button at the top right corner of the window. You can return a maximized window to its original size in a similar fashion. It's obvious that a maximized window remembers its original size and position. The `CWnd` function `GetWindowRect()` retrieves the screen coordinates of a window. If a window is maximized, `GetWindowRect()` returns the coordinates of the screen rather than the window's unmaximized coordinates. If a persistent frame class is to work for maximized windows, it has to know the window's unmaximized coordinates. `CWnd::GetWindowPlacement` retrieves the unmaximized coordinates together with some flags that indicate whether the window is currently minimized or maximized or both. The companion `SetWindowPlacement()` function lets you set the maximized and minimized status and the size and position of the window. To calculate the position of the top left corner of a maximized window, you need to account for
the window's border size, obtainable from the Win32 GetSystemMetrics() function. See Listing 1 for the CPersistentFrame::ActivateFrame code for an example of how SetWindowPlacement() is used.

Control Bar Status and the Registry

The MFC library provides two CFrameWnd member functions, SaveBarState() and LoadBarState(), for saving and loading control bar status to and from the Registry. These functions process the size and position of the status bar and docked toolbars. They don't process the position of floating toolbars, however.

Static Data Members

The CPersistentFrame class stores its Registry key names in static const char array data members. What were the other storage choices? String resource entries won't work because the strings need to be defined with the class itself. String resources make sense if CPersistentFrame is made into a DLL, however. Global variables are generally not recommended because they defeat encapsulation. Static CString objects don't make sense because the characters must be copied to the heap when the program starts. An obvious choice would have been regular data members. But static data members are better because, as constants, they are segregated into the program's read-only data section and can be mapped to multiple instances of the same program. If the CPersistentFrame class is part of a DLL, all processes that are using the DLL can map the character arrays. Static data members are really global variables, but they are scoped to their class so there's no chance of name collisions.

The Default Window Rectangle

You're used to defining rectangles with device or logical coordinates. A CRect object constructed with the statement:

CRect rect(CW_USEDEFAULT, CW_USEDEFAULT, 0, 0);

has a special meaning. When Windows creates a new window with this special rectangle, it positions the window in a cascade pattern with the top left corner below and to the right of the window most recently created. The right and bottom edges of the window are always within the display's boundaries. The CFrameWnd class's static rectDefault data member is constructed using CW_USEDEFAULT this way, so it contains the special rectangle. The CPersistentFrame class declares its own rectDefault default window rectangle with a fixed size and position as a static data member, thus hiding the base class member.

The MYMFC14 Example

The MYMFC14 program illustrates the use of a persistent frame window class, CPersistentFrame. Listing 1 shows the contents of the files Persist.h and Persist.cpp. In this example, you'll insert the new frame class into an AppWizard-generated SDI application. MYMFC14 is a "do-nothing" application, but you can insert the persistent frame class into any of your own SDI "do-something" applications.

#include <windows.h>

// Persist.h
// PersistentFrame.H

#ifndef _INSIDE_VISUAL_CPP_PERSISTENT_FRAME
#define _INSIDE_VISUAL_CPP_PERSISTENT_FRAME

class CPersistentFrame : public CFrameWnd
{ // remembers where it was on the desktop
  DECLARE_DYNAMIC(CPersistentFrame)
  DECLARE_DYNCREATE(CPersistentFrame)
private:
  static const CRect s_rectDefault;
  static const char s_profileHeading[];
  static const char s_profileRect[];
  static const char s_profileIcon[];
  static const char s_profileMax[];
  static const char s_profileTool[];
  static const char s_profileStatus[];
};

#include "Persist.h"
BOOL m_bFirstTime;
protected: // Create from serialization only
CPersistentFrame();
~CPersistentFrame();
//}}AFX_VIRTUAL

virtual void ActivateFrame(int nCmdShow = -1);
protected:
//}}AFX_VIRTUAL

DECLARE_MESSAGE_MAP()

