

Introduction to MFC Programming with Visual C++ Version 6.x

Introduction to MFC

An Example

One of the best ways to begin understanding the structure and style of a typical MFC program is to enter, compile, and run a small example. The listing below contains a simple "hello world" program. If this is the first time you've seen this sort of program, it probably will not make a lot of sense initially. Don't worry about that. We will examine the code in detail in the next tutorial. For now, the goal is to use the Visual C++ environment to create, compile and execute this simple program.

```
//hello.cpp
#include <afxwin.h>
// Declare the application class
class CHelloApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
// Create an instance of the application class
CHelloApp HelloApp;
// Declare the main window class
class CHelloWindow : public CFrameWnd
{
    CStatic* cs;
public:
    CHelloWindow();
};
// The InitInstance function is called each
// time the application first executes.
BOOL CHelloApp::InitInstance()
{
    m_pMainWnd = new CHelloWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
// The constructor for the window class
CHelloWindow::CHelloWindow()
{
    // Create the window itself
    Create(NULL,
        "Hello World!",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));
    // Create a static label
    cs = new CStatic();
    cs->Create("hello world",
```

```


WS_CHILD|WS_VISIBLE|SS_CENTER,
CRect(50,80,150,150),
this);
}

```

This small program does three things. First, it creates an "application object." Every MFC program you write will have a single application object that handles the initialization details of MFC and Windows. Next, the application creates a single window on the screen to act as the main application window. Finally, inside that window the application creates a single static text label containing the words "hello world". We will look at this program in detail in the next tutorial to gain a complete understanding of its structure.

The steps necessary to enter and compile this program are straightforward. If you have not yet installed Visual C++ on your machine, do so now. You will have the option of creating standard and custom installations. For the purposes of these tutorials a standard installation is suitable and after answering two or three simple questions the rest of the installation is quick and painless.

Start VC++ by double clicking on its icon in the Visual C++ group of the Program Manager. If you have just installed the product, you will see an empty window with a menu bar. If VC++ has been used before on this machine, it is possible for it to come up in several different states because VC++ remembers and automatically reopens the project and files in use the last time it exited. What we want right now is a state where it has no project or code loaded. If the program starts with a dialog that says it was unable to find a certain file, clear the dialog by clicking the "No" button. Go to the **Window** menu and select the **Close All** option if it is available. Go to the **File** menu and select the **Close** option if it is available to close any remaining windows. Now you are at the proper starting point. If you have just installed the package, you will see a window that looks something like this:

This screen can be rather intimidating the first time you see it. To eliminate some of the intimidation, click on the lower of the two "x" buttons () that you see in the upper right hand corner of the screen if it is available. This action will let you close the "InfoViewer Topic" window. If you want to get rid of the InfoViewer toolbar as well, you can drag it so it docks somewhere along the side of the window, or close it and later get it back by choosing the **Customize** option in the **Tools** menu.

What you see now is "normal". Along the top is the menu bar and several toolbars. Along the left side are all of the topics available from the on-line book collection (you might want to explore by double clicking on several of the items you see there - the collection of information found in the on-line books is gigantic). Along the bottom is a status window where various messages will be displayed.

Now what? What you would like to do is type in the above program, compile it and run it. Before you start, switch to the File Manager (or the MS-DOS prompt) and make sure your drive has at least five megabytes of free space available. Then take the following steps.

Creating a Project and Compiling the Code

In order to compile any code in Visual C++, you have to create a *project*. With a very small program like this the project seems like overkill, but in any real program the project concept is quite useful. A project holds three different types of information:

1. It remembers all of the source code files that combine together to create one executable. In this simple example, the file HELLO.CPP will be the only source file, but in larger applications you often break the code up into several different files to make it easier to understand (and also to make it possible for several

people to work on it simultaneously). The project maintains a list of the different source files and compiles all of them as necessary each time you want to create a new executable.

2. It remembers compiler and linker options particular to this specific application. For example, it remembers which libraries to link into the executable, whether or not you want to use pre-compiled headers, and so on.
3. It remembers what type of project you wish to build: a console application, a windows application, etc.

If you are familiar with makefiles, then it is easy to think of a project as a machine-generated makefile that has a very easy-to-understand user interface to manipulate it. For now we will create a very simple project file and use it to compile HELLO.CPP.

To create a new project for HELLO.CPP, choose the **New** option in the **File** menu. Under the **Projects** tab, highlight **Win32 Application**. In the Location field type an appropriate path name or click the Browse button. Type the word "hello" in for the project name, and you will see that word echoed in the Location field as well. Click the OK button. In the next window, use the default selection "An empty project", click "Finish", then click "OK" once more in the next window. Notice there is an option for the typical "Hello World" application, however it skips a few important steps you are about to take. Visual C++ will create a new subdirectory named HELLO and place the project files named HELLO.OPT, HELLO.NCB, HELLO.DSP, and HELLO.DSW in that directory. If you quit and later want to reopen the project, double-click on HELLO.DSW.

The area along the left side of the screen will now change so that three tabs are available. The InfoView tab is still there, but there is now also a ClassView and a FileView tab. The ClassView tab will show you a list of all of the classes in your application and the FileView tab gives you a list of all of the files in the project.

Now it is time to type in the code for the program. In the **File** menu select the **New** option to create a new editor window. In the dialog that appears, make sure the **Files** tab is active and request a "C++ Source File". Make sure the "Add to Project" option is checked for Project "hello", and enter "hello" for "File name". Visual C++ comes with its own intelligent C++ editor, and you will use it to enter the program shown above. Type (copy/paste) the code in the listing into the editor window. You will find that the editor automatically colors different pieces of text such as comments, key words, string literals, and so on. If you want to change the colors or turn the coloring off, go to the **Options** option in the **Tools** menu, choose the **Format** tab and select the **Source Windows** option from the left hand list. If there is some aspect of the editor that displeases you, you may be able to change it using the **Editor** tab of the **Options** dialog.

After you have finished entering the code, save the file by selecting the **Save** option in the **File** menu. Save it to a file named HELLO.CPP in the new directory Visual C++ created.

In the area on the left side of the screen, click the FileView tab and expand the tree on the icon labeled "hello files", then expand the tree on the folder icon labeled "Source Files". You will see the file named HELLO.CPP. Click on the ClassView tab and expand the "hello classes" tree and you will see the classes in the application. You can remove a file from a project at any time by going to the FileView, clicking the file, and pressing the delete button.

Finally, you must now tell the project to use the MFC library. If you omit this step the project will not link properly, and the error messages that the linker produces will not help one bit. Choose the **Settings** option in the **Project** menu. Make sure that the **General** tab is selected in the tab at the top of the dialog that appears. In the **Microsoft Foundation Classes** combo box, choose the third option: "Use MFC in a Shared DLL." Then close the dialog.

Introduction to MFC Programming with Visual C++ Version 6.x

A Simple MFC Program

In this tutorial we will examine a simple MFC program piece by piece to gain an understanding of its structure and conceptual framework. We will start by looking at MFC itself and then examine how MFC is used to create applications.

An Introduction to MFC

MFC is a large and extensive C++ class hierarchy that makes Windows application development significantly easier. MFC is compatible across the entire Windows family. As each new version of Windows comes out, MFC gets modified so that old code compiles and works under the new system. MFC also gets extended, adding new capabilities to the hierarchy and making it easier to create complete applications.

The advantage of using MFC and C++ - as opposed to directly accessing the Windows API from a C program-is that MFC already contains and encapsulates all the normal "boilerplate" code that all Windows programs written in C must contain. Programs written in MFC are therefore much smaller than equivalent C programs. On the other hand, MFC is a fairly thin covering over the C functions, so there is little or no performance penalty imposed by its use. It is also easy to customize things using the standard C calls when necessary since MFC does not modify or hide the basic structure of a Windows program.

The best part about using MFC is that it does all of the hard work for you. The hierarchy contains thousands and thousands of lines of correct, optimized and robust Windows code. Many of the member functions that you call invoke code that would have taken you weeks to write yourself. In this way MFC tremendously accelerates your project development cycle.

MFC is fairly large. For example, Version 4.0 of the hierarchy contains something like 200 different classes. Fortunately, you don't need to use all of them in a typical program. In fact, it is possible to create some fairly spectacular software using only ten or so of the different classes available in MFC. The hierarchy is broken into several different class categories which include (but is not limited to):

- Application Architecture
- Graphical Drawing and Drawing Objects
- File Services
- Exceptions
- Structures - Lists, Arrays, Maps
- Internet Services
- OLE 2
- Database
- General Purpose

We will concentrate on visual objects in these tutorials. The list below shows the portion of the class hierarchy that deals with application support and windows support.

- CObject
- CCmdTarget
- CWinThread
- CWinApp
- CWnd

- CFrameWnd
- CDialog
- CView
- CStatic
- CButton
- CListBox
- CComboBox
- CEdit
- CScrollBar

There are several things to notice in this list. First, most classes in MFC derive from a base class called **CObject**. This class contains data members and member functions that are common to most MFC classes. The second thing to notice is the simplicity of the list. The **CWinApp** class is used whenever you create an application and it is used only once in any program. The **CWnd** class collects all the common features found in windows, dialog boxes, and controls. The **CFrameWnd** class is derived from **CWnd** and implements a normal framed application window. **CDialog** handles the two normal flavors of dialogs: modeless and modal respectively. **CView** is used to give a user access to a document through a window. Finally, Windows supports six native control types: static text, editable text, push buttons, scroll bars, lists, and combo boxes (an extended form of list). Once you understand this fairly small number of pieces, you are well on your way to a complete understanding of MFC. The other classes in the MFC hierarchy implement other features such as memory management, document control, data base support, and so on.

To create a program in MFC, you either use its classes directly or, more commonly, you derive new classes from the existing classes. In the derived classes you create new member functions that allow instances of the class to behave properly in your application. You can see this derivation process in the simple program we used in Tutorial 1, which is described in greater detail below. Both **CHelloApp** and **CHelloWindow** are derived from existing MFC classes.

Designing a Program

Before discussing the code itself, it is worthwhile to briefly discuss the program design process under MFC. As an example, imagine that you want to create a program that displays the message "Hello World" to the user. This is obviously a very simple application but it still requires some thought.

A "hello world" application first needs to create a window on the screen that holds the words "hello world". It then needs to get the actual "hello world" words into that window. Three objects are required to accomplish this task:

1. An application object which initializes the application and hooks it to Windows. The application object handles all low-level event processing.
2. A window object that acts as the main application window.
3. A static text object which will hold the static text label "hello world".

Every program that you create in MFC will contain the first two objects. The third object is unique to this particular application. Each application will define its own set of user interface objects that display the application's output as well as gather input from the user.

Once you have completed the user interface design and decided on the controls necessary to implement the interface, you write the code to create the controls on the screen. You also write the code that handles the messages generated by these controls as they are manipulated by the user. In the case of a "hello world"

application, only one user interface control is necessary. It holds the words "hello world". More realistic applications may have hundreds of controls arranged in the main window and dialog boxes.

It is important to note that there are actually two different ways to create user controls in a program. The method described here uses straight C++ code to create the controls. In a large application, however, this method becomes painful. Creating the controls for an application containing 50 or 100 dialogs using C++ code to do it would take an eon. Therefore, a second method uses *resource files* to create the controls with a graphical dialog editor. This method is much faster and works well on most dialogs.

Understanding the Code for "hello world"

The listing below shows the code for the simple "hello world" program that you entered, compiled and executed in Tutorial 1. Line numbers have been added to allow discussion of the code in the sections that follow. By walking through this program line by line, you can gain a good understanding of the way MFC is used to create simple applications.

If you have not done so already, please compile and execute the code below by following the instructions given in Tutorial 1.

```
1 //hello.cpp

2 #include <afxwin.h>

3 // Declare the application class
4 class CHelloApp : public CWinApp
5 {
6     public:
7         virtual BOOL InitInstance();
8 };

9 // Create an instance of the application class
10 CHelloApp HelloApp;

11 // Declare the main window class
12 class CHelloWindow : public CFrameWnd
13 {
14     CStatic* cs;
15     public:
16     CHelloWindow();
17 };

18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22     m_pMainWnd = new CHelloWindow();
23     m_pMainWnd->ShowWindow(m_nCmdShow);
24     m_pMainWnd->UpdateWindow();
```

```

25     return TRUE;
26 }

27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29 {
30     // Create the window itself
31     Create(NULL,
32           "Hello World!",
33           WS_OVERLAPPEDWINDOW,
34           CRect(0,0,200,200));
35     // Create a static label
36     cs = new CStatic();
37     cs->Create("hello world",
38             WS_CHILD|WS_VISIBLE|SS_CENTER,
39             CRect(50,80,150,150),
40             this);
41 }

```

Take a moment and look through this program. Get a feeling for the "lay of the land." The program consists of six small parts, each of which does something important.

The program first includes `afxwin.h` (line 2). This header file contains all the types, classes, functions, and variables used in MFC. It also includes other header files for such things as the Windows API libraries.

Lines 3 through 8 derive a new application class named **CHelloApp** from the standard **CWinApp** application class declared in MFC. The new class is created so the **InitInstance** member function in the **CWinApp** class can be overridden. **InitInstance** is a virtual function that is called as the application begins execution.

In Line 10, the code declares an instance of the application object as a global variable. This instance is important because it causes the program to execute. When the application is loaded into memory and begins running, the creation of that global variable causes the default constructor for the **CWinApp** class to execute. This constructor automatically calls the **InitInstance** function defined in lines 18 through 26.

In lines 11 through 17, the **CHelloWindow** class is derived from the **CFrameWnd** class declared in MFC. **CHelloWindow** acts as the application's window on the screen. A new class is created so that a new constructor, destructor, and data member can be implemented.

Lines 18 through 26 implement the **InitInstance** function. This function creates an instance of the **CHelloWindow** class, thereby causing the constructor for the class in Lines 27 through 41 to execute. It also gets the new window onto the screen.

Lines 27 through 41 implement the window's constructor. The constructor actually creates the window and then creates a static control inside it.

An interesting thing to notice in this program is that there is no **main** or **WinMain** function, and no apparent event loop. Yet we know from executing it in Tutorial 1 that it processed events. The window could be minimized and maximized, moved around, and so on. All this activity is hidden in the main application class **CWinApp** and we therefore don't have to worry about it—event handling is totally automatic and invisible in MFC.

The following sections describe the different pieces of this program in more detail. It is unlikely that all of this information will make complete sense to you right now: It's best to read through it to get your first exposure to the concepts. In Tutorial 3, where a number of specific examples are discussed, the different pieces will come together and begin to clarify themselves. **The Application Object**

Every program that you create in MFC will contain a single application object that you derive from the **CWinApp** class. This object must be declared globally (line 10) and can exist only once in any given program.

An object derived from the **CWinApp** class handles initialization of the application, as well as the main event loop for the program. The **CWinApp** class has several data members, and a number of member functions. For now, almost all are unimportant. If you would like to browse through some of these functions however, search for **CWinApp** in the MFC help file by choosing the **Search** option in the **Help** menu and typing in "CWinApp". In the program above, we have overridden only one virtual function in **CWinApp**, that being the **InitInstance** function.

The purpose of the application object is to initialize and control your application. Because Windows allows multiple instances of the same application to run simultaneously, MFC breaks the initialization process into two parts and uses two functions-**InitApplication** and **InitInstance**-to handle it. Here we have used only the **InitInstance** function because of the simplicity of the application. It is called each time a new instance of the application is invoked. The code in Lines 3 through 8 creates a class called **CHelloApp** derived from **CWinApp**. It contains a new **InitInstance** function that overrides the existing function in **CWinApp** (which does nothing):

```
3 // Declare the application class
4 class CHelloApp : public CWinApp
5 {
6     public:
7         virtual BOOL InitInstance();
8 };
```

Inside the overridden **InitInstance** function at lines 18 through 26, the program creates and displays the window using **CHelloApp**'s data member named **m_pMainWnd**:

```
18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22     m_pMainWnd = new CHelloWindow();
23     m_pMainWnd->ShowWindow(m_nCmdShow);
24     m_pMainWnd->UpdateWindow();
25     return TRUE;
26 }
```

The **InitInstance** function returns a TRUE value to indicate that initialization completed successfully. Had the function returned a FALSE value, the application would terminate immediately. We will see more details of the window initialization process in the next section.

When the application object is created at line 10, its data members (inherited from **CWinApp**) are automatically initialized. For example, **m_pszAppName**, **m_lpCmdLine**, and **m_nCmdShow** all contain appropriate values. See the MFC help file for more information. We'll see a use for one of these variables in a moment.

The Window Object

MFC defines two types of windows: 1) frame windows, which are fully functional windows that can be re-sized, minimized, and so on, and 2) dialog windows, which are not re-sizable. A frame window is typically used for the main application window of a program.

In the code shown in listing 2.1, a new class named **CHelloWindow** is derived from the **CFrameWnd** class in lines 11 through 17:

```
11 // Declare the main window class
12 class CHelloWindow : public CFrameWnd
13 {
14     CStatic* cs;
15     public:
16     CHelloWindow();
17 };
```

The derivation contains a new constructor, along with a data member that will point to the single user interface control used in the program. Each application that you create will have a unique set of controls residing in the main application window. Therefore, the derived class will have an overridden constructor that creates all the controls required in the main window. Typically this class will also have an overridden destructor to delete them when the window closes, but the destructor is not used here. In Tutorial 4, we will see that the derived window class will also declare a message handler to handle messages that these controls produce in response to user events.

Typically, any application you create will have a single main application window. The **CHelloApp** application class therefore defines a data member named **m_pMainWnd** that can point to this main window. To create the main window for this application, the **InitInstance** function (lines 18 through 26) creates an instance of **CHelloWindow** and uses **m_pMainWnd** to point to the new window. Our **CHelloWindow** object is created at line 22:

```
18 // The InitInstance function is called each
19 // time the application first executes.
20 BOOL CHelloApp::InitInstance()
21 {
22     m_pMainWnd = new CHelloWindow();
23     m_pMainWnd->ShowWindow(m_nCmdShow);
24     m_pMainWnd->UpdateWindow();
25     return TRUE;
26 }
```

Simply creating a frame window is not enough, however. Two other steps are required to make sure that the new window appears on screen correctly. First, the code must call the window's **ShowWindow** function to make the window appear on screen (line 23). Second, the program must call the **UpdateWindow** function to make sure that each control, and any drawing done in the interior of the window, is painted correctly onto the screen (line 24).

You may wonder where the **ShowWindow** and **UpdateWindow** functions are defined. For example, if you wanted to look them up to learn more about them, you might look in the MFC help file (use the **Search** option in the **Help** menu) at the **CFrameWnd** class description. **CFrameWnd** does not contain either of these member functions, however. It turns out that **CFrameWnd** inherits its behavior-as do all controls and windows in MFC-from

the **CWnd** class (see figure 2.1). If you refer to **CWnd** in the MFC documentation, you will find that it is a huge class containing over 200 different functions. Obviously, you are not going to master this particular class in a couple of minutes, but among the many useful functions are **ShowWindow** and **UpdateWindow**.

Since we are on the subject, take a minute now to look up the **CWnd::ShowWindow** function in the MFC help file. You do this by clicking the help file's **Search** button and entering "ShowWindow". As an alternative, find the section describing the **CWnd** class using the **Search** button, and then find the **ShowWindow** function under the Update/Painting Functions in the class member list. Notice that **ShowWindow** accepts a single parameter, and that the parameter can be set to one of ten different values. We have set it to a data member held by **CHelloApp** in our program, **m_nCmdShow** (line 23). The **m_nCmdShow** variable is initialized based on conditions set by the user at application start-up. For example, the user may have started the application from the Program Manager and told the Program Manager to start the application in the minimized state by setting the check box in the application's properties dialog. The **m_nCmdShow** variable will be set to **SW_SHOWMINIMIZED**, and the application will start in an iconic state. The **m_nCmdShow** variable is a way for the outside world to communicate with the new application at start-up. If you would like to experiment, you can try replacing **m_nCmdShow** in the call to **ShowWindow** with the different constant values defined for **ShowWindow**. Recompile the program and see what they do.

Line 22 initializes the window. It allocates memory for it by calling the **new** function. At this point in the program's execution the constructor for the **CHelloWindow** is called. The constructor is called whenever an instance of the class is allocated. Inside the window's constructor, the window must create itself. It does this by calling the **Create** member function for the **CFrameWnd** class at line 31:

```
27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29 {
30     // Create the window itself
31     Create(NULL,
32         "Hello World!",
33         WS_OVERLAPPEDWINDOW,
34         CRect(0,0,200,200));
```

Four parameters are passed to the create function. By looking in the MFC documentation you can see the different types. The initial NULL parameter indicates that a default class name be used. The second parameter is the title of the window that will appear in the title bar. The third parameter is the style attribute for the window. This example indicates that a normal, overlappable window should be created. Style attributes are covered in detail in Tutorial 3. The fourth parameter specifies that the window should be placed onto the screen with its upper left corner at point 0,0, and that the initial size of the window should be 200 by 200 pixels. If the value **rectDefault** is used as the fourth parameter instead, Windows will place and size the window automatically for you.

The Static Text Control

The program derives the **CHelloWindow** class from the **CFrameWnd** class (lines 11 through 17). In doing so it declares a private data member of type **CStatic***, as well as a constructor.

As seen in the previous section, the **CHelloWindow** constructor does two things. First it creates the application's window by calling the **Create** function (line 31), and then it allocates and creates the control that belongs inside the window. In this case a single static label is used as the only control. Object creation is always a two-step process in MFC. First, the memory for the instance of the class is allocated, thereby calling the constructor to

initialize any variables. Next, an explicit **Create** function is called to actually create the object on screen. The code allocates, constructs, and creates a single static text object using this two-step process at lines 36 through 40:

```
27 // The constructor for the window class
28 CHelloWindow::CHelloWindow()
29 {
30     // Create the window itself
31     Create(NULL,
32           "Hello World!",
33           WS_OVERLAPPEDWINDOW,
34           CRect(0,0,200,200));
35     // Create a static label
36     cs = new CStatic();
37     cs->Create("hello world",
38              WS_CHILD|WS_VISIBLE|SS_CENTER,
39              CRect(50,80,150,150),
40              this);
41 }
```

The constructor for the **CStatic** item is called when the memory for it is allocated, and then an explicit **Create** function is called to create the **CStatic** control's window. The parameters used in the **Create** function here are similar to those used for window creation at Line 31. The first parameter specifies the text to be displayed by the control. The second parameter specifies the style attributes. The style attributes are discussed in detail in the next tutorial but here we requested that the control be a child window (and therefore displayed within another window), that it should be visible, and that the text within the control should be centered. The third parameter determines the size and position of the static control. The fourth indicates the parent window for which this control is the child. Having created the static control, it will appear in the application's window and display the text specified.

Introduction to MFC Programming with Visual C++ Version 6.x

MFC Styles

Controls are the user interface objects used to create interfaces for Windows applications. Most Windows applications and dialog boxes that you see are nothing but a collection of controls arranged in a way that appropriately implements the functionality of the program. In order to build effective applications, you must completely understand how to use the controls available in Windows. There are only six basic controls-**CStatic**, **CButton**, **CEdit**, **CList**, **CComboBox**, and **CScrollBar** -along with some minor variations (also note that Windows 95 added a collection of about 15 enhanced controls as well). You need to understand what each control can do, how you can tune its appearance and behavior, and how to make the controls respond appropriately to user events. By combining this knowledge with an understanding of menus and dialogs you gain the ability to create any Windows application that you can imagine. You can create controls either programatically as shown in this tutorial, or through resource files using the dialog resource editor. While the dialog editor is much more convenient, it is extremely useful to have a general understanding of controls that you gain by working with them programatically as shown here and in the next tutorial.

The simplest of the controls, **CStatic**, displays static text. The **CStatic** class has no data members and only a few member functions: the constructor, the **Create** function for getting and setting icons on static controls, and several others. It does not respond to user events. Because of its simplicity, it is a good place to start learning about Windows controls.

In this tutorial we will look at the **CStatic** class to understand how controls can be modified and customized. In the following tutorial, we examine the **CButton** and **CScrollBar** classes to gain an understanding of event handling. Once you understand all of the controls and classes, you are ready to build complete applications.

The Basics

A **CStatic** class in MFC displays static text messages to the user. These messages can serve purely informational purposes (for example, text in a message dialog that describes an error), or they can serve as small labels that identify other controls. Pull open a File Open dialog in any Windows application and you will find six text labels. Five of the labels identify the lists, text area, and check box and do not ever change. The sixth displays the current directory and changes each time the current directory changes.

CStatic objects have several other display formats. By changing the *style* of a label it can display itself as a solid rectangle, as a border, or as an icon. The rectangular solid and frame forms of the **CStatic** class allow you to visually group related interface elements and to add separators between controls.

A **CStatic** control is always a child window to some parent window. Typically, the parent window is a main window for an application or a dialog box. You create the static control, as discussed in Tutorial 2, with two lines of code:

```
CStatic *cs;  
...  
cs = new CStatic();  
cs->Create("hello world",  
    WS_CHILD|WS_VISIBLE|SS_CENTER,  
    CRect(50,80, 150, 150),  
    this);
```

This two-line creation style is typical of all controls created using MFC. The call to **new** allocates memory for an instance of the **CStatic** class and, in the process, calls the constructor for the class. The constructor performs any initialization needed by the class. The **Create** function creates the control at the Windows level and puts it on the screen.

The **Create** function accepts up to five parameters, as described in the MFC help file. Choose the **Search** option in the **Help** menu of Visual C++ and then enter **Create** so that you can select **CStatic::Create** from the list. Alternatively, enter **CStatic** in the search dialog and then click the **Members** button on its overview page.

Most of these values are self-explanatory. The **lpszText** parameter specifies the text displayed by the label. The **rect** parameter controls the position, size, and shape of the text when it's displayed in its parent window. The upper left corner of the text is determined by the upper left corner of the **rect** parameter and its bounding rectangle is determined by the width and height of the rect parameter. The **pParentWnd** parameter indicates the parent of the **CStatic** control. The control will appear in the parent window, and the position of the control will be relative to the upper left corner of the client area of the parent. The **nID** parameter is an integer value used as a control ID by certain functions in the API. We'll see examples of this parameter in the next tutorial.

The **dwStyle** parameter is the most important parameter. It controls the appearance and behavior of the control. The following sections describe this parameter in detail.

CStatic Styles

All controls have a variety of display *styles*. Styles are determined at creation using the **dwStyle** parameter passed to the **Create** function. The style parameter is a bit mask that you build by or-ing together different mask constants. The constants available to a **CStatic** control can be found in the MFC help file (Find the page for the **CStatic::Create** function as described in the previous section, and click on the **Static Control Styles** link near the top of the page) and are also briefly described below:

Valid styles for the CStatic class -

Styles inherited from CWnd:

- Styles inherited from CWnd: WS_CHILD Mandatory for CStatic.
- Styles inherited from CWnd: WS_VISIBLE The control should be visible to the user.
- Styles inherited from CWnd: WS_DISABLED The control should reject user events.
- Styles inherited from CWnd: WS_BORDER The control's text is framed by a border.

Styles native to CStatic:

- SS_BLACKFRAME The control displays itself as a rectangular border. Color is the same as window frames.
- SS_BLACKRECT The control displays itself as a filled rectangle. Color is the same as window frames.
- SS_CENTER The text is center justified.
- SS_GRAYFRAME The control displays itself as a rectangular border. Color is the same as the desktop.
- SS_GRAYRECT The control displays itself as a filled rectangle. Color is the same as the desktop.
- SS_ICON The control displays itself as an icon. The text string is used as the name of the icon in a resource file. The rect parameter controls only positioning.
- SS_LEFT The text displayed is left justified. Extra text is word-wrapped.
- SS_LEFTNOWORDWRAP The text is left justified, but extra text is clipped.
- SS_NOPREFIX "&" characters in the text string indicate accelerator prefixes unless this attribute is used.
- SS_RIGHT The text displayed is right justified. Extra text is word-wrapped.
- SS_SIMPLE A single line of text is displayed left justified. Any CTLCOLOR messages must be ignored by the parent.
- SS_USERITEM User-defined item.
- SS_WHITEFRAME The control displays itself as a rectangular border. Color is the same as window backgrounds.
- SS_WHERECT The control displays itself as a filled rectangle. Color is the same as window backgrounds.

These constants come from two different sources. The "SS" (Static Style) constants apply only to **CStatic** controls. The "WS" (Window Style) constants apply to all windows and are therefore defined in the **CWnd** object from which **CStatic** inherits its behavior. There are many other "WS" style constants defined in **CWnd**. They can be found by looking up the **CWnd::Create** function in the MFC documentation. The four above are the only ones that apply to a **CStatic** object.

A **CStatic** object will always have at least two style constants or-ed together: WS_CHILD and WS_VISIBLE. The control is not created unless it is the child of another window, and it will be invisible unless WS_VISIBLE is specified. WS_DISABLED controls the label's response to events and, since a label has no sensitivity to events such as keystrokes or mouse clicks anyway, specifically disabling it is redundant.

All the other style attributes are optional and control the appearance of the label. By modifying the style attributes passed to the **CStatic::Create** function, you control how the static object appears on screen. You can learn quite a bit about the different styles by using style attributes to modify the text appearance of the **CStatic** object, as discussed in the next section.

CStatic Text Appearance

The code shown below is useful for understanding the behavior of the **CStatic** object. It is similar to the listing discussed in Tutorial 2, but it modifies the creation of the **CStatic** object slightly. Please turn to Tutorial 1 for instructions on entering and compiling this code.

```
//static1.cpp
#include <afxwin.h>

// Declare the application class
class CTestApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

// Create an instance of the application class
CTestApp TestApp;

// Declare the main window class
class CTestWindow : public CFrameWnd
{
    CStatic* cs;
public:
    CTestWindow();
};

// The InitInstance function is called
// once when the application first executes
BOOL CTestApp::InitInstance()
{
    m_pMainWnd = new CTestWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

// The constructor for the window class
CTestWindow::CTestWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
```

```

        "CStatic Tests",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));

// Get the size of the client rectangle
GetClientRect(&r);
r.InflateRect(-20,-20);

// Create a static label
cs = new CStatic();
cs->Create("hello world",
        WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
        r,
        this);
}

```

The code of interest in listing 3.1 is in the function for the window constructor, which is repeated below with line numbers:

```

CTestWindow::CTestWindow()
{
    CRect r;

    // Create the window itself
1    Create(NULL,
        "CStatic Tests",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));
    // Get the size of the client rectangle
2    GetClientRect(&r);
3    r.InflateRect(-20,-20);
    // Create a static label
4    cs = new CStatic();
5    cs->Create("hello world",
        WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
        r,
        this);
}

```

The function first calls the **CTestWindow::Create** function for the window at line 1. This is the **Create** function for the **CFrameWnd** object, since **CTestWindow** inherits its behavior from **CFrameWnd**. The code in line 1 specifies that the window should have a size of 200 by 200 pixels and that the upper left corner of the window should be initially placed at location 0,0 on the screen. The constant **rectDefault** can replace the **CRect** parameter if desired.

At line 2, the code calls **CTestWindow::GetClientRect**, passing it the parameter **&r**. The **GetClientRect** function is inherited from the **CWnd** class (see the side-bar for search strategies to use when trying to look up functions in the Microsoft documentation). The variable **r** is of type **CRect** and is declared as a local variable at the beginning of the function.

Two questions arise here in trying to understand this code: 1) What does the **GetClientRect** function do? and 2) What does a **CRect** variable do? Let's start with question 1. When you look up the **CWnd::GetClientRect** function in the MFC documentation you find it returns a structure of type **CRect** that contains the size of the client rectangle of the particular window. It stores the structure at the address passed in as a parameter, in this case **&r**. That address should point to a location of type **CRect**. The **CRect** type is a class defined in MFC. It is a convenience class used to manage rectangles. If you look up the class in the MFC documentation, you will find that it defines over 30 member functions and operators to manipulate rectangles.

In our case, we want to center the words "Hello World" in the window. Therefore, we use **GetClientRect** to get the rectangle coordinates for the client area. In line 3 we then call **CRect::InflateRect**, which symmetrically increases or decreases the size of a rectangle (see also **CRect::DeflateRect**). Here we have decreased the rectangle by 20 pixels on all sides. Had we not, the border surrounding the label would have blended into the window frame, and we would not be able to see it.

The actual **CStatic** label is created in lines 4 and 5. The style attributes specify that the words displayed by the label should be centered and surrounded by a border. The size and position of the border is determined by the **CRect** parameter **r**.

By modifying the different style attributes you can gain an understanding of the different capabilities of the **CStatic** Object. For example, the code below contains a replacement for the **CTestWindow** constructor function in the first listing.

```
CTestWindow::CTestWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
        "CStatic Tests",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));

    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);

    // Create a static label
    cs = new CStatic();
    cs->Create("Now is the time for all good men to \
come to the aid of their country",
        WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
        r,
        this);
}
```

The code above is identical to the previous except the text string is much longer. As you can see when you run the code, the **CStatic** object has wrapped the text within the specified bounding rectangle and centered each line individually.

If the bounding rectangle is too small to contain all the lines of text, then the text is clipped as needed to make it fit the available space. This feature of the **CStatic** object can be demonstrated by decreasing the size of the rectangle or increasing the length of the string.

In all the code we have seen so far, the style `SS_CENTER` has been used to center the text. The **CStatic** object also allows for left or right justification. Left justification is created by replacing the `SS_CENTER` attribute with an `SS_LEFT` attribute. Right justification aligns the words to the right margin rather than the left and is specified with the `SS_RIGHT` attribute.

One other text attribute is available. It turns off the word wrap feature and is used often for simple labels that identify other controls (see figure 3.1 for an example). The `SS_LEFTNOWORDWRAP` style forces left justification and causes no wrapping to take place.

Rectangular Display Modes for CStatic

The **CStatic** object also supports two different rectangular display modes: solid filled rectangles and frames. You normally use these two styles to visually group other controls within a window. For example, you might place a black rectangular frame in a window to collect together several related editable areas. You can choose from six different styles when creating these rectangles: `SS_BLACKFRAME`, `SS_BLACKRECT`, `SS_GRAYFRAME`, `SS_GRAYRECT`, `SS_WHITEFRAME`, and `SS_WHITERECT`. The `RECT` form is a filled rectangle, while the `FRAME` form is a border. The color names are a little misleading—for example, `SS_WHITERECT` displays a rectangle of the same color as the window background. Although this color defaults to white, the user can change it with the Control Panel and the rectangle may not be actually white on some machines.

When a rectangle or frame attribute is specified, the **CStatic**'s text string is ignored. Typically an empty string is passed. Try using several of these styles in the previous code and observe the result.

Fonts

You can change the font of a **CStatic** object by creating a **CFont** object. Doing so demonstrates how one MFC class can interact with another in certain cases to modify behavior of a control. The **CFont** class in MFC holds a single instance of a particular Windows font. For example, one instance of the **CFont** class might hold a Times font at 18 points while another might hold a Courier font at 10 points. You can modify the font used by a static label by calling the **SetFont** function that **CStatic** inherits from **CWnd**. The code below shows the code required to implement fonts.

```
CTestWindow::CTestWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
        "CStatic Tests",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));
    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);
    // Create a static label
    cs = new CStatic();
```

```

cs->Create("Hello World",
    WS_CHILD|WS_VISIBLE|WS_BORDER|SS_CENTER,
    r,
    this);

// Create a new 36 point Arial font
font = new CFont;
font->CreateFont(36,0,0,0,700,0,0,0,
    ANSI_CHARSET,OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS,
    DEFAULT_QUALITY,
    DEFAULT_PITCH|FF_DONTCARE,
    "arial");
// Cause the label to use the new font
cs->SetFont(font);
}

```

The code above starts by creating the window and the **CStatic** object as usual. The code then creates an object of type **CFont**. The font variable should be declared as a data member in the **CTestWindow** class with the line "CFont *font". The **CFont::CreateFont** function has 15 parameters (see the MFC help file), but only three matter in most cases. For example, the 36 specifies the size of the font in points, the 700 specifies the density of the font (400 is "normal," 700 is "bold," and values can range from 1 to 1000. The constants FW_NORMAL and FW_BOLD have the same meanings. See the FW constants in the API help file), and the word "arial" names the font to use. Windows typically ships with five True Type fonts (Arial, Courier New, Symbol, Times New Roman, and Wingdings), and by sticking to one of these you can be fairly certain that the font will exist on just about any machine. If you specify a font name that is unknown to the system, then the **CFont** class will choose the default font seen in all the other examples used in this tutorial.

For more information on the **CFont** class see the MFC documentation. There is also a good overview on fonts in the API on-line help file. Search for "Fonts and Text Overview."

The **SetFont** function comes from the **CWnd** class. It sets the font of a window, in this case the **CStatic** child window. One question you may have at this point is, "How do I know which functions available in **CWnd** apply to the **CStatic** class?" You learn this by experience. Take half an hour one day and read through all the functions in **CWnd**. You will learn quite a bit and you should find many functions that allow you to customize controls. We will see other **Set** functions found in the **CWnd** class in the next tutorial.

Introduction to MFC Programming with Visual C++ Version 6.x

Message Maps

Any user interface object that an application places in a window has two controllable features: 1) its appearance, and 2) its behavior when responding to events. In the last tutorial you gained an understanding of the **CStatic** control and saw how you can use style attributes to customize the appearance of user interface objects. These concepts apply to all the different control classes available in MFC.

In this tutorial we will examine the **CButton** control to gain an understanding of message maps and simple event handling. We'll then look at the **CScrollBar** control to see a somewhat more involved example.

Understanding Message Maps

As discussed in Tutorial 2, MFC programs do not contain a main function or event loop. All of the event handling happens "behind the scenes" in C++ code that is part of the **CWinApp** class. Because it is hidden, we need a way to tell the invisible event loop to notify us about events of interest to the application. This is done with a mechanism called a *message map*. The message map identifies interesting events and then indicates functions to call in response to those events.

For example, say you want to write a program that will quit whenever the user presses a button labeled "Quit." In the program you place code to specify the button's creation: you indicate where the button goes, what it says, etc. Next, you create a message map for the parent of the button-whenver a user clicks the button, it tries to send a message to its parent. By installing a message map for the parent window you create a mechanism to intercept and use the button's messages. The message map will request that MFC call a specific function whenever a specific button event occurs. In this case, a click on the quit button is the event of interest. You then put the code for quitting the application in the indicated function.

MFC does the rest. When the program executes and the user clicks the Quit button, the button will highlight itself as expected. MFC then automatically calls the right function and the program terminates. With just a few lines of code your program becomes sensitive to user events.

The CButton Class

The **CStatic** control discussed in Tutorial 3 is unique in that it cannot respond to user events. No amount of clicking, typing, or dragging will do anything to a **CStatic** control because it ignores the user completely. However, The **CStatic** class is an anomaly. All of the other controls available in Windows respond to user events in two ways. First, they update their appearance automatically when the user manipulates them (e.g., when the user clicks on a button it highlights itself to give the user visual feedback). Second, each different control tries to send messages to your code so the program can respond to the user as needed. For example, a button sends a *command message* whenever it gets clicked. If you write code to receive the messages, then your code can respond to user events.

To gain an understanding of this process, we will start with the **CButton** control. The code below demonstrates the creation of a button.

```
// button1.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
// Declare the application class
class CButtonApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
    CButton *button;
public:
    CButtonWindow();
```

```

};
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
    m_pMainWnd = new CButtonWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
// The constructor for the window class
CButtonWindow::CButtonWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
           "CButton Tests",
           WS_OVERLAPPEDWINDOW,
           CRect(0,0,200,200));

    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);

    // Create a button
    button = new CButton();
    button->Create("Push me",
                 WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
                 r,
                 this,
                 IDB_BUTTON);
}

```

The code above is nearly identical to the code discussed in previous tutorials. The **Create** function for the **CButton** class, as seen in the MFC help file, accepts five parameters. The first four are exactly the same as those found in the **CStatic** class. The fifth parameter indicates the resource ID for the button. The resource ID is a unique integer value used to identify the button in the message map. A constant value `IDB_BUTTON` has been defined at the top of the program for this value. The "IDB_" is arbitrary, but here indicates that the constant is an ID value for a Button. It is given a value of 100 because values less than 100 are reserved for system-defined IDs. You can use any value above 99.

The style attributes available for the **CButton** class are different from those for the **CStatic** class. Eleven different "BS" ("Button Style") constants are defined. A complete list of "BS" constants can be found using **Search** on CButton and selecting the "button style" link. Here we have used the `BS_PUSHBUTTON` style for the button, indicating that we want this button to display itself as a normal push-button. We have also used two familiar "WS" attributes: `WS_CHILD` and `WS_VISIBLE`. We will examine some of the other styles in later sections.

When you run the code, you will notice that the button responds to user events. That is, it highlights as you would expect. It does nothing else because we haven't told it what to do. We need to wire in a message map to make the button do something interesting.

Creating a Message Map

The code below contains a message map as well as a new function that handles the button click (so the program beeps when the user clicks on the button). It is simply an extension of the prior code.

```
// button2.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
// Declare the application class
class CButtonApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
    CButton *button;
public:
    CButtonWindow();
    afx_msg void HandleButton();
    DECLARE_MESSAGE_MAP()
};
// The message handler function
void CButtonWindow::HandleButton()
{
    MessageBeep(-1);
}
// The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
    ON_BN_CLICKED(IDB_BUTTON, HandleButton)
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
    m_pMainWnd = new CButtonWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
```

```

// The constructor for the window class
CButtonWindow::CButtonWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
           "CButton Tests",
           WS_OVERLAPPEDWINDOW,
           CRect(0,0,200,200));
    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);
    // Create a button
    button = new CButton();
    button->Create("Push me",
                  WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
                  r,
                  this,
                  IDB_BUTTON);
}

```

Three modifications have been made to the code:

1. The class declaration for **CButtonWindow** now contains a new member function as well as a macro that indicates a message map is defined for the class. The **HandleButton** function, which is identified as a message handler by the use of the **afx_msg** tag, is a normal C++ function. There are some special constraints on this function which we will discuss shortly (e.g., it must be **void** and it cannot accept any parameters). The **DECLARE_MESSAGE_MAP** macro makes the creation of a message map possible. *Both the function and the macro must be public.*
2. The **HandleButton** function is created in the same way as any member function. In this function, we called the **MessageBeep** function available from the Windows API.
3. Special MFC macros create a message map. In the code, you can see that the **BEGIN_MESSAGE_MAP** macro accepts two parameters. The first is the name of the specific class to which the message map applies. The second is the base class from which the specific class is derived. It is followed by an **ON_BN_CLICKED** macro that accepts two parameters: The ID of the control and the function to call whenever that ID sends a command message. Finally, the message map ends with the **END_MESSAGE_MAP** macro.

When a user clicks the button, it sends a command message containing its ID to its parent, which is the window containing the button. That is default behavior for a button, and that is why this code works. The button sends the message to its parent because it is a child window. The parent window intercepts this message and uses the message map to determine the function to call. MFC handles the routing, and whenever the specified message is seen, the indicated function gets called. The program beeps whenever the user clicks the button.

The **ON_BN_CLICKED** message is the only interesting message sent by an instance of the **CButton** class. It is equivalent to the **ON_COMMAND** message in the **CWnd** class, and is simply a convenient synonym for it.

Sizing Messages

In the code above, the application's window, which is derived from the **CFrameWnd** class, recognized the button-click message generated by the button and responded to it because of its message map. The **ON_BN_CLICKED** macro added into the message map (search for the **CButton** overview as well as the **ON_COMMAND** macro in the MFC help file) specifies the ID of the button and the function that the window should call when it receives a command message from that button. Since the button automatically sends to its parent its ID in a command message whenever the user clicks it, this arrangement allows the code to handle button events properly.

The frame window that acts as the main window for this application is also capable of sending messages itself. There are about 100 different messages available, all inherited from the **CWnd** class. By browsing through the member functions for the **CWnd** class in MFC help file you can see what all of these messages are. Look for any member function beginning with the word "On".

You may have noticed that all of the code demonstrated so far does not handle re-sizing very well. When the window re-sizes, the frame of the window adjusts accordingly but the contents stay where they were placed originally. It is possible to make resized windows respond more attractively by recognizing resizing events. One of the messages that is sent by any window is a sizing message. The message is generated whenever the window changes shape. We can use this message to control the size of child windows inside the frame, as shown below:

```
// button3.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
// Declare the application class
class CButtonApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
    CButton *button;
public:
    CButtonWindow();
    afx_msg void HandleButton();
    afx_msg void OnSize(UINT, int, int);
    DECLARE_MESSAGE_MAP()
};
// A message handler function
void CButtonWindow::HandleButton()
{
    MessageBeep(-1);
}
// A message handler function
void CButtonWindow::OnSize(UINT nType, int cx,
    int cy)
```

```

{
    CRect r;
    GetClientRect(&r);
    r.InflateRect(-20,-20);
    button->MoveWindow(r);
}

// The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
    ON_BN_CLICKED(IDB_BUTTON, HandleButton)
    ON_WM_SIZE()
END_MESSAGE_MAP()

// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
    m_pMainWnd = new CButtonWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

// The constructor for the window class
CButtonWindow::CButtonWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
        "CButton Tests",
        WS_OVERLAPPEDWINDOW,
        CRect(0,0,200,200));

    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);

    // Create a button
    button = new CButton();
    button->Create("Push me",
        WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
        r,
        this,
        IDB_BUTTON);
}

```

To understand this code, start by looking in the message map for the window. There you will find the entry `ON_WM_SIZE`. This entry indicates that the message map is sensitive to sizing messages coming from the **CButtonWindow** object. Sizing messages are generated on this window whenever the user re-sizes it. The messages come to the window itself (rather than being sent to a parent as the `ON_COMMAND` message is by the button) because the frame window is not a child.

Notice also that the `ON_WM_SIZE` entry in the message map has no parameters. As you can see in the MFC documentation under the **CWnd** class, *it is understood that the `ON_WM_SIZE` entry in the message map will always call a function named **OnSize**, and that function must accept the three parameters shown*. The **OnSize** function must be a member function of the class owning the message map, and the function must be declared in the class as an **afx_msg** function (as shown in the definition of the **CButtonWindow** class).