CPersistentFrame()
{
    m_bFirstTime = TRUE;
}

CPersistentFrame::~CPersistentFrame()
{
}

void CPersistentFrame::OnDestroy()
{
    CString strText;
    BOOL bIconic, bMaximized;
    WINDOWPLACEMENT wndpl;
    wndpl.length = sizeof(WINDOWPLACEMENT);
}
// gets current window position and iconized/maximized status
BOOL bRet = GetWindowPlacement(&wndpl);
if (wndpl.showCmd == SW_SHOWNORMAL) {
    bIconic = FALSE;
    bMaximized = FALSE;
}
else if (wndpl.showCmd == SW_SHOWNMAXIMIZED) {
    bIconic = FALSE;
    bMaximized = TRUE;
}
else if (wndpl.showCmd == SW_SHOWNMINIMIZED) {
    bIconic = TRUE;
    if (wndpl.flags) {
        bMaximized = TRUE;
    }
    else {
        bMaximized = FALSE;
    }
}
strText.Format("%04d %04d %04d %04d",
            wndpl.rcNormalPosition.left,
            wndpl.rcNormalPosition.top,
            wndpl.rcNormalPosition.right,
            wndpl.rcNormalPosition.bottom);
AfxGetApp()->WriteProfileString(s_profileHeading,
            s_profileRect, strText);
AfxGetApp()->WriteProfileInt(s_profileHeading,
            s_profileIcon, bIconic);
AfxGetApp()->WriteProfileInt(s_profileHeading, s_profileMax,
            bMaximized);
SaveBarState(AfxGetApp()->m_pszProfileName);
CFrameWnd::OnDestroy();
}

void CPersistentFrame::ActivateFrame(int nCmdShow)
{
    CString strText;
    BOOL bIconic, bMaximized;
    UINT flags;
    WINDOWPLACEMENT wndpl;
    CRect rect;
    if (m_bFirstTime) {
        m_bFirstTime = FALSE;
        strText = AfxGetApp()->
            >GetProfileString(s_profileHeading, s_profileRect);
        if (!strText.IsEmpty()) {
            rect.left = atoi((const char*) strText);
            rect.top = atoi((const char*) strText + 5);
            rect.right = atoi((const char*) strText + 10);
            rect.bottom = atoi((const char*) strText + 15);
        }
        else {
            rect = s_rectDefault;
        }
        bIconic = AfxGetApp()->GetProfileInt(s_profileHeading,
            s_profileIcon, 0);
        bMaximized = AfxGetApp()->
            >GetProfileInt(s_profileHeading, s_profileMax, 0);
        if (bIconic) {
            nCmdShow = SW_SHOWMINNOACTIVE;
        }
        else {
            flags = WPF_RESTORETOMAXIMIZED;
        }
    }
}
else {
    flags = WPF_SETMINPOSITION;
}
else {
    if (bMaximized) {
        nCmdShow = SW_SHOWMAXIMIZED;
        flags = WPF_RESTORETOMAXIMIZED;
    }
    else {
        nCmdShow = SW_NORMAL;
        flags = WPF_SETMINPOSITION;
    }
}
wndpl.length = sizeof(WINDOWPLACEMENT);
wndpl.showCmd = nCmdShow;
wndpl.flags = flags;
wndpl.ptMinPosition = CPoint(0, 0);
wndpl.ptMaxPosition = CPoint(-::GetSystemMetrics(SM_CXBORDER), -::GetSystemMetrics(SM_CYBORDER));
wndpl.rcNormalPosition = rect;
LoadBarState(AfxGetApp()->m_pszProfileName);
// sets window's position and minimized/maximized status
BOOL bRet = SetWindowPlacement(&wndpl);
CFrameWnd::ActivateFrame(nCmdShow);
}

Listing 1: The CPersistentView class listing.

Here are the steps for building the MYMFC14 example program.