If you look in the MFC documentation there are almost 100 functions named "On..." in the **CWnd** class. **CWnd::OnSize** is one of them. All these functions have a corresponding tag in the message map with the form `ON_WM_`. For example, `ON_WM_SIZE` corresponds to **OnSize**. None of the `ON_WM_` entries in the message map accept parameters like `ON_BN_CLICKED` does. The parameters are assumed and automatically passed to the corresponding "On..." function like **OnSize**.

To repeat, because it is important: The **OnSize** function always corresponds to the `ON_WM_SIZE` entry in the message map. You must name the handler function **OnSize**, and it must accept the three parameters shown in the listing. You can find the specific parameter requirements of any **On...** function by looking up that function in the MFC help file. You can look the function up directly by typing **OnSize** into the search window, or you can find it as a member function of the **CWnd** class.

Inside the **OnSize** function itself in the code above, three lines of code modify the size of the button held in the window. You can place any code you like in this function.

The call to **GetClientRect** retrieves the new size of the window's client rectangle. This rectangle is then deflated, and the **MoveWindow** function is called on the button. **MoveWindow** is inherited from **CWnd** and re-sizes and moves the child window for the button in one step.

When you execute the program above and re-size the application's window, you will find the button re-sizes itself correctly. In the code, the re-size event generates a call through the message map to the **OnSize** function, which calls the **MoveWindow** function to re-size the button appropriately.

Window Messages

By looking in the MFC documentation, you can see the wide variety of **CWnd** messages that the main window handles. Some are similar to the sizing message seen in the previous section. For example, `ON_WM_MOVE` messages are sent when a user moves a window, and `ON_WM_PAINT` messages are sent when any part of the window has to be repainted. In all of our programs so far, repainting has happened automatically because controls are responsible for their own appearance. If you draw the contents of the client area yourself with GDI commands, the application is responsible for repainting any drawings it places directly in the window. In this context the `ON_WM_PAINT` message becomes important.

There are also some event messages sent to the window that are more esoteric. For example, you can use the `ON_WM_TIMER` message in conjunction with the **SetTimer** function to cause the window to receive messages at pre-set intervals. The code below demonstrates the process. When you run this code, the program will beep once each second. The beeping can be replaced by a number of useful processes.

```
// button4.cpp
#include <afxwin.h>
#define IDB_BUTTON 100
#define IDT_TIMER1 200
// Declare the application class
```

```

class CButtonApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
// Create an instance of the application class
CButtonApp ButtonApp;
// Declare the main window class
class CButtonWindow : public CFrameWnd
{
    CButton *button;
public:
    CButtonWindow();
    afx_msg void HandleButton();
    afx_msg void OnSize(UINT, int, int);
    afx_msg void OnTimer(UINT);
    DECLARE_MESSAGE_MAP()
};
// A message handler function
void CButtonWindow::HandleButton()
{
    MessageBeep(-1);
}
// A message handler function
void CButtonWindow::OnSize(UINT nType, int cx,
    int cy)
{
    CRect r;
    GetClientRect(&r);
    r.InflateRect(-20,-20);
    button->MoveWindow(r);
}
// A message handler function
void CButtonWindow::OnTimer(UINT id)
{
    MessageBeep(-1);
}
// The message map
BEGIN_MESSAGE_MAP(CButtonWindow, CFrameWnd)
    ON_BN_CLICKED(IDB_BUTTON, HandleButton)
    ON_WM_SIZE()
    ON_WM_TIMER()
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CButtonApp::InitInstance()
{
    m_pMainWnd = new CButtonWindow();

```

```

        m_pMainWnd->ShowWindow(m_nCmdShow);
        m_pMainWnd->UpdateWindow();
        return TRUE;
    }
// The constructor for the window class
CButtonWindow::CButtonWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
            "CButton Tests",
            WS_OVERLAPPEDWINDOW,
            CRect(0,0,200,200));

    // Set up the timer
    SetTimer(IDT_TIMER1, 1000, NULL); // 1000 ms.
    // Get the size of the client rectangle
    GetClientRect(&r);
    r.InflateRect(-20,-20);
    // Create a button
    button = new CButton();
    button->Create("Push me",
                  WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
                  r,
                  this,
                  IDB_BUTTON);
}

```

Inside the program above we created a button, as shown previously, and left its re-sizing code in place. In the constructor for the window we also added a call to the **SetTimer** function. This function accepts three parameters: an ID for the timer (so that multiple timers can be active simultaneously, the ID is sent to the function called each time a timer goes off), the time in milliseconds that is to be the timer's increment, and a function. Here, we passed NULL for the function so that the window's message map will route the function automatically. In the message map we have wired in the ON_WM_TIMER message, and it will automatically call the **OnTimer** function passing it the ID of the timer that went off.

When the program runs, it beeps once each 1,000 milliseconds. Each time the timer's increment elapses, the window sends a message to itself. The message map routes the message to the **OnTimer** function, which beeps. You can place a wide variety of useful code into this function.

Scroll Bar Controls

Windows has two different ways to handle scroll bars. Some controls, such as the edit control and the list control, can be created with scroll bars attached. When this is the case, the master control handles the scroll bars automatically. For example, if an edit control has its scroll bars active then, when the scroll bars are used, the edit control scrolls as expected without any additional code.

Scroll bars can also work on a stand-alone basis. When used this way they are seen as independent controls in their own right. You can learn more about scroll bars by referring to the **CScrollBar** section of the MFC reference

manual. Scroll bar controls are created the same way we created static labels and buttons. They have four member functions that allow you to get and set both the range and position of a scroll bar.

The code shown below demonstrates the creation of a horizontal scroll bar and its message map.

```
// sb1.cpp
#include <afxwin.h>
#define IDM_SCROLLBAR 100
const int MAX_RANGE=100;
const int MIN_RANGE=0;
// Declare the application class
class CScrollBarApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
// Create an instance of the application class
CScrollBarApp ScrollBarApp;
// Declare the main window class
class CScrollBarWindow : public CFrameWnd
{
    CScrollBar *sb;
public:
    CScrollBarWindow();
    afx_msg void OnHScroll(UINT nSBCode, UINT nPos,
        CScrollBar* pScrollBar);
    DECLARE_MESSAGE_MAP()
};
// The message handler function
void CScrollBarWindow::OnHScroll(UINT nSBCode,
    UINT nPos, CScrollBar* pScrollBar)
{
    MessageBeep(-1);
}
// The message map
BEGIN_MESSAGE_MAP(CScrollBarWindow, CFrameWnd)
    ON_WM_HSCROLL()
END_MESSAGE_MAP()
// The InitInstance function is called once
// when the application first executes
BOOL CScrollBarApp::InitInstance()
{
    m_pMainWnd = new CScrollBarWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}
// The constructor for the window class
```

```

CScrollBarWindow::CScrollBarWindow()
{
    CRect r;
    // Create the window itself
    Create(NULL,
           "CScrollBar Tests",
           WS_OVERLAPPEDWINDOW,
           CRect(0,0,200,200));

    // Get the size of the client rectangle
    GetClientRect(&r);
    // Create a scroll bar
    sb = new CScrollBar();
    sb->Create(WS_CHILD|WS_VISIBLE|SBS_HORZ,
              CRect(10,10,r.Width()-10,30),
              this,
              IDM_SCROLLBAR);
    sb->SetScrollRange(MIN_RANGE,MAX_RANGE,TRUE);
}

```

Windows distinguishes between horizontal and vertical scroll bars and also supports an object called a *size box* in the **CScrollBar** class. A size box is a small square. It is formed at the intersection of a horizontal and vertical scroll bar and can be dragged by the mouse to automatically re-size a window. Looking at the code in listing 4.5, you can see that the **Create** function creates a horizontal scroll bar using the SBS_HORZ style. Immediately following creation, the range of the scroll bar is set for 0 to 100 using the two constants MIN_RANGE and MAX_RANGE (defined at the top of the listing) in the **SetScrollRange** function.

The event-handling function **OnHScroll** comes from the **CWnd** class. We have used this function because the code creates a horizontal scroll bar. For a vertical scroll bar you should use **OnVScroll**. In the code here the message map wires in the scrolling function and causes the scroll bar to beep whenever the user manipulates it. When you run the code you can click on the arrows, drag the thumb, and so on. Each event will generate a beep, but the thumb will not actually move because we have not wired in the code for movement yet.

Each time the scroll bar is used and **OnHScroll** is called, your code needs a way to determine the user's action. Inside the **OnHScroll** function you can examine the first parameter passed to the message handler, as shown below. If you use this code with the code above, the scroll bar's thumb will move appropriately with each user manipulation.

```

// The message handling function
void CScrollBarWindow::OnHScroll(UINT nSBCode,
                                UINT nPos, CScrollBar* pScrollBar)
{
    int pos;
    pos = sb->GetScrollPos();
    switch ( nSBCode )
    {
        case SB_LINEUP:
            pos -= 1;
            break;

```

```

        case SB_LINEDOWN:
            pos += 1;
            break;
        case SB_PAGEUP:
            pos -= 10;
            break;
        case SB_PAGEDOWN:
            pos += 10;
            break;
        case SB_TOP:
            pos = MIN_RANGE;
            break;
        case SB_BOTTOM:
            pos = MAX_RANGE;
            break;

        case SB_THUMBPOSITION:
            pos = nPos;
            break;
        default:
            return;
    }
    if ( pos < MIN_RANGE )
        pos = MIN_RANGE;
    else if ( pos > MAX_RANGE )
        pos = MAX_RANGE;
    sb->SetScrollPos( pos, TRUE );
}

```

The different constant values such as `SB_LINEUP` and `SB_LINEDOWN` are described in the **CWnd::OnHScroll** function documentation. The code above starts by retrieving the current scroll bar position using **GetScrollPos**. It then decides what the user did to the scroll bar using a switch statement. The constant value names imply a vertical orientation but are used in horizontal scroll bars as well: `SB_LINEUP` and `SB_LINEDOWN` apply when the user clicks the left and right arrows. `SB_PAGEUP` and `SB_PAGEDOWN` apply when the user clicks in the shaft of the scroll bar itself. `SB_TOP` and `SB_BOTTOM` apply when the user moves the thumb to the top or bottom of the bar. `SB_THUMBPOSITION` applies when the user drags the thumb to a specific position. The code adjusts the position accordingly, then makes sure that it's still in range before setting the scroll bar to its new position. Once the scroll bar is set, the thumb moves on the screen to inform the user visually.

A vertical scroll bar is handled the same way as a horizontal scroll bar except that you use the `SBS_VERT` style and the **OnVScroll** function. You can also use several alignment styles to align both the scroll bars and the grow box in a given client rectangle.

Understanding Message Maps

The message map structure is unique to MFC. It is important that you understand why it exists and how it actually works so that you can exploit this structure in your own code.

Any C++ purist who looks at a message map has an immediate question: Why didn't Microsoft use virtual functions instead? Virtual functions are the standard C++ way to handle what message maps are doing in MFC, so the use of rather bizarre macros like `DECLARE_MESSAGE_MAP` and `BEGIN_MESSAGE_MAP` seems like a hack.

MFC uses message maps to get around a fundamental problem with virtual functions. Look at the **CWnd** class in the MFC help file. It contains over 200 member functions, all of which would have to be virtual if message maps were not used. Now look at all of the classes that subclass the **CWnd** class. For example, go to the contents page of the MFC help file and look at the visual object hierarchy. 30 or so classes in MFC use **CWnd** as their base class. This set includes all of the visual controls such as buttons, static labels, and lists. Now imagine that MFC used virtual functions, and you created an application that contained 20 controls. Each of the 200 virtual functions in **CWnd** would require its own virtual function table, and each instance of a control would therefore have a set of 200 virtual function tables associated with it. The program would have roughly 4,000 virtual function tables floating around in memory, and this is a problem on machines that have memory limitations. Because the vast majority of those tables are never used, they are unneeded.

Message maps duplicate the action of a virtual function table, but do so on an on-demand basis. When you create an entry in a message map, you are saying to the system, "when you see the specified message, please call the specified function." Only those functions that actually get overridden appear in the message map, saving memory and CPU overhead.

When you declare a message map with `DECLARE_MESSAGE_MAP` and `BEGIN_MESSAGE_MAP`, the system routes all messages through to your message map. If your map handles a given message, then your function gets called and the message stops there. However, if your message map does not contain an entry for a message, then the system sends that message to the class specified in the second parameter of `BEGIN_MESSAGE_MAP`. That class may or may not handle it and the process repeats. Eventually, if no message map handles a given message, the message arrives at a default handler that eats it.

For more information on the ClassWizard, AppWizard and the resource editors see the tutorials on these topics on the [MFC Tutorials page](#).

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

Understanding the Document/View Paradigm

The framework that the AppWizard generates revolves around a concept called the *Document/View Paradigm*. If you understand this paradigm then it is much easier to understand the files that the AppWizard generates, and it also makes it much easier for you to fit your code into the AppWizard framework. This tutorial describes the paradigm so that you completely understand its purpose and intent.

The App Wizard takes a document-centric approach to application design. The MFC class hierarchy contains two classes that help to support this approach: **CDocument** and **CView**. The AppWizard and MFC use this approach because most Windows applications work this way. Built into any framework generated by the AppWizard is the assumption that your application will want to load and work with multiple documents, and that each document will have one or more views open at a time. This approach makes it extremely easy to create both Single Document Interface(SDI) and Multiple Document Interface(MDI) applications. All applications can be thought of in terms of documents and views.

It is easiest to understand the document/view paradigm if you think about a typical MDI word processor like Microsoft Word. At any given time you can have one or more documents open. A document represents a single open file. The user generally has one view open on each document. The view shows the user a part of the document in an MDI window, and lets the user edit the document. However, Microsoft Word allows the user to split a window into multiple frames so that the user can have two or more views on the same document if desired. When the user edits in one of the views, it changes the data in the document associated with the view. If a document has multiple views open and the user changes data in one of the views, the document and all other related views should reflect the change. When the user saves the document, it is that data held by the document that gets saved to disk.

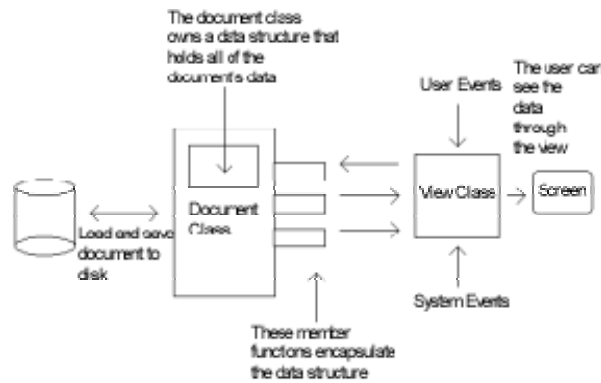
Many applications allow the user to open just one type of document. Microsoft Word, for example, works only with Microsoft Word documents. It may allow you to open other types of documents, but it first filters them to turn them into Word documents. Other applications open several different types of documents and can display all of them simultaneously in its MDI framework. Visual C++ is an example of this type of application. The most common type of document Visual C++ works with is a text file that contains code. However, you can open resources in the different resource editors as different types of documents in the MDI framework. Microsoft Works is similar. It can open word processing documents, but it can also open spreadsheet and database documents. Each of these documents has a completely unique view in the MDI frame, but all of the different views live there in harmony with one another. In addition, database documents in Works can be viewed both in a spreadsheet-like list, or in a customizable form that shows one complete record at a time.

Therefore, in the most general case, an application may be able to open several different types of documents simultaneously. Each type of document can have a unique viewing window. Any document may have multiple views open at once. In fact, a document might have more than one way of viewing its data, and those different views on a single document might be open simultaneously. Each document stores its data on disk. The views give the user a way to view and edit that data.

At a code level, the document and view classes separate functionality. Since this arrangement is typical of most applications, the framework generated by the AppWizard supports this structure implicitly. The document class is responsible for data. It reads the data from a file on disk and holds it in memory. The view class is responsible for presentation. The view takes data from the document class and presents it to the user in a view. The multiple views for a single document synchronize themselves through the data in the document. The MFC class hierarchy contains classes - **CDocument** and **CView** - that make this structure easy to create. The AppWizard derives new classes from the existing document and view classes and you build your application within those derived classes.

The goal of the document class is to completely encapsulate the data for one open document. It holds the data for the document in a data structure in memory and knows how to load the data from the disk and save it to the disk. The view class uses and manipulates the data in the document based on user events. The view class is responsible for letting the user view the contents of the document. (Note - The document class does not necessarily have to hold its data in memory. In certain types of applications it can act instead as a pipe to a binary file that remains on disk, or to a database.)

The relationship between the document and view classes is summarized in the figure below. When you are designing your own applications, you want the document class to completely encapsulate the data, and you want the view class to display information to the user. There should be a clear and obvious way for the view to interact with the document through member functions that you create if you want to properly encapsulate the data in the document.



When you create your own application, you typically will create some sort of data structure in the document class to hold the document's data. MFC contains a number of collection classes that you can use for this purpose, or you can create any other data structure that you like. The document class will be called when the data in the data structure needs to be loaded from disk or saved to disk. This is most commonly done through the document class's `Serialize` function. You will then add your own member functions to the class to encapsulate the data structure and allow the view class to manipulate the data held by the data structure. The view class contains the code to handle user events and draw to the screen.

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

Introduction to the AppWizard Files

The framework that the AppWizard generates typically consists of at least 20 different files. The first time that you see those files and begin to wade through them they can be extremely confusing unless you have an appropriate roadmap. The purpose of this tutorial is to provide you with that roadmap.

Let's start by creating a simple framework with the AppWizard. To create this framework we will use all of the default settings for the AppWizard's options. Once the AppWizard has generated the files for this default framework we can go through each one to see what they all do.

Select the **New** option in the **File** menu. In the dialog that appears, make sure that the **Project** tab is selected. We want to create an AppWizard workspace, so select "MFC AppWizard(exe)" from the choices. Along the right side of the window is a dialog that lets you name the project, choose the project type, and pick the project's directory. Choose the appropriate directory in which to create the new samp directory by leaving it blank (a new directory will be created in your current directory) or type the path directly. Name the project "samp". This name will be used as the new directory name as well, and that is fine - you will see the word "samp" echoed in the Location field as you type it. Click the **OK** button to create the new project.

You will next see a group of six colorful option dialogs. You can move between them with the **Next** and **Previous** buttons. Look through them now. We do not want to change any of the default options, so when you are through looking at them click the **Finish** button. You will see a final dialog that summarizes your choices. Click **OK** on this final dialog and the AppWizard will create the new framework, after displaying what options have been selected so that you can make sure it is what you really want.

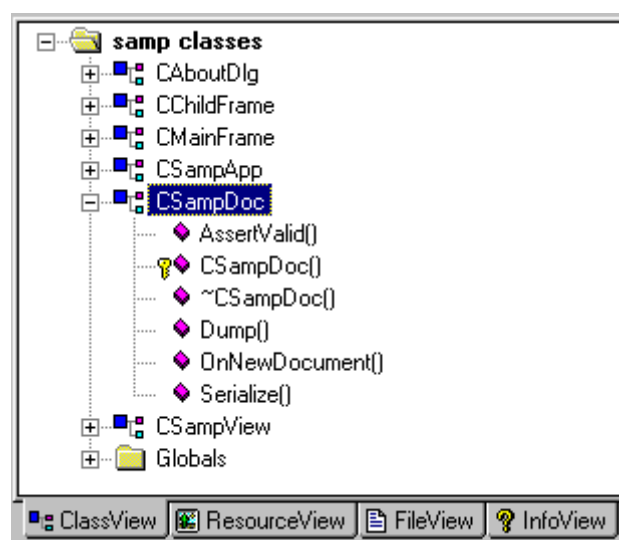
Use the File Manager to look into the new directory that the AppWizard created. It will contain about 20 files. Here's a quick summary of all of these different files:

- SAMP.DSP, SAMP.DSW, SAMP.NCB, SAMP.OPT - The project and workspace files for the application.

- SAMP.CLW - A data file used by the ClassWizard that you will never touch. If you accidentally delete it the ClassWizard will regenerate it for you.
- README.TXT - A text file that briefly describes most of the files in the directory.
- STDAFX.H, STDAFX.CPP - Short little files that help in the creation of *precompiled header* files. To speed compilation, Visual C++ can parse all of the header files (which are gigantic) and then save an image of the compiler's symbol table to disk. It is much quicker for the compiler to subsequently load the precompiled header file when compiling a CPP file. You will never touch these files.
- SAMP.H, SAMP.CPP - If you look inside these files you will find that they contain a class derived from **CWinApp**, as must all MFC programs. There is also an overridden **InitInstance** function. You will sometimes have occasion to modify these files, and will use the message map in the application class to handle messages that apply application-wide.
- MAINFRM.H, MAINFRM.CPP - These files contain a class derived from **CMDIFrameWnd**. In an SDI application the class is derived from **CFrameWnd** (you control whether the AppWizard generates an SDI or MDI application in the first of the AppWizard's six configuration screens). You will typically leave these two files alone.
- CHILDFRM.H, CHILDFRM.CPP - These files contain a class derived from **CMDIChildWnd** and control the look of the child windows in the MDI shell. In an SDI application these files are omitted. You will typically leave these two files alone.
- SAMPDOC.H, SAMPDOC.CPP - These two files derive a new class from **CDocument**. You will modify these files to create the document class for your application.
- SAMPVIEW.H, SAMPVIEW.CPP - These two files derive a new class from **CView**. You will modify these files to create the view class for your application.
- RESOURCE.H, SAMP.RC, SAMP.APS - These three files compose the *resource file* for your application. You will find in a moment that the file contains an accelerator table, an About dialog, two icons, two menus, a fairly extensive string table, a toolbar, and version information.

Typically you will modify only the last seven files when creating a new application. You will use the ClassWizard to help with the modifications to the document and view classes, and you will use the resource editors to modify the resource files.

A quick note on the workspace view in Visual C++ 6.x. The workspace view typically appears on the left side of the Visual C++ windows and looks like this (if you cannot see it, choose the **Workspace** option in the **View** menu):



The project window has 4 tabs along the bottom. In the figure the **ClassView** tab is selected. This view shows you all of the classes in the application. If you click on the small plus signs, you can see the functions in each class. You can double click on a class or function name and VC++ will take you to that point in the code. The **ResourceView** tab shows you the resources in the application. Open all of the resources and look at them by double clicking. You will find that this application has an accelerator table, an about dialog, two icons, two menus, a string table, a toolbar and some version information. The **FileView** tab shows you the files that make up the application (if you are upgrading from VC++ 2.x this is the view you are familiar with). You can delete files from the project here by selecting a file and hitting the delete key. The **InfoView** tab shows you the help files.

Build and execute this framework so that you can see its default behavior. To do this, choose the **Build samp.exe** option in the **Build** menu, and then choose the **Execute samp.exe** option in the **Build** menu. You will find that the application has the expected menu bar, and that several of the menu options fully or partially work. For example, the **Help** option brings up an about dialog and the **Open** option brings up an open dialog. The application has a toolbar, a status bar, an about box, etc. In other words, it is a fairly complete application framework.

To get an idea of how you might modify this framework using the ClassWizard, try the following example. Select the **ClassWizard** option in the **View** menu. Make sure that the **Message Maps** tab is selected. Make sure that the **CSampView** class is selected in the **Class Name** combo box. In the **Object Ids** list choose the first item: **CSampView**. In the **Messages** list choose WM_MOUSEMOVE. Click the **Add Function** button. Make sure that the **OnMouseMove** function is selected in the **Member Functions** list and click the **Edit Code** button. The ClassWizard will create the new **OnMouseMove** function, add it appropriately to the message map, and then deposit you at that point in the view class. Modify the function so that it looks like this:

```
void CSampView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (nFlags == MK_LBUTTON)
    {
        CClientDC dc(this);
        dc.Rectangle(point.x, point.y,
            point.x+5, point.y+5);
    }
    CView::OnMouseMove(nFlags, point);
}
```

Build and execute the application. When you drag the mouse in the window you will find that it draws small 5 by 5 rectangles wherever the mouse goes. The **CClientDC** class creates a "device context" that allows you to draw in a window. We use the dc to draw each rectangle. **CClientDC** derives its behavior from the **CDC** class, which you will find, when you look it up in the MFC help file (click on the word and then press F1), contains about 100 different drawing functions. Experiment if you like with these functions. They are all fairly self explanatory. While you are in the help file it wouldn't hurt to take a look at the **CDocument** and **CView** classes to start getting a feel for them.

To get an idea of how the resource editors work, open SAMP.RC by clicking on the ResourceView tab in the project window. Open the **Dialog** folder and then double click on the IDD_ABOUTBOX dialog and the template for the About box will appear. Click on one of the static text strings to bring it into focus. Hit **Enter** and a dialog appears which you can use to modify the string. Under the **General** tab, change the text in the "Caption:" field. Build and execute to see the changes to the About box.

In the next tutorial we will make a number of changes to the document class, view class and resource file to create a simple but complete drawing editor.

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

Creating a Simple Drawing Editor

Probably the best way to understand the document and view classes, as well as the ClassWizard and resource editors, is to run through a simple example application that makes use of all of these features in straightforward ways. In this tutorial we will use the AppWizard framework that you generated in the previous tutorial. As you recall, we modified it so that you could draw small rectangles using the mouse. However, it has no way to load or save the "drawings", nor can it refresh the screen on exposure. In this tutorial we will solve these problems using the Document/View paradigm. We will also add a new menu option and dialog to the application to demonstrate simple uses of the resource editors.

Start with the "samp" application framework that you created in tutorial 3. In order to turn this framework into a complete application we must take the following steps:

- Add a data structure to the document class to hold the points that the user draws.
- Add code to the document class so that the data structure in the document class is properly loaded from and saved to disk.
- Add code to the view class so that whenever the user draws a point it is added to the document's data structure.
- Add code to the view class so that it properly redraws the drawing when the application receives an exposure event.

Altogether, we will have to write only about 15 lines of code to accomplish all of this.

The following instructions walk you through these different steps.

Step 1: Add a data structure to the document class

MFC contains a variety of different collection classes, and by using them we can very easily add a robust data structure to hold the points that the user draws. Look up the collection classes in the MFC help file by choosing the **Search** option in the **Help** menu, and then typing "hierarchy chart". There should be a category toward the right of the chart with Arrays, Lists and Maps. These are the collection classes.

In this application we need to store the points drawn by the user. This is most easily done using a pair of **CDWordArrays**. These are simple array classes that store values of type **DWORD** (a DWORD being a 32-bit signed integer), and we will store the x value of each point in one of the arrays and the y value of each point in the other. Click on the CDwordArray class in the hierarchy chart to learn more about the class. The MFC array classes have a number of useful features that will make our lives easier: The MFC arrays automatically size themselves as new elements are added and give you virtually infinite room, they do range checking in debug mode, and they know how to read and write themselves to disk.

Open the SAMPDOC.H file (probably the easiest way to do this is to double click on the CSampDoc class in the ClassView seen in tutorial 3, or go to the FileView and double click on SAMPDOC.H in the Header Files folder)

and find the public attributes section. There will be a section labeled "//attributes" with a "public:" marker within it. In that position type:

```
CDWordArray x, y;
```

If you then save the header file, the ClassView of the project window will display the two new variables in the CSampDoc class.

As an alternative, you can use the tools. Right-click on CSampDoc in the ClassView, and then choose **Add Member Variable...** from the list. You will have to do this twice, once for x and once for y. Both variables will be added to the public implementation section of the header file, and you can watch it happen if you have the header file open. For now, put x and y in the attributes section.

In this simple example we will treat the arrays as public members to make things more obvious, but in a real application it would be beneficial to make the data structure private and provide new member functions in the document class to allow the manipulation of the data structure. These functions would allow for the complete encapsulation of the document class.

Step 2: Allow for loading and saving of the document's data structure

The AppWizard framework and the MFC classes do a good bit of the work related to loading and saving files to disk. For example, when you choose the **Open** option of the **File** menu in your drawing program, it already pulls up the proper File Open dialog. The framework then takes the file name selected by the user, opens it, creates a binary "archive" and attaches it to that file, calls several functions in the **CDocument** class, and finally calls a function named **Serialize** in the CSampDoc class, passing it the archive (to see all of this code in action, finish adding the code described in this tutorial and put a break point in the document's **Serialize** function, run the program under the debugger, choose File Open, and then, when the program stops, choose the **Call Stack** option in the **View** menu. You will be able to examine the MFC source code). This same **Serialize** function is also called when it is time to save the file. All that you have to do is fill in this function and your data structure will automatically be loaded and saved to disk. To make matters even easier, the **CDWordArray** class (and all other MFC collection classes) know how to serialize themselves.

Find the **Serialize** function in the CSampDoc class. The easiest way to do this is to open the ClassView, click on plus sign next to CSampDoc, and then double click on the Serialize function. Change it so that it looks like this:

```
void CSampDoc::Serialize(CArchive& ar)
{
    x.Serialize(ar);
    y.Serialize(ar);
}
```

The **x** and **y** variables will handle all details of serialization, including understanding whether they should load or save themselves to the archive. The archive knows which direction the data should move.

Step 3: Modify the view class so it saves points into the document's data structure

First we will add a member variable to the view class to store the size of the rectangles that the editor draws, rather than using the hard-coded value of 5 as in tutorial 3. We will then later add a dialog that lets us change this value in order to demonstrate the addition of dialogs. Open SAMPVIEW.H and find the public attributes section. Add the following declaration:

```
int w;
```

Note that in this same section the view has a member function named **GetDocument** that returns a pointer back to the view's document. This function will be important in a moment.

Now find the constructor for the CSampView class (you can double-click on it in the ClassView and add the following line to initialize the new member:

```
w = 5;
```

In the CSampView class, find the **OnMouseMove** function that you added in tutorial 3 (you can find it by hand in SAMPVIEW.CPP, or use the ClassView). Change the function so that it looks like this:

```
void CSampView::OnMouseMove(UINT nFlags, CPoint point)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (nFlags == MK_LBUTTON)
    {
        CClientDC dc(this);
        dc.Rectangle(point.x, point.y,
            point.x+w, point.y+w);
        pDoc->x.Add(point.x);
        pDoc->y.Add(point.y);
        pDoc->SetModifiedFlag();
    }
    CView::OnMouseMove(nFlags, point);
}
```

Look up **OnMouseMove** in the MFC help file for information on its parameters. The easiest way to do this is to select Search in the Help menu and type OnMouseMove.

The function has been modified so that it retrieves a pointer to the view's document and then saves the current point in the document's **x** and **y** members. The function also calls the document's **SetModifiedFlag** function (see the MFC help file) to set the document's dirty bit. Once set, the document will automatically query the user about saving the file if the user attempts to close it or quit without saving. This function also makes use of the new **w** member in the view.

Step 4: Handle exposure events in the view class

Whenever the view class receives an exposure (WM_PAINT) event, it calls the **OnDraw** member function. This function also is called to handle printing. By putting exposure-handling code into this function you can complete the application. Find the **OnDraw** function in the CSampView class. Modify it as shown below:

```
void CSampView::OnDraw(CDC* pDC)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int i;
    for (i=0; i<pDoc->x.GetSize(); i++)
        pDC->Rectangle(pDoc->x[i],
            pDoc->y[i],
            pDoc->x[i]+w,
            pDoc->y[i]+w);
}
```

This function simply asks one of the arrays how many elements it contains and then runs through a **for** loop that many times, drawing all of the points that the user has previously entered.

Compile and run the application. You will find that it properly handles exposure events, and that it can now load and save files to disk. You have properly modified the document and view classes to create a complete, fully functional drawing editor. Congratulations!

Note that, as described in tutorial 3, the document class holds the data structure and loads and saves the data to disk. The view class uses the data in the document to handle redrawing, and manipulates the data when the user moves the mouse. This is proper use of the document and view classes.

It would not be a bad idea to now stop, close this tutorial, and try to recreate the application "blind". You will find that you generate a number of questions when you try to do it without help, and those questions can teach you a lot.

What we would like to do now is add a new menu option and dialog to the application so that the user can modify the size of the rectangles that the application draws. This exercise will show you how easy it is to modify the menu and create new dialog resources.

Step 5: Add a new menu option

In the application's workspace window choose the ResourceView tab and double click on the Menu resource section. You will find that the application has two menus. Double click on both and look at them.

IDR_MAINFRAME is short, and is used when there are no windows open in the MDI framework.

IDR_SAMPTYPE is longer and appears when windows are open. We want to modify IDR_SAMPTYPE, so open it by double clicking on it.

At the end of the menu bar you will find an empty rectangle. Click on it and type "&Option". Press the Return key. The & character indicates which letter in the menu title you want to use as a mnemonic, and will appear underlined when the menu is shown on the screen. You can move the & anywhere within the string. Now a new rectangle will appear below **Option**. Click on it and type "&Size". Press the Return key. Now click on **Option** and

drag it to a more appropriate place in the menu bar, perhaps between **Edit** and **View**. That's all it takes to create a new menu and option.

Now double-click on the new size option. Note that the editor has automatically assigned it the obvious ID of `ID_OPTION_SIZE`. You can change the ID as you see fit, but there is rarely a reason to do so. We will use this value inside a message map to respond to the new option. [note: If you are unfamiliar with the concept of a message map, visit the [MFC Tutorials page](#) and see the introductory MFC tutorial]

Build and execute the application. You will find that it has a new menu option but that the option is disabled.

Now open the ClassWizard by choosing the **ClassWizard** option in the **View** menu. Make sure the **Message Maps** tab is visible. Make sure the **CSampView** class is selected. In the **Object IDs** list choose `ID_OPTION_SIZE`. In the **Messages** list choose **COMMAND** (`UPDATE_COMMAND_UI` is used primarily to enable and disable the menu options - look up **CCmdUI** in the MFC help file for more information). Click the **Add Function** button. The ClassWizard will pick the obvious name **OnOptionsSize** for the new function and show it to you. Click **OK** to accept the name. Click the **Edit Code** button with the **OnOptionsSize** function selected and modify the function as shown below:

```
void CSampView::OnOptionsSize()
{
    Beep(500, 500);
}
```

Build and execute the application. Now when you choose the **Size** option you will find that the application beeps. You can see that it is trivial to wire new menu options into an AppWizard application.

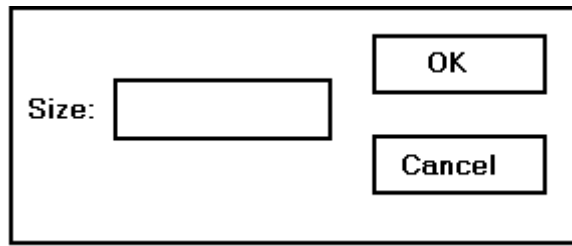
Step 6: Add a new dialog

To add a new dialog to the application we have to do four things:

- Create a new dialog template with the dialog resource editor,
- Create a new dialog class for that template,
- Add DDX variables(s) for the controls in the dialog, and
- wire the dialog into the application.

Here are the steps:

Select the **Resource...** option from the **Insert** menu. Choose **Dialog** and click on **New**. You should see a new dialog containing OK and Cancel buttons, and a small palette of tools should also appear. If the palette is not in evidence, choose the **Customize** option in the **Tools** menu and check the box marked **Controls**. One button on the palette says "Aa" and is used to create static text controls. Another is labeled "ab|" and is used to create editable text controls. Create a dialog that looks like this:



Click on the new edit control and press **Enter**. You will find its ID to be IDC_EDIT1. This is fine here, but in a real dialog you would likely change it to something more meaningful. Select the OK button and note its ID is IDOK. This is normal and you will not want to change it. Right click in the title bar of the new dialog itself and choose the **Properties** option. Note that the dialog's ID is IDD_DIALOG1. Note also that you can change its title here.

With the dialog still open on the screen, choose the **ClassWizard** option in the **View** menu. The ClassWizard will see the new dialog and assume that your desire is to create a new dialog class for it. This class will act as a liaison between the dialog resource and your application, and you will need to create a new dialog class for each dialog that you add to an application (although you will rarely or never touch this class except through the ClassWizard). The first dialog you see lets you specify that you want to create a new dialog class. There are several fields in the new dialog class creation dialog. In the **Name** field type "CSizeDlg". Make sure that **Base Class** contains **CDialog**, that **File** contains SIZEDLG.CPP, that the **Dialog ID** field contains the dialog's ID of IDD_DIALOG1, and that **OLE Automation** is set to None. Click the **OK** button to create the dialog class. Click the **OK** button in the ClassWizard to close it.

To try out the new dialog, Find the **OnOptionsSize** function in the CSampView class. Change it so it appears as below:

```
void CSampView::OnOptionsSize()
{
    CSizeDlg dlg;
    dlg.DoModal();
}
```

Also, be sure to include SIZEDLG.H as the last header file included in SAMPVIEW.CPP.

Build and execute the application and select the size option. You will find that the dialog appears properly and disappears as expected when you click the OK or Cancel buttons.

What we would like to do now is get the value typed by the user into the edit field so that we can modify the view's **w** member. Select the **ClassWizard** option in the **View** menu. Select the **Member Variables** tab at the top of the ClassWizard. Make sure that **Class Name** is set to **CSizeDlg**. We want to add a member variable to the **CSizeDlg** class to allow us to get the value from the dialog's edit control. In the list, double click on IDC_EDIT1. In the dialog that appears, set the **Member Variable Name** to "m_size". Set the **Category** to "Value". Set the **Variable Type** to UINT. Click the OK button. In the new **Minimum Value** and **Maximum Value** fields that appear at the bottom of the ClassWizard type 2 and 50 respectively.

The **m_size** variable is a **DDX** variable. DDX=Dialog Data Exchange. This new variable will always contain the value that the user types into the edit control, or you can set it to display a default value to the user. The values you typed for the minimum and maximum are known as **DDV** values. DDV=Dialog Data Validation. Anything the user types will be checked against these values when the user clicks the dialog's OK button.

Replace the code in **OnOptionsSize** with the following:

```
void CSampView::OnOptionsSize()
{
    CSizeDlg dlg;
    dlg.m_size = w;
    if (dlg.DoModal() == IDOK)
        w = dlg.m_size;
}
```

Build and execute the program. You will find that if you change the value in the dialog, the size of the rectangles drawn by the application will change appropriately. The code is simply setting or retrieving the value from the edit control using the **m_size** variable as its proxy. The value in **m_size** is copied into the edit control when the dialog appears, and the value in the edit control is copied into **m_size** when the user clicks the OK button.

You may notice that the edit control does not initially have focus. Fix this by opening the dialog resource and choosing the **Tab Order** option in the **Layout** menu. Click on each control in order to establish the tab order.

You may want to store the **w** value with each point that the user draws. To do this, add a new **CDWordArray** variable to the document class, serialize it appropriately, and change the view class to set and retrieve this array's values in the same way you change **x** and **y**.

Conclusion

In this tutorial you have seen how easy it can be to create and modify a very robust and capable program using the AppWizard, ClassWizard and the resource editors. In the next tutorial we will fix one niggling little problem left in this application.

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x Synchronizing Views

In the previous tutorial you learned how to modify the document and view classes to create a simple drawing editor. There is one subtle problem with that program, however. In this tutorial you will learn how to solve that problem using the view's **OnUpdate** function.

To demonstrate the problem, run the application that you created in the previous tutorial. Draw something in the default window. Now choose the **New Window** option in the **Window** menu. This option opens a second view on the same document. This second window will display the same thing that the first window does because both share the same document. Now choose the **Tile** option in the **Window** menu. You can see that both views are identical. Now draw into one of the views. What you will find is that the views will not be synchronized. What you draw into one view does not appear in the other. However, if you iconify the application and then expand the icon, you will find that the views are once again identical. Both receive exposure events, and both draw from the same document data, so they must look the same.

What we would like to do is modify the code so that, when you draw in one view, all views attached to the same document are immediately updated as well. The framework already contains the functions necessary to do this—all you have to do is wire them in properly.

The **CDocument** class maintains a list of all views attached to the document. It also contains a function called **UpdateAllViews**. This function, when called, calls the **OnUpdate** function of each view attached to the document. By default the **OnUpdate** function does nothing, but you can modify it to do anything you like. Optionally you can pass the **OnUpdate** function two programmer-defined parameters to further customize its activities.

What we would like to create here is a mechanism that causes all views attached to a document to paint the last point added to the data structure whenever any of the views for that document adds a new point. To do this, first modify the **OnMouseMove** function in the view class so that it contains a call to **UpdateAllViews**, as shown below:

```
void CSampView::OnMouseMove(UINT nFlags, CPoint point)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (nFlags == MK_LBUTTON)
    {
        CClientDC dc(this);
        dc.Rectangle(point.x, point.y,
            point.x+w, point.y+w);
        pDoc->x.Add(point.x);
        pDoc->y.Add(point.y);
        pDoc->SetModifiedFlag();
        pDoc->UpdateAllViews(this, 0, 0);
    }
    CView::OnMouseMove(nFlags, point);
}
```

This call to **UpdateAllViews** indicates that the document should call the **OnUpdate** function in all views attached to it *except* the one indicated by **this**. It does this because the current view has already drawn the point and there is no reason to do it a second time. The latter two parameters in the call to **UpdateAllViews** will be passed directly to **OnUpdate**. We do not have any use for these parameters in this simple example so we pass zeros. It would not hurt to read about both **CDocument::UpdateAllViews** and **CView::OnUpdate** in the MFC help file. Also read about **CView::OnInitialUpdate** while you are there.

Now use the ClassWizard to override the **OnUpdate** function. Choose the **ClassWizard** option in the **View** menu. Make sure that the **Message Maps** tab is selected. Make sure that **CSampView** is the class selected in the Class Name field. Click on **CSampView** in the **Object IDs** list. Search down until you find **OnUpdate** in the **Messages** list. This function is a virtual function and we can override it with the ClassWizard. Select **OnUpdate** in the list, click the **Add Function** button and then click the **Edit Code** button. Modify the function so that it looks like this:

```
void CSampView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    CSampDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    int i = pDoc->x.GetSize();
    if (i > 0)
    {
```

```

        i--;
        CClientDC dc(this);
        dc.Rectangle(pDoc->x[i],
            pDoc->y[i],
            pDoc->x[i]+w,
            pDoc->y[i]+w);
    }
}

```

The goal of this function is to get the last point in the data structure and draw it. It therefore gets the size of one of the arrays, checks to make sure that the array is not empty, and then draws the last point. The **if** statement is necessary because the **OnInitialUpdate** function gets called when the view is created, and by default it calls **OnUpdate**. You could override this function to remove the default behavior and the **if** statement would no longer be necessary. However, it is not a bad safety feature.

Build and execute the application. Choose the **New Window** option in the **Window** menu, followed by the **Tile** option. Draw in one of the windows and you will find that both views update simultaneously. This is proper behavior, and will work regardless of the number of views that are open on the same document. It is also very efficient.

There are other ways in which to use the **UpdateAllViews/OnUpdate** to accomplish the same thing. For example, **OnMouseMove** might draw nothing and let the **OnUpdate** function handle all drawing. Or you might pass the new point as one of the parameters. Experiment with different techniques until you find the one you like best.

Understanding the AppWizard and ClassWizard in Visual C++ Version 6.x

Understanding Document Templates

One of the more interesting, and best hidden, features of any AppWizard framework is something called *document templates*. In this tutorial you will learn about document templates and see how you can add new ones to your applications. By the end of the tutorial you will have used document templates to create a single MDI application that can display both text and drawing documents simultaneously.

Creating a Text Editor

Let's start by using the AppWizard to create a second type of application. In the previous tutorials we have created a drawing editor. Here we will quickly create a text editor. It is interesting to note that you can create a complete text editor - one with all the features of Notepad, along with several others as well - without writing a single line of code. Take the following steps:

- In Visual C++, select the **New** option from the **File** menu, make sure the **Project** tab in the subsequent dialog is selected, and name the new project "Ed". Make sure the type is set to **MFC AppWizard (EXE)** and select an appropriate directory. Click **OK** and look over the AppWizard's options in the six configuration screens.
- We want to change two things in the configuration screens: First we want to give a default file extension, and second we want to change the view class. In the fourth screen of the six, click the **Advanced** button and type "tex" into the **File Extension** field. In the sixth screen, click on **CEdView**, and at the bottom of the dialog change the **Base Class** to **CEditView** using the combo box.

- Compile and run the program. You will find that you have a complete MDI text editor. You can load and save text files, cut, copy and paste text, print files, and so on.
- If you look at the help page for the **CEditView** class, you will find that it automatically understands certain menu options. In particular, if you add menu options with the IDs of ID_EDIT_FIND, ID_EDIT_REPEAT, ID_EDIT_REPLACE and ID_EDIT_SELECT_ALL, the program will *automatically* recognize these new options and perform them properly. You do not need to add anything but the menu options. Do that now and test the program.

This application was so easy to create because the **CEditView** class has all of the behavior of a normal text editor built into it. There is just one line of code that the AppWizard had to add to make the whole thing work, and that line can be found in the **Serialize** function of the document class. It looks like this:

```
((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
```

That line loads and saves text files. Just so you are aware of it, the **CEditView** class violates the strict separation of document and view. The **CEditView** class contains a normal **CEdit** control, and this control holds the editor's data itself. Therefore, the data resides inside the **CEditView** class rather than in the document class, and the line of code above gets or sets that data. Because of this odd structure, you will want to remove the **New Window** option from the **Window** menu. Since the document does not hold the data, it is not possible to have multiple views display the same document. This seems like a small price to pay for the ease of using the **CEditView** class to create quickie text editors.

Now that you have created a complete text editor, let's see what steps are necessary to create a single MDI application that can display both text and drawing documents. To do this, we will take the drawing program from the previous tutorials and modify it so that it can also display text documents. To do this, three steps are required:

- Step 1: Start with the drawing program and add a new document and view class for the text editor
- Step 2: Create a new document template for the new document type
- Step 3: Add three new resources to the drawing editor

Once you have completed these three steps, the program will be able to display both text and drawing documents simultaneously.