Run AppWizard to generate mfcproject\mymfc14 (or any other directory that you have designated your Visual C++ project for). Accept all default settings but two: select Single Document and deselect Printing and Print Preview and ActiveX Controls. The options and the default class names are shown in the following illustration.
Modify MainFrm.h. You must change the base class of CMainFrame. To do this, simply change the line:

```cpp
class CMainFrame : public CFrameWnd
```

To:

```cpp
class CMainFrame : public CPersistentFrame
```

Listing 1.

Also, add the line:

```cpp
#include "persist.h"
```
// MainFrm.h : interface of the CMainFrame class

#include "persist.h"
#if !defined(AFX_MAINFRM_H__FEC13E64_E044_4034_905C_DE)
#define AFX_MAINFRM_H__FEC13E64_E044_4034_905C_DE
#endif

Listing 2.

Modify **MainFrm.cpp**. Globally replace all occurrences of `CFrameWnd` with `CPersistentFrame`. Use the find and replace menu.

Modify **mymfc14.cpp**. Replace the line:

```cpp
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

With the line:

```cpp
SetRegistryKey("Programming using Visual C++ and MFC");
```
Listing 3.

Add the `Persist.h` and `Persist.cpp` file to the project. You can create the `Persist.h` and `Persist.cpp` files and add to the mymfc14 project as shown below and don’t forget to copy the source code for the `Persist.h` and `Persist.cpp` as shown in the previous listing into the files respectively then save those files. Don’t forget to make sure the `Add to project` check box is ticked.

Figure 4: Adding new files to the project.
Rebuild the ClassWizard file to include the new CPersistentFrame class.

Use Windows Explorer to delete the ClassWizard file mymfc14.clw.

Back in Visual C++, choose ClassWizard from the View menu. Follow Visual C++’s instructions if it asks you to close any files. Click Yes when asked if you would like to rebuild the CLW file.
The **Select Source Files** dialog box will appear. Make sure all of the header and source files are listed in the **Files In Project** box, as shown in the following illustration.

![Select Source Files - mymfc14.clw](image)

Figure 8: Rebuilding the CLW file dialog prompt.

Then click **OK** to regenerate the **CLW** file. Notice that **CPersistentFrame** is now integrated into ClassWizard. You'll now be able to map messages and override virtual functions in the **CPersistentFrame** class.

![Select Source Files - mymfc14.clw](image)

Figure 9: Rebuilding the CLW file to integrate newly created class.
Figure 10: The CPersistentFrame is now integrated into ClassWizard after the CLW rebuilding.

Build and test the MYMFC14 application. Resize and move the application's frame window, and then close the application.

Figure 11: MYMFC14 program output with persistent property.

When you restart the application, does its window open at the same location at which it was closed? Experiment with maximizing and minimizing, and then change the status and position of the control bars. Does the persistent frame remember its settings?

Save the CPersistentFrame class as a Gallery component for future use. In the ClassView window, right-click on CPersistentFrame and select Add To Gallery.
Figure 12: Saving our new created CPersistentFrame class for future use, class reusability.

Bring up the **Components And Controls Gallery** by choosing **Add To Project** from the **Project** menu and then choosing **Components And Controls**.

Figure 13: **Component and Controls** menu used to save newly created class.

Notice that Visual C++ created the file *Persistent Frame.ogx* in a folder named *mymfc14*.
Change this folder's name to **Persistent Frame**. Now you can add the `CPersistentFrame` class to any project by simply adding **Persistent Frame.ogx**. We will add `CPersistentFrame` to **mymfc22A** project later.
Examine the Windows Registry. Run the Windows `regedit.exe` (or `regedt32`) program. Navigate to the HKEY_CURRENT_USER\Software\Programming Visual C++ and MFC\mymfc14 key.

You should see data values similar to those shown in the following illustration.

Notice the relationship between the Registry key and the `SetRegistryKey()` function parameter, "Programming using Visual C++ and MFC". If you supply an empty string as the `SetRegistryKey()` parameter, the program name (mymfc14, in this case) is positioned directly below the Software key.

**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
5. Win32 programming Tutorial.
7. Unicode and Multibyte character set: Story and program examples.