Step 1: Add a new document and view class

Open the workspace file for the drawing editor (samp) in Visual C++. Choose the **ClassWizard** option in the **View** menu. Click the **Add Class** button and select **New**. You will see a dialog with several fields. In the **Name** field type **CEdDoc**. In the **Base Class** field choose **CDocument**. The ClassWizard will choose a file name of EDDOC.CPP, and this name is fine. Click the **OK** button. Click the **Add Class** button again to create another new class. Type **CEdView** into the **Name** field and choose **CEditView** for the base class type. The file name EDVIEW.CPP chosen by the ClassWizard is fine. Click the **OK** button. Close the ClassWizard by clicking its **OK** button.

Now modify the **Serialize** function in the new document class (CEdDoc) so it contains the line seen in the text editor:

```
((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
```

The two new CPP files were automatically added to the drawing project by the ClassWizard. You will add the header files to the CSampApp class in the next step.

Step 2: Add a new document template

Open the CSampApp class, which contains the application class for the application derived from **CWinApp**, in the ClassView so that you can see a list of its functions. Find the **InitInstance** function. Double click on it. Look for the following lines:

```
CMultiDocTemplate* pDocTemplate;  
pDocTemplate = new CMultiDocTemplate(  
    IDR_SAMPTYPE,  
    RUNTIME_CLASS(CSampDoc),  
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame  
    RUNTIME_CLASS(CSampView));  
AddDocTemplate(pDocTemplate);
```

These lines create a *document template*. The **CWinApp** class (see the help file) has built into it the ability to hold a list of document templates. When it holds more than one, it changes the behavior of the application slightly. For example, the **New** option pops up a list that lets the user choose what type of document he/she wishes to create. What we want to do is change the program so that it contains two templates: one for drawing documents, and another for text documents. Modify the above code so that it looks like this:

```
CMultiDocTemplate* pDocTemplate;  
pDocTemplate = new CMultiDocTemplate(  
    IDR_EDTYPE,  
    RUNTIME_CLASS(CEdDoc),  
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame  
    RUNTIME_CLASS(CEdView));  
AddDocTemplate(pDocTemplate);  
pDocTemplate = new CMultiDocTemplate(  
    IDR_SAMPTYPE,  
    RUNTIME_CLASS(CSampDoc),  
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame  
    RUNTIME_CLASS(CSampView));  
AddDocTemplate(pDocTemplate);
```

Note that a new document template has been created. The new one goes first (more on the reason for that in a moment). It specifies **IDR_EDTYPE**, **CEdDoc** and **CEdView**. But what does that mean?

The purpose of a document template is to relate a resource type (**IDR_EDTYPE**), a document class, a view class, and a window class. When the application framework needs to create a new instance of a document for the user, it looks to the document template, which tells it to create a new instance of the appropriate document, view and window classes. The resource ID is used when the framework needs to change resources. It identifies a specific menu, icon and string resource. So, for example, when the user clicks on a window in the MDI shell, the application framework brings that window to the foreground *and* it changes the menu to the one appropriate for that window, according to the window's document template.

We put the text document template first because, if the user attempts to open a document whose extension is unknown to the application, the application tries to open it under the first document template registered. Since text documents are far more likely than drawing documents, the text template is placed first in the list of document templates.

Be sure to include EDDOC.H and EDVIEW.H at the top of SAMP.CPP.

Step 3: Create resources

The new document template specifies a resource ID of IDR_EDTYPE. If you open the ResourceView and look through its resources, you will find that it already contains three resources of type IDR_SAMPTYPE as needed by the drawing editor: a menu, an icon, and a string near the top of the string table. The easiest way to create new resources for the text editor type is to copy these three IDR_SAMPTYPE resources to the clipboard, paste them back, and then change their names to IDR_EDTYPE using the **Properties** option in the **View** menu. Then modify them as appropriate. For example, to the IDR_EDTYPE menu you will want to add the ID_EDIT_FIND, ID_EDIT_REPEAT, ID_EDIT_REPLACE and ID_EDIT_SELECT_ALL options (also delete the **Option** menu that got copied). You will also want to remove the **New Window** option from the Window menu. Change the IDR_EDTYPE icon as you see fit. Change the IDR_EDTYPE string so that it looks like this:

```
\nEd\nEd\nEd Files (*.tex)\n.TEX\nEd.Document\nEd Document
```

For more information about this mysterious string, search for the **GetDocString** function in the help file. It will explain what all seven of the substrings do. Now that you understand the strings, modify the IDR_SAMPTYPE string as well so it contains a file extension:

```
\nDrawing\nDrawing\nDrawing Files (*.drw)\n.DRW\nDrawing.Document\nDraw Document
```

Change the two strings in any way that you like.

Step 4 : Build and execute

Build the new application and run it. When it starts you should see a new dialog that lets you choose whether you want to create a text or drawing document. Choose text, and verify that the text editor works properly. Now choose **New** and create a drawing document. Draw something. Note that when you change windows the menu bar changes as appropriate.

Conclusion

You can see that adding a new document template is easy, and there is really no limit to the number of templates a single application might have. As you create more complex applications, you will find this to be an extremely useful feature of the AppWizard framework.

Adding Context Menus to your Visual C++ / MFC Application

Introduction

Windows applications are increasingly making use of context menus in various parts of their GUIs. To be consistent with other Windows apps, and to meet your users' expectations, context menus should have a place in your applications as well.

This article will show you how to easily add context menus to your Visual C++ application and will illustrate the process with a working example. I'll also discuss the issues I encountered as I learned how to do this myself, so that you can be aware of these potential hurdles. The source for the entire example project (created with VC++ 5.0) is available for [download](#).

Basics

The general process is simple. The three basic steps for creating a context menu are:

- Create the menu resource(s)
- Add a message handling function for WM_CONTEXTMENU
- If the point clicked is in the desired area, create a CMenu object and call its member TrackPopupMenu

Example

The example project is just a simple dialog-based application, which has a tree control in the dialog (see Figure 1). For the example, I wanted to respond to right-clicks on items in the tree control. I'll present the step by step process of creating the context menus and discuss what I did specifically for the example.

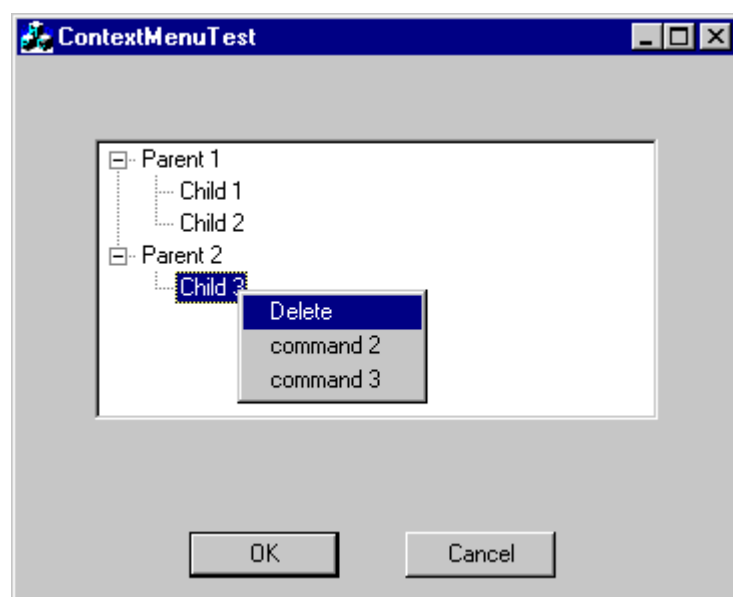


Figure 1 -- Dialog-based application

Step 1 -- Create the menu resource(s).

Using the resource editor, add a new menu. Type anything you want for the top level item in the menu. It doesn't matter what you put there; it won't be displayed anyway. In the popup menu items that appear below the top level item, create your context menu. See [Figure 2](#).

For this example, I created two menus so that I could display two different context menus depending on where the mouse click occurred. One menu (IDR_MENU2) has three items (Delete, Command 2, and Command 3). I want to display this menu when clicking on a child item in the tree. The other menu (IDR_MENU1) has only one item

(Delete), and will be displayed when clicking on a parent item in the tree. It would be equally valid to have one single menu resource with two top-level items and submenus instead of two separate menu resources.

You will, of course, need to wire the menu choices to actual methods if you want them to do anything. The easiest thing to do is to use Class Wizard to create handlers for the menu items. For the example, I created a handler for the Delete menu item, and it just removes the item from the tree.

Step 2 -- Add the WM_CONTEXTMENU message handler.

Using the Class Wizard, add a handler function for the WM_CONTEXTMENU message. That will result in a new method called OnContextMenu(CWnd* pWnd, CPoint point). The OnContextMenu() method will be called each time there is a right-click inside the application window.

Step 3 -- Implement the WM_CONTEXTMENU message handler.

This step is the real meat of this exercise. In the message handler, we must decide whether or not to display a context menu. If the method decides not to display the context menu, then we'll pass the message through to the base class, so the WM_CONTEXTMENU message can be processed there.

One important thing to be aware of is the frame of reference for the point of the mouse click. The CPoint parameter coming into the method is in screen coordinates. We'll have to convert it to other reference points for use in other methods. Keep in mind that the upper left corner of the display is considered (0, 0) and the coordinates increase as you go left to right and top to bottom.

The determination of whether or not a menu should be displayed is based on the location of the right mouse click. So you'll need to convert that point to be relative to the window you're interested in. If you want to respond to right-clicks anywhere in the window, then you'll need to convert the coordinates to be relative to the client area of the window. If you're only interested in clicks inside a control in the window, then you'll need the point to be relative to that control. For the example, we're only interested in clicks inside the tree control. More specifically, we're interested in clicks on an item in the tree control. If a tree item wasn't clicked, then there's no need to display the menu. We'll just pass the message on to the base class. But to make that determination, we must first convert the coordinates so that they are relative to the tree control by using CWnd::ScreenToClient().

Now we can determine what item (if any) was clicked in the tree. Luckily, CTreeCtrl provides the HitTest() method that does just that. (Not all controls provide a HitTest method. You may have to write your own.) The UINT parameter you give it comes back with a bitmasked value that indicates where the hit occurred. See the documentation for CTreeCtrl::HitTest() for more information on the flags coming back in the UINT parameter. For the example, I only wanted to pop up the menu if the item's label or image was clicked, so I checked to see if the bit corresponding to TVHT_ONITEM was set in the mask.

Now, assuming we've made it this far, and an item was clicked, we're almost ready to display the menu. For the tree control, we have to explicitly select the item that was clicked, so that the menu's message handler will work properly. Additionally, for the example, we have to determine which context menu to display. As stated above in Step 1, the menu displayed depends on whether the item clicked was a parent item or a leaf item. So create the menu and load it using the appropriate resource. Since we want to display the pop-up portion of the menu and not the top-level, we call GetSubMenu(0) to get a pointer to the pop-up menu.

The final step to display the menu is to call `CMenu::TrackPopupMenu()`. This method displays the menu and handles the selection. Check the documentation for details on parameters. So here's the finished `OnContextMenu()` method.

Summary

So, as you can see, adding context menus to your application is not difficult at all. Just by adding a new menu resource or two, and implementing a handler method for the `WM_CONTEXTMENU` message, you're well on your way to having functioning context menus. And with some attention to a couple of details like point coordinates and proper treatment of unprocessed messages, you've got it licked.

Terms

Context Menu -- The floating menu that pops up when you right-click an object

Client Area -- The internal area of a window that excludes the title bar and frame

Frame of Reference -- The point to which another point is relative can also be thought of as the frame of reference. In other words, if point A is relative to point B, then point B serves as the origin of point A's coordinate system.

Screen Coordinates -- Point coordinates that are relative to the upper-left corner of the screen

WM_CONTEXTMENU message handler

```
void CContextMenuDlg::OnContextMenu(CWnd* pWnd, CPoint screenPoint)
{
    // Make a copy and convert the point to be
    // relative to the tree control
    CPoint treePoint = screenPoint;
    m_tree.ScreenToClient(&treePoint);

    // Now get the item that was clicked (if any).
    UINT nFlags;
    HTREEITEM hItemClicked = m_tree.HitTest(treePoint, &nFlags);

    // If an item's label or bitmap was clicked,
    // then we'll display the menu
    if ((NULL != hItemClicked) && (TVHT_ONITEM & nFlags))
    {
        // Select the item
        m_tree.SelectItem(hItemClicked);

        // Get the menu appropriate to the item clicked
        CMenu menu;
        if (m_tree.ItemHasChildren(hItemClicked))
        {
            // Parent item
            menu.LoadMenu(IDR_MENU1);
        }
        else
        {

```

```

        // Child item
        menu.LoadMenu(IDR_MENU2);
    }

    // Display the menu (use screen coords)
    menu.GetSubMenu(0)->TrackPopupMenu(
        TPM_LEFTALIGN | TPM_RIGHTBUTTON, screenPoint.x,
        screenPoint.y, this);
}
else
{
    // We're not going to display the menu,
    // so pass the message to the base class
    CDialog::OnContextMenu(pWnd, screenPoint);
}
}

```

Custom MFC Base Classes for Fun & Profit

Distributing Code Changes Across Your Applications

The Problem

Somewhere today in a cubicle, the following scenario is playing out. An unsuspecting Windows developer has just completed the fiftieth dialog class in a project, and the phone rings. Marketing has called to tell our fellow developer that Legal has decided that all of his dialogs must have disclaimer text in every dialog caption.

In my latest latest personal encounter with this situation, a "slight code change" needed to be added so third-party help systems could execute correctly. This code change had to be added to every CDialog-based class. Before resigning myself to a long day of hacking dialog code (if lucky), I took a walk and thought about another option.

The Solution

The good news is, there is a solution. The bad news is, it won't help someone in the above situation avoid making changes to every dialog class. The first time, all of the dialogs will have to be modified, but only slightly. However, after this first set of changes, should the dialogs ever need to be tweaked again, you'll need to make only a single change. If implemented when at the start of the project, however, you'll avoid ever needing to touch all of the dialogs.

The solution requires simply putting a piece of code between our dialogs and the MFC CDialog class. This is very easy to do, and provides you with a way to make a single code-change propagate through each dialog in your project.

The Example

To demonstrate the new class, I built a new project called "DevJournal" using the Visual Studio Application Wizard. I selected Dialog-based, and allowed the remainder of the options to remain at their default values. Incidentally, I am using Visual Studio 6.0. I will try my best to not use things that are 6.0-specific, but it may happen. If you have trouble in version 5.0, write me, and I will try to address the issues.

My normal working method is to build and execute small pieces to avoid not knowing which change made what break. Accordingly, once I have the DevJournal project set up in Visual Studio, I build it, both Debug and Release. Execution displays a single dialog with OK and Cancel buttons, and the text "TODO: Place dialog controls here."

Enabling the About Box

Once this is executing correctly, we can get into the code a bit. First, pull up the resource for the main dialog. In my case, it was named IDD_DEVJOURNAL_DIALOG. I selected, and deleted the Static "TODO: Place dialog controls here" text, and moved the OK and Cancel buttons to the bottom and right. To the left of OK, I added a button labeled "About" with an ID of IDC_BABOUT_DEVDLG.

Double-click the About button to add a method for handling the about, and to place yourself into the method. Add the following code to launch the About box that MFC built for you:

```
CAboutDlg dlg;  
dlg.DoModal();
```

Now rebuild and run the application to make sure the About box appears on cue.

The Change Request

At this point, we have the application prior to the Legal department's request. In my example, I'm going to suggest that Legal wants the background of each dialog to be painted with the company logo. In our example, that is only two, but the code would have to be inserted twice, so let's examine how we can do it only once.

Building our Base Class

From the menu bar select Insert|New Class. I will use CDJDialog as the Name and select CDialog as the Base Class. Take all other defaults, and press OK. Visual Studio will now give you a warning that you have no resource for this class, because it is smart enough to realize that you are building a CDialog-based class. Select Yes here. If you try to rebuild at this point, you will receive an error message because of an undefined ID "_UNKNOWN_RESOURCE_ID_" in the AFX_DATA section of DJDialog.h. This is correct so far.

Now we modify our DJDialog.h file as follows. Change the constructor from

```
CDJDialog(CWnd* pParent = NULL);
```

to

```
CDJDialog(UINT IDD, CWnd* pParent = NULL);
```

This is because we are going to pass in the IDD to our parent, and we keep the defaulted pParent variable to the right of our UINT. Remove the enum line from the AFX_DATA section. Each CDJDialog-based class will have its

own enum, and pass that value in as the IDD in the constructor. For now, that is enough for the include file. Now we modify the DJDialog.cpp file as follows. Modify the constructor from

```
CDJDialog::CDJDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CDJDialog::IDD, pParent)
```

to

```
CDJDialog::CDJDialog(UINT IDD, CWnd* pParent /*=NULL*/)
    : CDialog(IDD, pParent)
```

This passes the IDD that our new class was instantiated with to the CDialog base class in the same manner that it receives the member enum IDD. Rebuild at this point to ensure a clean compile, and no changes in the manner of execution.

Wiring our Class into the Project Files

Now we insert our class between our application and the CDialog. Modify DevJournalDlg.h as follows. First, include our new dialog class just above the CDevJournalDlg class declaration.

```
#include "DJDialog.h"
```

Change the constructor from

```
class CDevJournalDlg : public CDialog
```

to

```
class CDevDialogDlg : public CDJDialog
```

Modify the DevJournalDlg.cpp as follows by changing the CAbout class declaration from

```
class CAboutDlg : public Cdialog
```

to

```
class CAboutDlg : public CDJDialog
```

Modify the CAbout constructor from

```
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
```

to

```
CAboutDlg::CAboutDlg() : CDJDialog(CAboutDlg::IDD)
```

Modify the CAbout BEGIN_MESSAGE_MAP from

```
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
```

to

```
BEGIN_MESSAGE_MAP(CAboutDlg, CDJDialog)
```

Modify the CDevJournalDlg constructor from

```
CDevJournalDlg::CDevJournalDlg(CWnd* pParent /*=NULL*/)
: CDialog(CDevJournalDlg::IDD, pParent)
```

to

```
CDevJournalDlg::CDevJournalDlg(CWnd* pParent /*=NULL*/)
: CDJDialog(CDevJournalDlg::IDD, pParent)
```

As with the CAbout class, above, modify the BEGIN_MESSAGE_MAP entry from

```
BEGIN_MESSAGE_MAP(CDevJournalDlg, CDialog)
```

to

```
BEGIN_MESSAGE_MAP(CDevJournalDlg, CDJDialog)
```

This is all that is necessary to base our dialogs on our new class. Build and run to ensure completeness at this point. Don't yawn yet, because if it runs as it did before that means your subclass is working along with everyone else!

Testing the Base Class

To prove the CDJDialog class is functional, bring up the class wizard, select the CDJDialog class, and add a member function for the WM_INITDIALOG message. Take the default, and then press "Edit Code". In place of the "TODO ..." line insert something like

```
AfxMessageBox("CDJDialog OnInitDialog", MB_OK);
```

This won't do anything, unless we also modify our two CDJDialog-based classes. In CDevJournalDlg.cpp , modify the base-class call in OnInitDialog() from

```
CDialog::OnInitDialog();
```

to

```
CDJDialog::OnInitDialog();
```

The CAbout class does not currently handle the WM_INITDIALOG method. In a manner similar to modifying CDJDialog, add a WM_INITDIALOG handler to the CAbout class. Since this code is added after our base class substitution, Class Wizard will automatically be aware of CDJDialog, and no changes to that part of the code are required.

Build and run to test. Because of the AfxMessageBox placement, the message appears prior to either dialog appearing. This is a good test of the execution of our base class.

At this point, the hard work is finished. Any functionality you would like to have appear in every dialog class can now be added to a single class, and will immediately work in every dialog in your project.

Drawing the Background

In all the excitement, I almost forgot the Legal department's request. I have a bitmap named DJ.BMP that I want to use to paint the dialogs. To do so, we must first bring it into the project.

Begin with the DJ.BMP file in the 'res' directory of your project. In Visual Studio, from the menu select Insert|Resource|Bitmap|Import. Select DJ.BMP in the project res directory and press Import. This will add the resource with the ID IDB_BITMAP1. Double-click the right-hand client area of Visual Studio, but outside the bitmap area. If you click on the bitmap you will modify the image. The double-click will bring up properties of the bitmap. I changed the ID to be IDB_DJ.

Once more, invoke Class Wizard. This may be a good time to mention that I make sure the Class Wizard toolbar item is readily available. For reference, the image is in the 'View' category in Tools|Customize.

Inside Class Wizard, make sure the Message Map tab is selected, and select CDJDialog as the Class. Under Messages, select WM_PAINT, double-click, and accept the default OnPaint member function. Select Edit code to bring up the dialog code. Explaining the code that effects the dialog painting is probably more than needs to be included here. Please email me if you would like to see that code explained.

Essentially, the bitmap is loaded from its bound location in the application's resources, and selected into a memory device context. The memory device context is then stretched to fit into our dialog's client area, causing either an expand or contract to take place. A few housekeeping chores also take place, but essentially this is it. The interesting thing to note here is that the device context manipulated is that of each of our original dialogs, all done by that code in our new base class.

Replace the Wizard-created "TODO..." line with the following:

```
CBitmap bmDlg;  
bmDlg.LoadBitmap(IDB_DJ);  
CDC dcMemory;  
dcMemory.CreateCompatibleDC(&dc);  
CBitmap *pBitmapOld = dcMemory.SelectObject(&bmDlg);  
  
CRect RectDlg;  
GetWindowRect(&RectDlg);  
BITMAP bm;  
bmDlg.GetBitmap(&bm);  
dc.StretchBlt(0, 0, RectDlg.Width(), RectDlg.Height(),  
             &dcMemory, 0, 0, bm.bmWidth, bm.bmHeight, SRCCOPY);  
dcMemory.SelectObject(pBitmapOld);
```

Build and run the application to check your results.

You will notice that the About dialog box picked up the background while the main dialog did not. A little investigation will show why. If you open the class view of Visual Studio, you'll notice that CAbout does not handle the OnPaint method. Therefore that task is passed on to the base class, CDJDialog, and we just wrote the code to do that task.

Why, then, does CdevJournalDlg, which does handle OnPaint, not work as planned? First, CDevJournalDlg handles OnPaint for displaying an icon when minimized. If we look at the OnPaint() code, we find an else clause that calls the base class. In most cases, calling CDialog methods would work just fine, but in this particular case, we have our own OnPaint(), and we should be calling CDJDialog::OnPaint() instead.

Make this change, and before building, comment out our now-unnecessary AfxMessageBox line in CDJDialog::OnInitDialog().

If all goes well, you should be seeing bitmapped backgrounds on both dialogs.

Conclusion

Once you understand the concepts, the actual work of doing this should go fairly quickly. I have considered adding a custom CDialog-based class to every future project for just this sort of use.

Creating a Simple MFC Program

Introduction

MFC was created to make programming in Windows easier. As an object-oriented wrapper for Win32, it automates many routine programming tasks (mostly passing references around). Paradigms like the document/view architecture were added to automate even more tasks for the programmer, but in the process, some control was taken away.

This tutorial will show you how to create a very simple MFC program and how to customize it for your application. This program will be a base for other tutorials that I write for OpenGL and DirectX. I will be going through this tutorial in Visual C++ 6.0.

- [\[Download simple MFC_basic.zip \]](#)
- [\[Download simple MFC_extended.zip \]](#)

We will begin by creating a standard MFC Doc/View application using AppWizard. Select New from the File menu, click on the Projects tab, and select MFC AppWizard (exe). Enter SimpleMFC as the project name and press OK.

1. Select Single Document and press Next.
2. Select None for database support and press Next.
3. Select None for compound document support and remove support for ActiveX controls. Press Next.
4. Deselect all of the check boxes in Step 4. Press Next.
5. Generate comments and use a shared DLL. Press Next.
6. Press Finish.

We now have a standard Doc/View application with the following classes:

CAboutDlg
CMainFrame
CSimpleMFCApp
CSimpleMFCDoc
CSimpleMFCView

Go to the file view and remove the following files from the project and delete them from the directory:

SimpleMFCView.h
SimpleMFCView.cpp
SimpleMFCDoc.h
SimpleMFCDoc.cpp

You can also remove IDR_SIMPLETYPE from the icon resource tab and delete the associated file SimpleMFCDoc.ico (be sure to do it in that order).

We now have a project that doesn't compile.

Edit SimpleMFC.cpp:

- Remove the #includes for SimpleMFCView.h and SimpleMFCDoc.h.
- Edit the constructor to look like this:
 -
 - `CSimpleMFCApp::CSimpleMFCApp()`
 - `{`
 - `m_pMainWnd = NULL;`
 - `// Place all significant`
 - `// initialization in InitInstance`
 - `}`
 -
- Edit `initInstance` to look like this:
 -
 - `BOOL CSimpleMFCApp::InitInstance()`
 - `{`
 - `// Standard initialization`
 - `// If you are not using these features`
 - `// and wish to reduce the size`
 - `// of your final executable, you should`
 - `// remove from the following`
 - `// the specific initialization routines`
 - `// you do not need.`
 -
 - `#ifdef _AFXDLL`
 - `Enable3dControls();`
 - `// Call this when using MFC in a shared DLL`
 - `#else`
 - `Enable3dControlsStatic();`
 - `// Call this when linking to MFC statically`
 - `#endif`
 -
 - `// The one and only window has been initialized,`
 - `// so show and update it.`
 - `m_pMainWnd = new CMainFrame();`
 - `m_pMainWnd->ShowWindow(SW_SHOW);`
 - `m_pMainWnd->UpdateWindow();`
 -
 - `return TRUE;`
 - `}`
 -
- Remove everything below `InitInstance`. This includes `CAboutDlg` and the code to bring up the About dialog. (If you would like to leave the About dialog in, feel free to do so. You will be able to activate it later.)

- Remove the items in the message map at the top of the file. This section should look like this:
-
- `BEGIN_MESSAGE_MAP(CSimpleMFCApp, CWinApp)`
- `//{{AFX_MSG_MAP(CSimpleMFCApp)`
- `// 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`
- `// Standard file based document commands`
- `END_MESSAGE_MAP()`
-

Remove the definition for OnAppAbout from SimpleMFC.h.

We still have a program that doesn't compile. CMainFrame is expecting to be created dynamically within the Doc/View architecture, thus its constructor is protected. We will fix that.

Edit MainFrm.h:

- Remove "DECLARE_DYNCREATE(CMainFrame)"
- Change "protected: // create from serialization only" to "public:"

Edit MainFrm.cpp:

- Remove "IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)".
- Edit the constructor to look like this:
-
- `CMainFrame::CMainFrame()`
- `{`
- `RECT r;`
- `r.left = 100;`
- `r.top = 100;`
- `r.right = 200;`
- `r.bottom = 200;`
-
- `Create(NULL, "Simple MFC App",`
- `WS_POPUP|WS_THICKFRAME,`
- `r, NULL, NULL, 0, NULL);`
- `}`
-

If you run the program, a window with a border will appear. It will not have a title or menu bar. This is about the simplest Windows application you can make. Press Alt-F4 and Windows will send a WM_CLOSE message to the application to close it.

Let's take a look at what is going on here. CSimpleMFCApp is your application derived from CWinApp. CWinApp contains a pointer to the main application window. We have a global instance of CSimpleMFCApp. When it is created it instantiates an instance of our CMainFrame and assigns it as the CWinApp main window. It then displays the window. This is all that the application has to do. We will not be modifying it further.

CMainFrame creates the actual internal system window. Remember that MFC is just a wrapper for Win32. CMainFrame, which is derived from CFrameWnd, is not actually an MS-Windows window. It just manages the handle to a window that MS-Windows creates. (The fact that the product is named Windows and the objects themselves are called "windows" may make this confusing.)

In the Doc/View architecture, the window handle is created "dynamically" (which basically means the base class creates it in a predefined way). Since we are not using the Doc/View architecture, and we want more control over the window, we create the window handle ourselves by calling CFrameWnd::Create in the CFrameWnd constructor. CFrameWnd::Create eventually calls the Win32 create, but it manages the window handle for us. Create takes several parameters which are the key to customizing this window.

```
BOOL Create( LPCTSTR lpszClassName,
            LPCTSTR lpszWindowName,
            DWORD dwStyle = WS_OVERLAPPEDWINDOW,
            const RECT& rect = rectDefault,
            CWnd* pParentWnd = NULL,
            LPCTSTR lpszMenuName = NULL,
            DWORD dwExStyle = 0,
            CCreateContext* pContext = NULL );
```

- lpszClassName - This is a string that describes the "window class" that you want to use. The window class tells Windows if this window will be a frame, or a button, or a scrollbar, or whatever. You can also define your own classes. This value can not be changed once the window has been created. We pass in NULL to use the default window class for a CFrameWnd.
- lpszWindowName - This is the name of the window. If the window has a title bar, it will display this string.
- dwStyle - This is probably the most important parameter in defining how your window will look. But don't worry, you can always change the styles once the window is created. We'll play with some different styles later on. The styles we use create a basic top level (popup) window with a resizable border.
- rect - This defines the size and position of the window on the screen.
- pParentWnd - This is the parent of the window you are creating. Since we are creating a top level window, we have no parent.
- lpszMenuName - This tells the application which resource to use for the menu. If you have a menu resource defined you can pass in "#128", where 128 is the resource id of your menu. (128 should be the resource id of the menu in this particular example.)
- dwExStyle - These are extended styles and are very useful in further controlling the appearance of your window. You can set these at any time after the window has been created.

- pContext - This is used in the Doc/View architecture. Just pass in NULL for this parameter.

That's about it. Once the window is created, it uses a function inside MFC for the message handling, so you don't need to do anything there. In the next section, we'll play around with this window, so you can customize it to suit your needs.

Creating a Simple MFC Program

Creating a New Project

We want to create a new project based on SimpleMFC. We can either repeat the process described above (not my favorite - I won't remember all of those steps), or we can copy the existing one. Copy the entire directory for SimpleMFC into its parent directory. You should get a new directory with the name "Copy of SimpleMFC". Change the name of this directory to "SimpleExtMFC". Delete any object files and executables you may have copied (or just delete the "Debug" and "Release" directories if they exist).

Change the names of all of the files containing SimpleMFC to SimpleExtMFC. The easiest way to do this is in a DOS prompt. Set the prompt to the new directory and type "rename SimpleMFC*. * SimpleExtMFC*. *". Do the same in the "res" directory.

Do not open the project file yet. In Developer Studio, do an "Edit-Find In Files" for SimpleMFC. Be sure to set the search location to the new directory. Search all file types (*. *). Do not match whole word only. You will find matches in the following files:

MainFrm.cpp
ReadMe.txt
SimpleExtMFC.clw
SimpleExtMFC.cpp
SimpleExtMFC.dsp
SimpleExtMFC.dsw
SimpleExtMFC.h
SimpleExtMFC.plg
SimpleExtMFC.rc
StdAfx.cpp
SimpleExtMFC.rc2

Replace all instances of SimpleMFC with SimpleExtMFC in these files. (You can actually delete SimpleExtMFC.clw. This is the classwizard file, and will be rebuilt if you delete it.) Open each file and use the Replace menu to replace all instances. (Hint: if you press F3, the first occurrence of SimpleMFC will become highlighted, which makes using the Find and Replace dialog much easier.) Save all of the files and open your brand new project.

Creating a Simple MFC Program

Adding a Popup Menu

One of the most useful controls in modern applications is the popup menu on right-clicking the mouse. We are going to add a popup menu to our application.

First, we need to create a menu resource. Select Insert-Resource from the menu, select Menu, and press New. Right click on the IDR_MENU1 label and select Properties (see how useful that was). Change the name to IDR_POPUPMENU.

You should also have an empty menu available for editing. Normal menu bars lay out their options horizontally, left to right. Popup menus lay out their components vertically, top to bottom. We need to create a menu pane for our popup menu. In the first menu cell type the label "Popup Top" and press Enter. Select "Popup Top" and our menu pane will appear below it. Type "&Close" into the pane. Enter "ID_POPUP_CLOSE" as the ID. Save your resource definitions. We have just created the resource definition for our popup menu. Now we have to make our application open it.

Open ClassWizard (Ctrl-W or View-ClassWizard). Rebuild the database if asked. If a dialog pops up that asks you to associate a class with the new menu resource, you can associate the CMainFrame class with it. Verify that CMainFrame is selected as the class. We want to display the popup menu on a right mouse button up event, so locate WM_RBUTTONDOWN in the Messages list. Double click on it to add a message handler. Double click on the new method to edit it. Edit OnRButtonDown to look like this:

```
void CMainFrame::OnRButtonDown(UINT nFlags, CPoint point)
{
    CMenu popupMenu;
    popupMenu.LoadMenu(IDR_POPUPMENU);
    CMenu* subMenu = popupMenu.GetSubMenu(0);
    ClientToScreen(&point);
    subMenu->TrackPopupMenu(0, point.x, point.y,
        AfxGetMainWnd(), NULL);

    CFrameWnd::OnRButtonDown(nFlags, point);
}
```

LoadMenu loads the menu resource. GetSubMenu gets the particular popup menu we are using to display the Close menu item. ClientToScreen translates the cursor's screen position to the window position. Finally, TrackPopupMenu actually opens the menu. Simple, right? Go ahead and run the program. You'll notice that our option is greyed out in the popup menu. We still need to hook it up to do something. Close the program and go back into ClassWizard.

Verify that CMainFrame is still selected as your class. Under Object IDs, select ID_POPUP_CLOSE. Double click on COMMAND under Messages. This will create the member function OnPopupClose (don't change the name). Edit OnPopupClose to look like this:

```
void CMainFrame::OnPopupClose()
{
    PostMessage(WM_CLOSE);
}
```

Now, when the Close option is selected, we will send a WM_CLOSE message to the application ourselves. Compile and run the program. The popup menu should be fully functional.

Creating a Simple MFC Program

Moving the Window

Windows generally lets you move a window by dragging it around by the title bar. We don't have a title bar, so we need another way to move the window. We could just add a "Move" item to our popup menu and post a WM_ENTERSIZEMOVE event in the same way we posted WM_CLOSE, but I think I'd like the user to be able to drag the window around the screen.

The first part is easy. Add another menu item called "Move" to our popup menu under the "Close" menu item. Create a function to handle it called OnPopupMove. Edit OnPopupMove to look like this:

```
void CMainFrame::OnPopupMove()
{
    PostMessage(WM_SYSCOMMAND, SC_MOVE);
}
```

You can use WM_SYSCOMMAND to trigger events from the system menu. In fact, why don't we change OnPopupClose to use the WM_SYSCOMMAND as well. Change "PostMessage(WM_CLOSE)" to "PostMessage(WM_SYSCOMMAND, SC_CLOSE)". There isn't any change in functionality, but it helps keep things consistent.

Notice that when we select Move, initially we can move the window with the arrow keys, but not with the mouse. The same thing would happen if we tried to post an SC_MOVE event on mouse down. Windows doesn't provide us access to the code to move things, so we will have to write it ourselves. We want to start moving on a left mouse button down, move on a mouse move, and stop moving on a left mouse button up. First, add functions to handle ON_WM_LBUTTONDOWN, ON_WM_MOUSEMOVE, and ON_WM_LBUTTONUP.

To move the window, we first need to know when the mouse was pressed inside of it. Add the following member variable to CMainFrame:

```
BOOL m_bMouseDown;
```

Initialize it to FALSE in the constructor, set it to TRUE in OnLButtonDown, and set it to FALSE in OnLButtonUp. Add an if clause on it in OnMouseMove. This isn't good enough to tell when the mouse is pressed. It is possible for the button to be pressed while the cursor is inside of the window and released while the cursor is outside of the window. In that case we would never receive the mouse release and would continue moving the window. We need to capture the mouse so that we always get mouse events - even if the mouse is not in the window.

Add "SetCapture()" to OnLButtonDown and "ReleaseCapture()" to OnLButtonUp. During OnMouseMove, we want to move the window. We will need to know where in the window the mouse was pressed, and move the window by the distance between it and the current position. Add The following member variable to CMainFrame:

```
CPoint m_mouseDownPoint;
```

Set the variable to the point parameter in OnLButtonDown. In OnMouseMove, check that the mouse is down. Then find the size difference between the "mouse down point" and the new point. Get the window rectangle. Subtract the rectangle from the size and move the window to the new position.

That's it. We now have a window that we can drag around with the mouse or move via the menu using the arrow keys. The final code for OnLButtonDown, OnMouseMove, and OnLButtonUp follows:

```
void CMainFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_bMouseDown = TRUE;
    SetCapture();
    m_mouseDownPoint = point;

    CFrameWnd::OnLButtonDown(nFlags, point);
}
```

```
void CMainFrame::OnMouseMove(UINT nFlags, CPoint point)
{
    if( m_bMouseDown )
    {
        CSize s = m_mouseDownPoint - point;
        CRect r;
        GetWindowRect(&r);
        r = s - &r;
        MoveWindow(r);
    }

    CFrameWnd::OnMouseMove(nFlags, point);
}
```

```
void CMainFrame::OnLButtonUp(UINT nFlags, CPoint point)
{
    m_bMouseDown = FALSE;
    ReleaseCapture();

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

Creating a Simple MFC Program

More Menu Fun

We've added a few of the items from the system menu. Let's go ahead and add the rest: Restore, Size, Minimize, and Maximize. First, add menu options for the four items. Second, in the ClassWizard, add member functions to

handle each option. Third, post the appropriate WM_SYSCOMMAND event for each function (here's a hint: SC_RESTORE, SC_SIZE, SC_MINIMIZE, SC_MAXIMIZE). Run the program. Try out the menu items.

Notice that all of the options are always enabled. When you are maximized, you can still select maximize even though it doesn't do anything. We want to disable some of the menu items. Use the UPDATE_COMMAND_UI message map to do this. Create an UPDATE_COMMAND_UI handler for Move, Maximize, Minimize, Restore and Size. These functions pass in a CCmdUI object. Call Enable() to enable or disable the menu option for that function (you can also use CCmdUI to set checks and text).

We want to disable Move and Size when the application is not maximized:

```
pCmdUI->Enable(!IsIconic()&&!IsZoomed());
```

We want to disable Maximize when the application is maximized:

```
pCmdUI->Enable(!IsZoomed());
```

We want to disable Minimize when the application is minimized:

```
pCmdUI->Enable(!IsIconic());
```

We want to disable Restore when the application is not maximized or minimized:

```
pCmdUI->Enable(IsIconic()||IsZoomed());
```

Now when we use the menu, the options should be enabled appropriately.

We have one more thing to do. Notice that you can still move the window when the application is maximized. We want to disable this. Modify OnMouseMove to look like the following:

```
void CMainFrame::OnMouseMove(UINT nFlags, CPoint point)
{
    if( ( m_bMouseDown )&&( !IsZoomed() ) )
    {
        CSize s = m_mouseDownPoint - point;
        CRect r;
        GetWindowRect(&r);
        r = s - &r;
        MoveWindow(r);
    }
}
```

```
CFrameWnd::OnMouseMove(nFlags, point);  
}
```

This will prevent us from moving the window manually while it is maximized. Following is the code added for this section:

```
void CMainFrame::OnPopuptopRestore()  
{  
    PostMessage(WM_SYSCOMMAND, SC_RESTORE);  
}
```

```
void CMainFrame::OnPopuptopSize()  
{  
    PostMessage(WM_SYSCOMMAND, SC_SIZE);  
}
```

```
void CMainFrame::OnPopuptopMinimize()  
{  
    PostMessage(WM_SYSCOMMAND, SC_MINIMIZE);  
}
```

```
void CMainFrame::OnPopuptopMaximize()  
{  
    PostMessage(WM_SYSCOMMAND, SC_MAXIMIZE);  
}
```

```
void CMainFrame::OnUpdatePopupMove(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsIconic()&&!IsZoomed());  
}
```

```
void CMainFrame::OnUpdatePopuptopMaximize(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsZoomed());  
}
```

```
void CMainFrame::OnUpdatePopuptopMinimize(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsIconic());  
}
```

```
void CMainFrame::OnUpdatePopuptopRestore(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(IsIconic()||IsZoomed());  
}
```

```
void CMainFrame::OnUpdatePopuptopSize(CCmdUI* pCmdUI)
```

```
{  
    pCmdUI->Enable(!IsIconic()&&!IsZoomed());  
}
```

Creating a Simple MFC Program

Changing Your Style

The last part of this introduction to a simple MFC application deals with changing the style of your window.

When the window is maximized, I'd like to get rid of the borders (after all, if I'm dealing with graphics I want all the screen space I can get). We will remove the `WS_THICKFRAME` style from the window on `WM_MAXIMIZE` and add it back on `WM_RESTORE`.

Add `"ModifyStyle(WS_THICKFRAME, 0, 0)"` to `OnPopuptopMaximize` and `"ModifyStyle(0, WS_THICKFRAME, 0)"` to `OnPopuptopRestore`. The thick borders will be removed when you maximize the window.

There is still some border left. This is the 3D look that Windows provides. I want to remove this as well. The 3D look is controlled by the Extended Style `WS_EX_CLIENTEDGE`. Add `"ModifyStyleEx (WS_EX_CLIENTEDGE, 0, 0)"` to `OnPopuptopMaximize` and `"ModifyStyleEx (0, WS_EX_CLIENTEDGE, 0)"` to `OnPopuptopRestore`.

Just for fun, let's add a real System Menu to our window. This will let us right-click on the icon in the task bar. Add `"ModifyStyle(0, WS_SYSMENU, 0)"` to the constructor after `Create` (we could just add it into `Create`, but what fun would that be?).

Using `ModifyStyle` and `ModifyStyleEx` you can change the appearance of you window in all sorts of ways - add/remove menus, scrollbars, make the window stay on top, give it a title bar, among other things. Play around with it. Note that you will have to repaint your window after the style changes. The maximize and restore do it for us here.

Conclusion

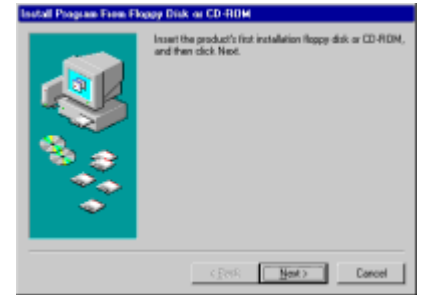
I hope that at this point you find yourself a little more familiar with how windows work in Windows. I've provided the source code with the article. Until next time, have fun.

Creating Wizards in MFC Applications

Introduction

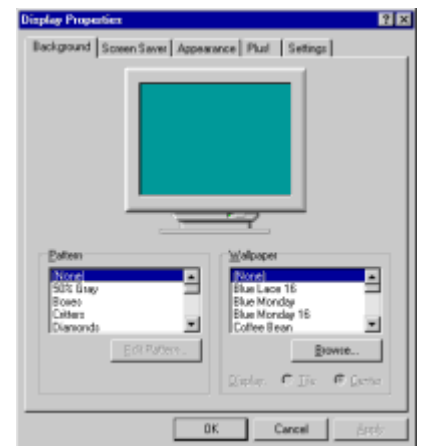
One of the most useful GUI concepts is the wizard. Allowing the application to guide a user through the process of completing a task makes even most complicated actions a breeze. In many ways wizards have revolutionized the modern Windows environment.

The purpose of a wizard is to collect and organize the user interface needed to complete a specific task. This is accomplished by creating multiple "page" dialogs that the user steps through. It is much easier for a user to navigate through a set of dialogs using a wizard than it is to move through a collection of individual modal dialogs. Also by breaking the user interface up over multiple pages, functionally related items can be more easily grouped, and the user is never overwhelmed by a large number of settings.



An example of a Wizard

In the MFC environment, wizards are extremely easy to implement. MFC provides the CPropertySheet and CPropertyPage classes for building tabbed dialogs like the one shown below. Wizards are implemented in the same fashion as tabbed dialogs, with only a few minor modifications.



Generic tabbed dialog

To understand Wizards, it is helpful to understand tabbed dialogs, or property sheets as known formally. Let's therefore start by looking at the implementation for a generic property sheet dialog.

Using the CPropertySheet Class

The CPropertySheet class is used to manage a group of property pages. Creating and displaying a modal property sheet dialog is simple using this class. Creating and displaying a property sheet is done the following way:

```
void CMyApp::OnSomeCommand()
{
    // Create the property sheet
    CPropertySheet propSheet( "Information" );

    // Create and initialize page one
    CPage1 page1;
    page1.m_name = "Name";
```

```

page1.m_phone = "Phone";
page1.m_address = "Address";

// Create and initialize page two
CPage2 page2;
page2.m_city = "City";
page2.m_state = "State";

// Add the pages to the sheet
propSheet.AddPage( &page1 );
propSheet.AddPage( &page2 );

if( propSheet.DoModal() == IDOK )
{
    TRACE( "Name: %s\nPhone: %s\nAddress: %s\n",
        (LPCSTR)page1.m_name,
        (LPCSTR)page1.m_phone,
        (LPCSTR)page1.m_address );

    TRACE( "City: %s\nState: %s\n",
        (LPCSTR)page2.m_city,
        (LPCSTR)page2.m_state );
}
}

```

The code first creates a property sheet object, passing the constructor the title of the property sheet. Then it creates and initializes an object for each property page. DDX takes care of initializing the controls on the page with the values of the member variables. Each page is added to the property sheet with `AddPage`. As with normal dialogs, `DoModal` is used to display the property sheet dialog and will not return until the user selects either the OK or Cancel buttons. Now that we have created a simple property sheet, we will explore how to better use these two classes. The property sheet above was created in two steps. First, the property sheet object was created and then its pages were created and added. Another, more object-oriented approach to creating a property sheet is to create a new class derived from `CPropertySheet`. The new class would contain a member variable for each property page and the constructor for the class would call `AddPage` for each page. You can use the ClassWizard to derive a class from `CPropertySheet` and then modify the new class as necessary.

```

class CMyPropertySheet : public CPropertySheet
{
public:
    CPage1 m_page1;
    CPage2 m_page2;
    CMyPropertySheet();
};

CMyPropertySheet::CMyPropertySheet() : CPropertySheet( "Information" )
{
    AddPage( &m_page1 );
}

```

```
    AddPage( &m_page2 );  
}
```

The main reason to derive a class from CPropertySheet is to enhance it. With your own class you can add buttons or modify the default buttons. Or you can create a modeless property sheet dialog. If you wish to create a modeless property sheet dialog, you can use the class's Create member function. In this case, you must create your own CPropertySheet-derived class because the default buttons, OK, Cancel, and Apply are not created for modeless property sheets. You must also provide a way in the new class to close and destroy the modeless property sheet dialog. This is the purpose of EndDialog. This function is used to destroy the property sheet dialog when OK, Cancel or Close is selected.

There are also several page management functions in the CPropertySheet class. RemovePage performs the opposite of AddPage and removes the specified page from the property sheet. Only the property page's window is destroyed. The actual CPropertyPage-derived object is not destroyed until its property sheet is. GetPage will return a pointer to the page specified by an index between 0 and the value of GetPageCount. These functions can be used to iterate through the property sheet's pages.

As with normal dialogs, you override the virtual functions OnCancel and OnOK to handle the Cancel and OK buttons. The OnCancel function is called when the Cancel button is selected (it will be labeled Close instead of Cancel if it has been renamed with CancelToClose). The OnOK function is called for two different actions. It's called when the user chooses either the OK or Apply button. The difference is that the Apply button does not call EndDialog to dismiss the property sheet. If you need to handle the Apply button in a separate function, you can manually provide a message map entry for ID_APPLY_NOW in the page's parent CPropertySheet-derived class.

Two other virtual functions can be overridden to allow more control over the CPropertyPage class. OnSetActive is called when the page is chosen by the user and becomes the active page. The default action is to create the window for the page, if not previously created, and to make the page the active page. You can override this function to perform tasks that need to be done when a page is activated, such as custom initialization. Note that the controls for a property page are not created until the page itself is created, so make sure you call the base class OnSetActive to create the page before referencing any of the page's controls. OnKillActive is the opposite of OnSetActive and is called when the page is no longer to be the active page. The property page's OnOK function is only called if this function returns successfully. The default action is to call the DDX function UpdateData to copy settings from the controls in the property page to the member variables of the property page. If the data was not updated successfully because of a DDV error, the page retains focus. You can override this function to perform special data validation tasks that should be done before the active page loses focus. Note that the data is transferred from the controls to the member variables in OnKillActive, not in OnOK. Note that all these functions are page-dependent. That is, each property page class has its own OnOK, OnCancel, OnSetActive, and OnKillActive.

Some other useful functions are CancelToClose and SetModified. You can use the CancelToClose function to notify the user that he has made an unrecoverable change to the data in a page. This function will change the text of the Cancel button to read Close. This alerts the user that they have made a permanent change that cannot be cancelled. Note that the CancelToClose member function does nothing in a modeless property sheet because a modeless property sheet does not have a Cancel button by default.

The SetModified function is used to enable or disable the Apply button. Each page has a flag that marks the page as being "dirty." When the data for a page has been changed you can call SetModified with TRUE to enable the button. The Apply button will become disabled again only when none of the property pages is "dirty." Note that each page has its own "dirty" flag independent of the other pages.

Building a Wizard

Now let's look at creating a wizard using the CPropertySheet class. Let's start by creating an SDI project with the proper dialog resources and their associated classes. The following steps show how to set up the basic components for our sample wizard.

1. Create a standard AppWizard SDI application.
2. In the resource editor add an item called "Wizard" to the application's menu.
3. Using the ClassWizard create a command handler in CMainFrame for the menu item, call the handler function "OnWizard".
4. Use the ClassWizard to add a new class to the project derived from CPropertySheet, label the class CWizardDlg.
5. Add three new dialog resources and place an edit on each dialog. Give each dialog the ID IDD_PAGE1, IDD_PAGE2 etc. Then change the caption of each dialog to "Page 1" "Page 2" etc. (the caption from each page will be used as the caption for the wizard dialog while the page is active).
6. Add a static text object to each dialog and enter the text "Page 1", "Page 2" etc.
7. Create a class derived from CPropertyPage for each dialog. To do this, open the ClassWizard click the Add button. Name the class CPage1, CPage2 etc. Select CPropertyPage as the base class. Then select the appropriate dialog resource.

Now that the underlying architecture is place we can wire the wizard into the application. First add the header files for each of the wizard pages to WizardDlg.h:

```
#include "Page1.h"  
#include "Page2.h"  
#include "Page3.h"
```

Next add the following code to the CWizardDlg class definition:

```
public:  
    CPage1 m_Page1;  
    CPage2 m_Page2;  
    CPage3 m_Page3;
```

Add the following code to both CWizardDlg constructors:

```
AddPage( &m_Page1 );  
AddPage( &m_Page2 );  
AddPage( &m_Page3 );  
  
SetWizardMode();
```

This code adds each of the pages to the `CWizardDlg` instance. The call to `SetWizardMode()` changes the style of the dialog from the standard tabbed dialog to a wizard style.

Next the dialog needs to be wired into the command handler for the menu. Include the `CWizardDlg` header file in `MainFrm.cpp` and then add the following code to the `OnWizard` function:

```
CWizardDlg wizardDlg( "Test Wizard" );  
wizardDlg.DoModal();
```

This creates an instance of the wizard and passes in the title (which will be ignored, the wizard will instead use the captions from each page for the dialog title), then calls the `DoModal` function to display the dialog. Now we can test the basic functionality of the dialog.

Compile and run the application. After selecting the "Wizard" from the menu bar the dialog will appear displaying the first page. The Next and Back buttons function however they do not gray when on the first or last page. Also the Next button should convert to a Finish button once the last page is reached. To provide these features the `CPropertyPage` classes allow you to handle the following messages and then manually change the button states.

<code>OnSetActive</code>	(Called when page becomes active)
<code>OnKillActive</code>	(Called when page is no longer active)
<code>OnWizardNext</code>	(Called when wizard Next button is clicked)
<code>OnWizardBack</code>	(Called when wizard Back button is clicked)
<code>OnWizardFinish</code>	(Called when wizard Finish button is clicked)

The `OnSetActive()` message plays the main role in adding the button functionality. Once a page is notified that it is active, it must then call `SetWizardButtons()` in the parent `CWizardDlg` object to change the button states. Using the ClassWizard, add a `OnSetActive()` handler to each of the `CPage` classes.

In the `CPage1 OnSetActive()` function add the following code:

```
CWizardDlg* pParent = (CWizardDlg*)GetParent();  
ASSERT_KINDOF(CWizardDlg, pParent);  
  
pParent->SetWizardButtons(PSWIZB_NEXT);
```

This will get a pointer to the parent `CWizardDlg` and the call `SetWizardButtons()` to disable the back button. The following flags can be passed as parameters to `SetWizardButtons()`:

```
PSWIZB_NEXT  
PSWIZB_BACK  
PSWIZB_FINISH  
PSWIZB_DISABLEDFINISH
```

See the [SetWizardButtons\(\) documentation](#) for more information on using these flags.

In CPage2 OnSetActive() add:

```
CWizardDlg* pParent = (CWizardDlg*)GetParent();  
ASSERT_KINDOF(CWizardDlg, pParent);  
  
pParent->SetWizardButtons(PSWIZB_BACK | PSWIZB_NEXT);
```

In CPage3 OnSetActive() add:

```
CWizardDlg* pParent = (CWizardDlg*)GetParent();  
ASSERT_KINDOF(CWizardDlg, pParent);  
  
pParent->SetWizardButtons(PSWIZB_BACK | PSWIZB_FINISH);
```

Next add an include statement for WizardDlg.h to each of the Page cpp files. Then recompile and run the application. On opening the wizard you should notice that the Back button is disabled. Once on the last page the Next button will convert to a Finish button. You now have functionally complete wizard.

The next step is to add a control and look at data exchange from the wizard. As was mentioned in the CPropertySheet intro each page in the wizard is treated as a separate dialog. To add a control open the Page1 dialog template and drag an edit control onto the dialog. Then open the ClassWizard's Member Variables tab for the CPage1 class and map the edit control to a CString. Label the string m_EditStr.

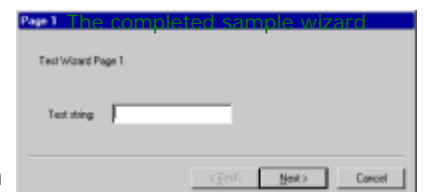
Now we need to change the OnWizard handler in CMainFrame so that it can determine whether the user clicked Cancel or Finish, and then display the contents of the edit. Change the OnWizard function so that it reads:

```
CWizardDlg wizardDlg( "Test Wizard" );  
  
if( wizardDlg.DoModal() == ID_WIZFINISH )  
{  
    TRACE1("Edit contained: %s", wizardDlg.m_Page1.m_EditStr );  
}
```

The if statement checks to see if the user clicked the Finish button (if Cancel was clicked the result would be IDCANCEL as in any other dialog). Then data from the edit is retrieved by calling wizardDlg.m_Page1.m_EditStr.

Conclusion

This should give you a basic understanding of wizards and their implementation



using MFC.

Building Database Applications with the CRecordset Class

At Interface Technologies, any database used in any sort of production setting is stored on a true SQL server on the network. However, we tend to develop initial database prototypes in Access before moving them to SQL. The reasons for doing this tend to be:

- Everyone can get to Access - you don't need passwords or a direct link to the SQL server (especially important if you are working on a portable on the road) to use an Access database.
- Access takes about 10 minutes to learn to use.
- You can change an Access database on the fly.

Once the initial database and client application have been developed, then it is fairly easy (a day or two of effort) to move the database across to the SQL server. The client application needs to change very little or not at all during the port.

To lay out the initial database, look at your problem and come up with a concept in your mind for the tables it needs. Then lay out the tables in Access by creating a new database file. For example, say you wanted to create a simple address list database in Access. Perhaps you have decided you would like to have a "People" table, an "Address" table and a "Zip code" table in the database. You would create the three new tables with the following fields:

People Table:

- PersonID, Number, Long Integer, Primary Key
- LastName, Text, 20
- FirstName, Text, 20
- Address, Number, Long Integer

Address Table:

- Address ID, Number, Long Integer, Primary Key
- Street, Text, 50
- ZipCode, Text, 10

ZipCode Table:

- Zip, Text, 10
- City, Text, 20
- State, Text, 2

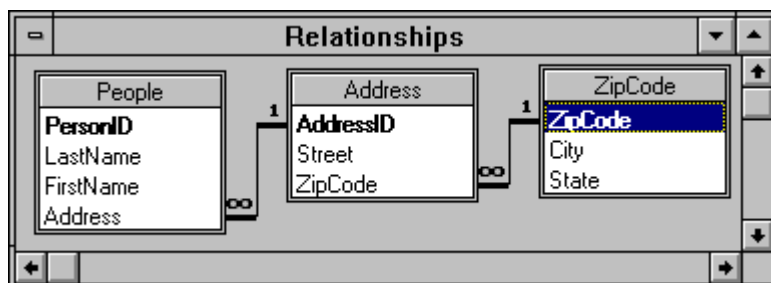
As an example, the People table would look like this in Access:

Table: People	
Field Name	Data Type
PersonID	Number
LastName	Text
FirstName	Text
Address	Number

Field Properties	
Field Size	Long Integer
Format	
Decimal Places	Auto
Input Mask	
Caption	
Default Value	0
Validation Rule	
Validation Text	
Required	No
Indexed	Yes (No Duplicates)

Note that all tables have a primary key. This is not a requirement, but most will (although any association table (e.g. - a table that associates multiple phone numbers to people) generally will not). Note that the primary key field is defined as a LONG INTEGER in the first two tables, and as TEXT in the last one. If the primary key is an integer ID, be sure to make it a LONG one.

Now you can create a relationship diagram, which will allow you to set up referential integrity constraints. You should definitely do this, because the database will then reject any additions you make that violate referential integrity. This can save a lot of database corruption headaches later on. Here's an appropriate diagram for this sample database:



Simply click on the primary key and drag it to the field that refers to it (the foreign key) in the second table. Click on the "Enforce Referential Integrity" check box in the dialog that appears to introduce referential integrity checks.

There is one other thing you will need to do. For each primary key that consists of a LONG INTEGER, you will need to set up a separate table that contains just one field and a single record. This table will be used to safely create new primary key numbers. You have to do this for two reasons:

- It is the only safe way we have found to increment primary keys to create new records
- If you ever want to distribute things to remote sites that are separated from the main SQL database, your will have to use this separate table technique anyway, so you might as well put it in place now.

Therefore, in the above database you would create two new tables, one called "NumPeople" and the other called "NumAddresses". They would both contain a single LONG INTEGER field called LastNumber (or something like that). Make that field the primary key. Then create one row in that table and set the field to the initial value (e.g.: 1).

It is our impression that, in Access, there is no way that you can make an "addition safe" database when multiple people are accessing a single database simultaneously (this statement is true only because the ODBC driver for Access currently does not support pessimistic locking). Once you move the database to SQL, this "NumPeople" approach will allow you to create an addition-safe database by using stored procedures.

A few things to keep in mind when creating Access databases:

1. Make sure that your table and column names do not contain any spaces
2. *Your primary and foreign keys should have different column names*
3. Access has reserved words. Therefore, you cannot name a table or column "Money", for example, because Access reserves this word.

Setting up ODBC

Once you have set up your database in Access (and one of the nicest things about Access is you set up just a few tables initially and easily expand things over time), you need to create an ODBC data source for it. Open the Control Panel. Then open the ODBC administration tool. You will see a dialog showing all current ODBC data sources on the system. Click the Add button to add a new data source.

You will see a dialog showing all of the drivers available. If the Access driver is not in the list, you will need to install it using the VC++ install program. Do a customized installation and add the Access driver.

Choose the Access driver. In the next dialog, name the data source. Typical names we have used are FHI3 (for version 3 of the FHI database) and ITIDB1 (for version 1 of the ITI client database). Add a description if you want. Then click the Select button and find the MDB file for your database. Click OK and your ODBC data source will be created.

Conclusion

That's all there is to it. You now have a database prototype for your application, and VC++ will be able to find and understand your new database because of the ODBC data source. You will have to create an ODBC data source on every machine where you want to use the client application that you develop. When you change over to an SQL database, you will need to point the ODBC data source at the SQL database instead of an Access file.

Building Database Applications with the CRecordset Class

by Marshall Brain

Now that you have created a database and an ODBC data source to access it, you can create a VC++ client application to manipulate the database. You could use ODBC calls directly in this application, and this probably isn't a bad way to do things. However, it is significantly quicker and easier to use the CRecordset class to access the different tables in your database. At ITI we do not use the CRecordView class at all because we find it too constraining, but CRecordset is extremely flexible.

Rule 1

The first and most important general rule when using the CRecordset class is this: you want to create a new, distinct CRecordset class to handle every different way that you access a table. That means that EVERY table in the database will have its own CRecordset class handling the table, and then there will be another CRecordset

class for any and every different join that you do. Also, if for some reason there are cases where you want to be able to access a table and retrieve only one or two columns from it (perhaps to improve performance), then you will want to create a different CRecordset for that. This means a big program will have MANY different CRecordset classes. That is OK and completely normal.

ALWAYS create any CRecordset class using the ClassWizard. This makes the creation of these classes extremely easy. At ITI we always preface record sets with "RS", so typical CRecordset class names are CRSPeople, CRSAddresses, etc.

Rule 2

The second general rule is this: You should NEVER modify the code that the ClassWizard produces for a CRecordset object. Instead, we inherit from the CRecordset class that the ClassWizard produced and put any additions in this inherited file. We preface these file names with "RSA", so you end up with classes like CRSAPeople, CRSAddresses, etc. The reason you want to do this is because, if you change a database table, you can then easily regenerate the RS file with the ClassWizard to pick up the changes to the table. The RSA file, however, will require no changes. If you had made the changes directly to the RS file then you would lose them during the regeneration process.

In general, you want to place into the RSA file ALL manipulations that you do to a given database table. You do this so that you don't have table manipulation code floating all over your application. The only possible exception to this rule is adding data to the table, because in that case you sometimes end up passing 15 parameters and that is more trouble than it is worth. There is an example RSA file in tutorial 4.

Rule 3

The third general rule is this: You want to have CRecordset classes open only as they are needed. An instance of a CRecordset class seems to take up a lot of RAM, and opening a CRecordset is not difficult or time consuming. Therefore, you should open CRecordset instances when you need them and close them (or let them go out of scope naturally, and in that case they close automatically) when you are done.

You generally do not want to have any globally open CRecordset (or RSA) files in your application. We did try the approach of opening all CRecordset classes in InitInstance in the early going of one program we created because it seemed to take so long to open a CRecordset. We later found out that:

- The opening delays were caused by opening the database (see the next tutorial), and that if you open the database separately you can eliminate these delays
- A big program uses many, many record set classes, and trying to open all of them in InitInstance takes too long
- The many open instances have a big memory load

Therefore, we now open record sets only as we need them.

To use the CRecordset class in an application, the only thing that you have to do is insert the proper header file into the stdafx.h file. The AppWizard can do this for you, or you can do it yourself by including <afxdb.h> in stdafx.h.

The following sections show specific code techniques that we use in our applications.

Building Database Applications with the CRecordset Class

by Marshall Brain

If you look in a Visual C++ book, there are example pieces of code that look like this:

```
CRSPeople people;
people.Open();
int x = people.GetRecordCount();
...
```

The problem with this particular approach is that there is a rather substantial delay (say 5 seconds) to execute these three lines of code. That delay is not caused by opening the table, but instead by opening the ODBC connection to the database, which happens behind the scenes in the constructor for the CRecordset class. Therefore, if you can open the database connection once, globally, and then use that connection throughout the program, you speed things up tremendously. Here is a typical piece of code that you would find in InitInstance to open the database globally:

```
try
{
    db.Open("ITIDB1;UID=iti;PWD=iti", FALSE,
    FALSE, "ODBC;", FALSE);
    db.SetSynchronousMode(TRUE);
}
catch(CDBException *e)
{
    AfxMessageBox(CString(
    "Cannot open system database.\n")
    + e->m_strError, MB_ICONSTOP);
    throw;
}
```

The following declaration would be found in the application class's header file as a public member:

```
CDatabase db;
```

Now that the database has been globally opened once, you use it throughout your program. For example:

```
CltidbApp *app=(CltidbApp *)AfxGetApp();
CRSPeople people(&(app->db));
people.Open(CRecordset::dynaset);
int x = people.GetRecordCount();
...
```

You could also argue that the overridden constructor in the RSA file could handle the database connection, and I believe that is valid. That should save you from having to create an app variable everywhere.

Building Database Applications with the CRecordset Class RSA Files

For every database table you create, you will use the ClassWizard to create a CRecordset class. We prefix these files with RS, for example CRSPeople. This code is all automatically generated, and you will never touch it. You will create another RS file for the associated "Num" table in the database. For example, you will likely have a CRSNumPeople table.

You should then create an RSA file for the People table. This file should encapsulate all common database transactions on that table. This file you generate by hand, and it would look something like this (here the file is called CRSAClient instead of CRSAPeople - same idea)

(header):

```
// rsaclien.h : header file
//
// do not include anywhere else
#include "rsclient.h"
////////////////////////////////////
// CRSAClients recordset
class CRSAClients : public CRSClients
{
public:
    CRSAClients(CDatabase* pDatabase = NULL);
    void LoadCB(class CComboBox &cb);
    void LoadLB(class CListBox &lb);
    CString GetClientName(long clientID);
    long GetNextClientID();
    DECLARE_DYNAMIC(CRSAClients)

// Implementation
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
};
```

(code):

```
// rsaclien.cpp : implementation file
//
#include "stdafx.h"
#include "itidb.h"
#include "rsnumpeo.h"
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
```

```
#endif
```

```
////////////////////////////////////
```

```
// CRSAClients
```

```
IMPLEMENT_DYNAMIC(CRSAClients, CRSClients)
```

```
CRSAClients::CRSAClients(CDatabase* pdb)  
    : CRSClients(pdb)
```

```
{  
}
```

```
void CRSAClients::LoadCB(class CComboBox &cb)
```

```
{  
    CltidbApp *app=(CltidbApp *)AfxGetApp();  
    int index;  
    m_strSort="LastName ASC, FirstName ASC";  
    m_strFilter="";  
    try  
    {  
        Requery();  
        cb.ResetContent();  
        while (!IsEOF())  
        {  
            index=cb.AddString(m_LastName + ", " + m_FirstName);  
            cb.SetItemData(index, m_PersonID);  
            MoveNext();  
        }  
    }  
    catch(CDBException *e)  
    {  
        AfxMessageBox(e->m_strError, MB_OK | MB_ICONEXCLAMATION);  
        e->Delete();  
    }  
}
```

```
void CRSAClients::LoadLB(class CListBox &lb)
```

```
{  
    CltidbApp *app=(CltidbApp *)AfxGetApp();  
    int index;  
    m_strSort="Name ASC";  
    m_strFilter="";  
    try  
    {  
        Requery();  
        lb.ResetContent();  
        while (!IsEOF())  
        {
```



```

        index=lb.AddString(m_LastName);
        lb.SetItemData(index, m_PersonID);
        MoveNext();
    }
}
catch(CDBException *e)
{
    AfxMessageBox(e->m_strError, MB_OK | MB_ICONEXCLAMATION);
    e->Delete();
}
}

```

CString CRSAClients::GetClientName(long clientID)

```

{
    m_strSort="";
    m_strFilter.Format("(ClientID = %d)", clientID);
    try
    {
        Requery();
        if (!IsEOF())
        {
            return m_LastName + m_FirstName;
        }
        else
        {
            CString st;
            st.Format( "Client %d not found.", clientID );
            AfxMessageBox( st, MB_OK | MB_ICONEXCLAMATION);
        }
    }
    catch(CDBException *e)
    {
        AfxMessageBox(e->m_strError, MB_OK | MB_ICONEXCLAMATION);
        e->Delete();
    }
    return "";
}

```

long CRSAClients::GetNextClientID()

```

{
    long personID=-1;
    CltidbApp *app=(CltidbApp *)AfxGetApp();
    CRSNumPeople numPeople(&(app->db));
    numPeople.Open(CRecordset::dynaset);
    //    numPeople.SetLockingMode(pessimistic); cannot do this in Access
    try
    {
        if (IsBOF())

```

```

    {
        AfxMessageBox("Database may have a problem, call ITI.",
            MB_OK | MB_ICONEXCLAMATION);
        personID = -1;
    }
    else
    {
        numPeople.Edit();
        numPeople.m_LastNumber = numPeople.m_LastNumber + 1;
        personID = numPeople.m_LastNumber;
        numPeople.Update();
    }
}
catch(CDBException *e)
{
    AfxMessageBox(e->m_strError, MB_OK | MB_ICONEXCLAMATION);
    e->Delete();
    personID = -1;
}
return personID;
}

```

```

//////////
// CRSClients diagnostics
#ifdef _DEBUG

```

```

void CRSClients::AssertValid() const
{
    CRSClients::AssertValid();
}

```

```

void CRSClients::Dump(CDumpContext& dc) const
{
    CRSClients::Dump(dc);
}

```

```

#endif //_DEBUG

```

There are four useful functions in this class. All of these tend to be so useful that you will want to implement them in every RSA file you create:

```

void LoadCB(class CComboBox &cb);
void LoadLB(class CListBox &lb);
CString GetClientName(long clientID);
long GetNextClientID();

```

The first two load a Combo Box and List Box respectively. Note that they use the "item data" (SetItemData) part of the list to hold the ID of each database record, and that can be extremely useful. The next function, given an ID, returns the name. The next one creates the next ID value, which is useful when you want to add a new record to the table. It uses the NumClients table to generate the new record ID.

Note that all of these functions catch database exceptions and handle them appropriately. You will want to do the same.

When filling the list boxes or combo boxes, it is also common to accept a limiting parameter, as shown here:

```
void CRSAAddress::LoadCB(class CComboBox &cb, long companyID)
{
    CltidbApp *app=(CltidbApp *)AfxGetApp();
    int index;
    m_strSort="Street ASC";
    m_strFilter.Format("(Company = %d)", companyID);
    try
    {
        Requery();
        cb.ResetContent();
        while (!IsEOF())
        {
            index=cb.AddString(m_Street + ", " + m_Zip);
            cb.SetItemData(index, m_AddressID);
            MoveNext();
        }
    }
    catch(CDBException *e)
    {
        AfxMessageBox(e->m_strError, MB_OK | MB_ICONEXCLAMATION);
        e->Delete();
    }
}
```

The second parameter is used to set the filter string, and limits the number of records that get added to the list box. You can parameterize the filter string as described in the MFC Encyclopedia, or simply do it by handle as shown here.

Building Database Applications with the CRecordset Class Database Dialogs

by Marshall Brain

It is good to place ALL of the code for manipulating the database either in a RSA class or into the dialog class that manipulates a given table. That way, the dialog class is completely self-contained. Then a function like OnAddClient, which is responsible for handling the menu option that adds a new client to the database, would have just 2 lines:

```

CClientDlg dlg;
dlg.DoModal();

```

There is much to be said for this approach. The dialog class needs a minimum of two functions in order to be self-contained: OnInitDialog and OnOK. The first sets the dialog up so it appears properly when it comes onto the screen, and the second handles data verification and saving when the user presses the OK button.

The Phone Number dialog in a typical client database might use this approach. The dialog looks like this:

This dialog lets the user type in a phone number and a comment. In addition, the user can select the phone number type from the combo box. To pull up this dialog, the code is extremely simple and looks like this:

```

CPhoneDlg dlg;
dlg.m_ClientID = ...;
dlg.DoModal();

```

The m_ClientID field was added by hand to the dialog class (you could just as easily set it using a constructor parameter). It is used to pass the current client ID to the dialog, because the dialog will need this ID in order to add records to the phone number table. Otherwise, the dialog is completely self-contained. Here is the header for the dialog class:

```

// phonedlg.h : header file
//
// CPhoneDlg dialog
class CPhoneDlg : public CDialog
{
    // Construction
public:
    CPhoneDlg(CWnd* pParent = NULL);    // standard constructor

    // Dialog Data
   //{{AFX_DATA(CPhoneDlg)
    enum { IDD = IDD_PHONE_ADD };
    CComboBox    m_PhoneType;

```

```

        CString    m_PhoneNumber;
        CString    m_Comment;
    //}}AFX_DATA
    long m_ClientID;

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CPhoneDlg)
protected:
    // DDX/DDV support
    virtual void DoDataExchange(CDataExchange* pDX);
//}}AFX_VIRTUAL

// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CPhoneDlg)
    virtual void OnOK();
    virtual BOOL OnInitDialog();
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

```

Here is the code file:

```

// phonedlg.cpp : implementation file
//
#include "stdafx.h"
#include "itidb.h"
#include "phonedlg.h"
#include "rsaphntp.h"
#include "rsaphnnm.h"
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////
// CPhoneDlg dialog
CPhoneDlg::CPhoneDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CPhoneDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CPhoneDlg)
    m_PhoneNumber = _T("");
    m_Comment = _T("");
    //}}AFX_DATA_INIT

```

```

        m_ClientID = -1;
    }
void CPhoneDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CPhoneDlg)
    DDX_Control(pDX, IDC_PHONE_TYPE, m_PhoneType);
    DDX_Text(pDX, IDC_PHONE_NUMBER, m_PhoneNumber);
    DDX_Text(pDX, IDC_COMMENT, m_Comment);
    DDV_MaxChars(pDX, m_Comment, 50);
   //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CPhoneDlg, CDialog)
   //{{AFX_MSG_MAP(CPhoneDlg)
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
////////////////////////////////////
// CPhoneDlg message handlers
void CPhoneDlg::OnOK()
{
    UpdateData(TRUE);
    long sel=m_PhoneType.GetCurSel();
    if (sel==LB_ERR)
    {
        AfxMessageBox("Please select a phone number type");
        return;
    }
    if (m_PhoneNumber.GetLength()==0)
    {
        AfxMessageBox("Please enter a phone number");
        return;
    }
    try
    {
        CltidbApp *app=(CltidbApp *)AfxGetApp();
        CRSAPhoneNumbers phoneNumbers(&(app->db));
        phoneNumbers.Open(CRecordset::dynaset);
        phoneNumbers.AddNew();
        phoneNumbers.m_Type = m_PhoneType.GetItemData(sel);
        phoneNumbers.m_Number = m_PhoneNumber;
        phoneNumbers.m_Comment = m_Comment;
        phoneNumbers.m_PhoneID = phoneNumbers.GetNextPhoneNumberID();
        phoneNumbers.m_Person = m_ClientID;
        phoneNumbers.Update();
    }
    catch(CDBException *e)
    {
        AfxMessageBox(e->m_strError, MB_OK | MB_ICONEXCLAMATION);
    }
}

```

```

        e->Delete();
        return;
    }
    CDialog::OnOK();
}
BOOL CPhoneDlg::OnInitDialog()
{
    ASSERT(m_ClientID >=0); //
    CltidbApp *app=(CltidbApp *)AfxGetApp();
    CRSAPhoneTypes phoneTypes(&(app->db));
    phoneTypes.Open(CRecordset::dynaset);
    CDialog::OnInitDialog();
    phoneTypes.LoadCB(m_PhoneType);

    return TRUE; // return TRUE unless you set the focus to a control
                // EXCEPTION: OCX Property Pages should return FALSE
}

```

The OnInitDialog function has to fill the Phone Type combo box. The OnOK function makes sure that the user has entered a phone number and selected a type for it. It then adds a record to the database. If addition fails the dialog recycles to let the user try again. It could be argued that the addition code should be placed in the RSA file, but then you end up with addition functions sometimes having 10 or 15 parameters. It is hard to say whether one approach is better than the other.

Note also how the try/catch blocks catch all database exceptions. The database therefore is able to display its errors directly to the user in a message box. This saves you having to perform all sorts of unweildy checks yourself, and is a big time saver.

Conclusion

You can see from these tutorials that the use of the CRecordset class is extremely straightforward. By making use of this class and the databases it supports in your own applications, you can create extremely robust applications with very large data capacities.

Building Database Applications with the CRecordset Class A User's Guide to the Different Types of Recordsets

by Marshall Brain

Accepting all the defaults when using Appwizard or Classwizard to generate a recordset is the logical way to get started using the CRecordset class. However, as you experiment with the results of doing this, you may find that you are not getting the performance or the behavior you want from your recordset. Fortunately, there is more to CRecordset than the wizard defaults! This article will briefly describe the different types of recordsets, some details of their use, and their advantages and disadvantages.

An important point to keep in mind is that Crecordset is ultimately just a C++ extension of ODBC. As such, it is limited by the design of ODBC. Furthermore, since ODBC is part implementation (the ODBC driver manager and the ODBC cursor library core components) and part specification, ODBC is itself limited by the ODBC driver in use. Acting as the bridge between a particular DBMS (be it Access, SQL Server, or Oracle) and the ODBC core

components, the driver determines through its design and implementation what subset of the ODBC specification's full range of functionality is actually available. So, keep in mind that given a poorly implemented driver some or all of the alternatives to the default CRecordset mentioned here may be beyond your grasp.

ODBC Cursor Types

The ODBC Programmer's Reference is the bible for ODBC purists. The information on ODBC cursor types presented here is simply a condensation of the comprehensive information available in the Programmer's Reference. For Visual C++ users the Programmer's Reference is available via the infoviewer under SDK->ODBC SDK 2.10.

ODBC specifies that four basic types of cursors can be supported. (A cursor in database terms is essentially a roving pointer to the current record of interest. Cursors rove through something called a result set—the collection of records returned by a particular query, generally a SELECT statement.) In its current implementation, CRecordset can be configured to use any one of these types (provided the ODBC driver in use supports the requested cursor type, of course).

CRecordset Type	ODBC Cursor Type	Characteristics
forwardOnly	Forward Only	scroll in forward direction only, not updatable
snapshot	Static	bi-directional scrolling, updatable, usually uses ODBC cursor library
dynaset	Keyset Driven	bi-directional scrolling, updatable, uses SQLSetPos for updates, inserts, and deletes, many drivers do not support
dynamic	Dynamic	bi-directional scrolling, updatable, uses SQLSetPos for updates, inserts, and deletes, most drivers do not support

Appwizard/Classwizard Support and Beyond

Appwizard and Classwizard only expose the snapshot and dynaset type recordset options. As you may have noted, snapshot is the default recordset type. If you want to use one of the other types of recordsets, you must manually override what the wizards give you. Two easy ways exist to do this:

1) The wizards initialize the CRecordset member variable `m_nDefaultType` to either snapshot or dynaset in the constructor of the CRecordset derived class they create for you. By simply changing the value that is assigned to this variable to one of the values in the first column of the above table (these values comprise an enumerated type), you can change the recordset type.

2) As its name implies, `m_nDefaultType` is the default type of the recordset. You can override that default on a per instance basis by specifying one of the enumerated recordset types in the call to your CRecordset derived class's `Open()` function (e.g. `m_pSet->Open(CRecordset::dynamic);`). If you explicitly specify the type in `Open()`, the `m_nDefaultType` setting is ignored.

The Role of CDatabase and the ODBC Cursor Library

As you may be aware, CRecordset is only part of the MFC database story. There is an underlying class, CDatabase, that plays an essential and often forgotten role. CDatabase is easily overlooked because you do not need to explicitly instantiate objects of type CDatabase. As mentioned in Part 3 of this series, when a CRecordset

derived class is opened MFC automatically instantiates and opens a CDatabase object to make the necessary connection to the datasource. Since such a CDatabase is created solely for use by a single recordset, the recordset assumes the responsibility for closing and destructing the CDatabase object when it is itself destructed.

The preceding description assumes that you have not explicitly specified that your recordset should use a separately created CDatabase for its connection. Instantiating a CDatabase object yourself allows you to reuse the connection to the database for a single recordset that you construct and destruct repeatedly or to share it amongst multiple recordsets (be forewarned that this can impose certain limitations when using options and operations that function at the database connection level such as transactions). Having declared and created your own CDatabase, you can force your recordsets to use that database by passing a pointer to it into the constructor of your recordsets or via the m_pDatabase member of the recordsets (assign the address of the CDatabase object to the m_pDatabase member of your recordset prior to opening the recordset or the recordset will implicitly create a new database and ignore your database). You additionally have the option of opening the database object yourself or letting the first recordset open do it for you (see below for issues regarding doing this).

While creating and opening your own CDatabase objects has definite advantages, it carries with it one definite disadvantage—since you are essentially prohibiting the MFC framework from working some of its magic when you open the database yourself, you can easily get yourself into a bind. Here is how: As mentioned in the above table, snapshots use something called the ODBC cursor library (they may not if you specify a read-only snapshot—depending upon the driver you use). The cursor library provides scrollability and other functionality that many drivers don't provide in their static cursors. This is all well and good so long as you are using a recordset type that can coexist with the cursor library. The problem is that neither dynaset nor dynamic recordsets can be used with the cursor library (you will receive a CDBException informing you that the "ODBC driver does not support dynasets" or "Dynamic cursors not supported by ODBC driver" depending on the recordset type you try to open). How this ties in with opening your own CDatabase objects is that it is at the CDatabase level that the cursor library is loaded or not. Specifically, if you are opening your own CDatabase object, you must be certain that the optional fifth parameter to CDatabase::Open() matches the type of recordset you are opening off that database. Specify no value for the optional fifth parameter or explicitly specify TRUE (i.e. load the cursor library) for most snapshots (some drivers may not require the cursor library for read-only snapshots—you'll have to experiment) and FALSE (i.e. don't load the cursor library) for forwardOnly, dynaset and dynamic recordsets. Additionally, if you are sharing a CDatabase object amongst multiple recordsets you must be sure all those recordsets that use the database are of a type compatible with the way the database was opened (this is true whether you explicitly open the database or let the first recordset Open() open it for you).

For CDatabase::OpenEx() users (an alternative to CDatabase::Open()), the CDatabase::useCursorLib bit in the dwOptions bitmask parameter plays the same role as the fifth parameter to CDatabase::Open()

Pros and Cons of Recordset Types and Other Words of Wisdom

So far we have looked at the details of the different types of recordsets but we haven't really spelled out when to use which and for what. Now we will look at some of the pros and cons of each recordset type:

Forward Only

This is the stripped down, no optional equipment installed, minimum overhead recordset type. If you want to populate a list box with all the names in an address book table stored in your database or you need to do a some kind of fast, sequential access to the result set, this is the way to go. Of course, don't expect to scroll backwards or do any updating. Also, CRecordset::MoveLast() can not be used because, while it may seem that moving to the

last record is a forward only operation, you must actually move past the last record in order to realize it is the last. As a result, a backwards move is actually required to position the cursor on the last record and that is forbidden.

Snapshot

This type of recordset is the default because it is one that most drivers support and it provides updatability (one or both with the help of the cursor library). It can also be a major source of poor performance, bizarre problems, and general headaches.

One of the fundamental problems with snapshots is that they do rely on the cursor library. The ODBC cursor library is a DLL that is optionally loaded (remember `CDatabase::Open()`?) between the ODBC driver manager and the ODBC driver you are using. The cursor library does some weird things to provide scrollability to your application when the driver may not natively support such functionality. Specifically, it caches data. As you scroll forward through the result set your query returned, the cursor library stores the data (and other information) it fetches first to memory and ultimately to temporary files (naming convention: `CTTXXXX.TMP` where `XXXX` is a non-padded hexadecimal, e.g. `CTTA.TMP` or `CTTFFF3.TMP`). When you scroll backwards you are actually reading from the cache and not from the database. It is this process that is often the downfall of snapshot type recordsets. Here are some of the problems, their causes and suggested resolutions:

Temp File Problems

Problem: You find that all these `.TMP` files are left on your system after you run your MFC database program. Or, you find that your application starts generating weird errors that refer to creating or reading/writing a file buffer, perhaps after you have displayed a file dialog box (e.g. to open or save a file to disk).

Cause: This is due to the fact that there is a bug in the cursor library that causes the temp files it creates to be written to whatever is the current directory. So, if your application uses a current directory that has read-only permissions or if your application's current directory changes for any reason, the cursor library will have trouble creating or later finding its temp files. Furthermore, if the cursor library can't find its temp files it can't clean them up and it also may not be able to scroll to records that were cached in the now unlocatable files.

Resolution: Either don't use snapshots, use read-only snapshots if your driver doesn't require the cursor library, or be sure that your current directory doesn't vary between database operations. You can achieve the last by bracketing code that might change the current directory with a `GetCurrentDirectory()` call to determine the current directory and a `SetCurrentDirectory()` call to restore it. This will result in all temp files being written where the cursor library can find, use, and clean them up.

Bad Performance on First Update/Insert/Delete

Problem: When using snapshots you may notice that the first time you call `Update()` or `Delete()` on the recordset to change, insert, or delete a row you have a considerable wait. Subsequent operations are much faster and the length of the initial wait appears to be related to the number of records returned by your query.

Cause: When using snapshots, updates, inserts, and deletes are done using SQL statements. This requires that a second ODBC hstmt be used (an hstmt is the ODBC handle that represents an executable SQL statement). Some drivers, most notably the Microsoft SQL Server driver, can not support multiple active hstmts on a database connection when you are not using server side cursors which snapshots do not use. The cursor library handles this situation less than gracefully by essentially doing a `MoveLast()` to get all pending results from the initial hstmt--the one used to return your result set. Once all this data is cached, the first hstmt is essentially inactive and

the update/insert/delete can proceed. It is this caching of every record between your current position in the result set and the end of the result set that takes so much time. Since the caching only happens on the first Update() or Delete() call, subsequent modifications are not affected by the slow down.

Resolution: Make your result sets small by applying a WHERE clause (see the m_strFilter member of CRecordset) or by limiting the columns you select (this can have unexpected side effects as noted below). Doing this will reduce the amount of data that has to be cached. Use a driver that can support multiple active hstmts per connection. Use a different type of recordset.

No Records or Multiple Records Updated or Deleted

Problem: Intending to update or delete a single record, you receive an exception indicating that either no records were affected or multiple records were affected.

Cause: As mentioned, the cursor library uses SQL statements to update and also to delete records. The exact statements that are generated are what are known as positioned update (or delete) statements because they employ a WHERE clause of the form WHERE CURRENT OF cursorname. When the cursor library intercepts these statements on their way to the ODBC driver, it translates the CURRENT OF cursorname clause into an actual WHERE clause of the form WHERE colA = valueA AND colB = valueB and so on. Where does the cursor library come up with this new WHERE clause? It generates the clause from the cache it maintains. The cursor library attempts to uniquely identify the row you want to affect by building a WHERE clause that includes every column you specify in your initial SELECT statement (usually generated for you by wizard created code—note that there are exceptions to the ‘every column’ part) and the values for those columns based on your current cursor position in the cache. If the columns you are selecting uniquely identify each row (i.e. no two rows will have the same set of values for the columns you select) and no one has been changing data behind your back (which may happen in a multi-user environment), then this approach works fine. However, if you have lots of redundant data in your result set or you select too few columns or ones that tend to have the same values from row to row, you may affect more rows than you intended. Conversely, if the data in the cursor library cache no longer represents actual rows in the underlying database, then your updates and deletes will fail since there was nothing there to change.

Resolution: If you modify your recordset to reduce the columns that are selected, be sure to retain columns that will uniquely identify rows to reduce the likelihood of updating multiple records. Database design rules shun redundant data so you may need to reevaluate your design if you encounter this problem. As far as updates affecting no rows, if the cause is multi-user interaction you should handle the exception that is generated and call Requery() to refresh the result set. The application or the user can then reposition the cursor to the desired row to retry the update which will hopefully succeed this time.

Snapshots Often Aren’t as Static as You’d Like

Problem: You expect that once you open your snapshot recordset changes by other users will no longer be reflected in the result set. This behavior is expected and often desirable since, in some situations, you want an unchanging view of your data. In practice, however, as you scroll through your result set you find that changes that were made since you opened the recordset do show up. Apparently one of the features of the snapshot doesn’t hold true--the data isn’t truly static.

Cause: As implemented by many drivers, snapshots are not really snapshots until you’ve traversed the entire result set. The basic reason is that membership in the result set isn’t fixed until the last row in the result set is fetched. In fact, since the cursor library is in use, once the last record is read the static nature of the data is

guaranteed regardless of the driver's behavior since all subsequent fetches are from the data cache and not from the database.

Resolution: In the event that you observe the above mentioned behavior, you can use CRecordset in such a way as to minimize the problem. If you want as true a snapshot as you can get with a driver that doesn't implement snapshots in the expected manner, call CRecordset::MoveLast() immediately after opening the recordset. You will lock down the result set with minimum likelihood of capturing unwanted changes from other connections to the database.

Snapshots Aren't All Bad

Having just trashed the reputation of snapshots, let me redeem them in your eyes a bit. Snapshots can actually improve performance in some cases. If your application will be constantly scrolling back and forth through the result set and your DBMS is on a possibly overloaded server accessed over a potentially slow network, a snapshot's use of the cursor library can speed up fetches. Whereas other recordset types must go to the database for every fetch, once a snapshot has fetched and cached data it need not access the database except to modify records. So, users of DBMSs such as Oracle or SQL Server may benefit from using a snapshot now and then.

Dynaset

Dynasets are usually preferred to snapshots since they don't require cursor library support, use a more efficient update/delete mechanism, and take advantage of using "server side cursors" for server based DBMSs (in particular Microsoft SQL Server). By not using the cursor library dynasets avoid the problems and overhead of maintaining a data cache. Updates, insert, and deletes are performed using an ODBC function--SQLSetPos(). This function takes advantage of the fact that the same keyset driven cursor that is used for fetching data can be used to modify data. No SQL is involved so there is no extra SQL parsing required by the DBMS. Using server side cursors also means that much of the processing burden is put on the DBMS which is presumably designed to best do the job.

Users of Microsoft SQL Server will need to follow a few rules to use and optimize dynasets. In order to even open an updatable dynaset on a SQL Server table you must have a unique key in your table. To get the most out of a dynaset you should be aware that including a BLOB (Binary Large Object) amongst the columns you select will reduce performance when updating or deleting records. The reason is that the Microsoft SQL Server driver can not support certain functionality required by MFC to allow the use of SQLSetPos to perform updates and deletes. As a result, selecting a BLOB field will result in the use of update, insert, and delete SQL statements (but not the loading of the cursor library).

One disadvantage of dynasets when used with Microsoft SQL Server is that, as a keyset driven cursor, they require some initial startup overhead. With particularly large result sets the delay on opening a dynaset is due to the considerable amount of processing that is required on the DBMS server to generate the keyset that is then used by the cursor. To minimize this processing time you can reduce your result set via a WHERE clause (see the m_strFilter member of CRecordset) or use a dynamic cursor instead.

One feature of a dynaset is that as you scroll through the result set, you will see changes in the data you fetch due to the activity of other connections or users. This is one of the reasons to use a dynaset—to get the latest data available so you don't oversell concert tickets or promise a room that was booked two seconds ago. One property of dynasets as implemented by some drivers that may be unexpected is that, until you scroll to the last record of the result set, records added by other users have the potential to show up in your result set. However, once you

have moved to the last record this behavior is no longer seen and the membership of the result set is then fixed. This is essentially the same phenomenon seen with snapshots and is due to the way in which specific drivers chose to implement the different cursor types. If this behavior is undesirable, dynamic cursors may provide a viable alternative.

Dynamic

This type of recordset is based on the most complex cursor to implement. Very few drivers support dynamic cursors because of this. Microsoft SQL Server is the only example of which I am aware. You can basically think of a dynamic recordset as a snapshot that has no initial overhead of keyset creation and whose result set membership never becomes fixed. So, a dynamic cursor can open faster than a keyset driven cursor and you will always see both changes in data and newly added rows that are due to the activity of other connections or users. While there is slightly more overhead in scrolling from row to row in a dynamic recordset than in a dynaset, on SQL Server the difference is undetectable.

In Conclusion

Knowing the types of recordsets and when to use which will enable you to fine tune your MFC database applications to get the best mix of performance and features that your ODBC driver allows. Using the right recordset can make the difference between glacial slowness and nearly instantaneous response and between total failure and complete success. One last note to keep in mind is that bugs that may be exposed by one type of recordset may be side stepped by using another. So, don't be afraid to experiment.