Visual C++ and MFC Programming

Table of Contents

Table of Figures

Chapter 1:
Introduction to Microsoft Visual C++

✍ The: Microsoft Visual C++ Environment

✍ Floatable and Dockable Windows

✍ Visual C++ Projects and Files

☞ Help

1.1 The Microsoft Visual C++ Environment

1.1.1 Introduction

Microsoft Visual C++ is a programming environment used to create applications for the Microsoft Windows operating systems. To use this ebook, you must have installed either Microsoft Visual C++ 6.0 or Microsoft Visual Studio 6.0.

Although Microsoft Visual C++ .Net provides two programming environments for the price of one, in this book, we will use Microsoft Visual Studio 6.0 and we will cover only Microsoft Foundation Class (MFC) library programs.

After installing it, to use the programming environment, you must first open it. To do that, you would click Start -> (All) Programs -> Microsoft Visual Studio 6.0 -> Microsoft Visual Studio 6.0.

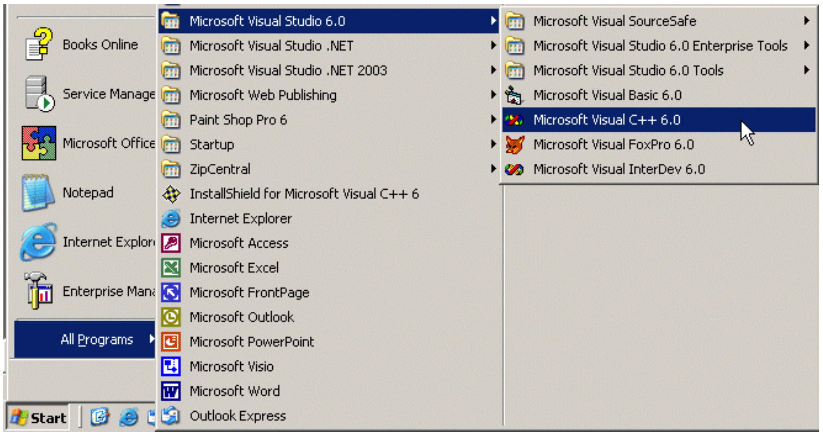
If you were using version 6, to open it, you would click Start -> (All) Programs -> Microsoft Visual Studio 6.0 -> Microsoft Visual C++ 6.0

This book uses the -> arrow for the menu requests.
From now on, in this book,

Request Example	Means
Edit -> Copy	Click Edit then click Copy
View -> Toolbars -> Custom	Click View position the mouse on Toolbars, and then click Custom

Practical Learning: Launching Microsoft Visual C++

?? To start Microsoft Visual C++ or Visual Studio, on the Taskbar, click Start (All) Programs -> Microsoft Visual Studio 6.0 -> Microsoft Visual Studio 6.0



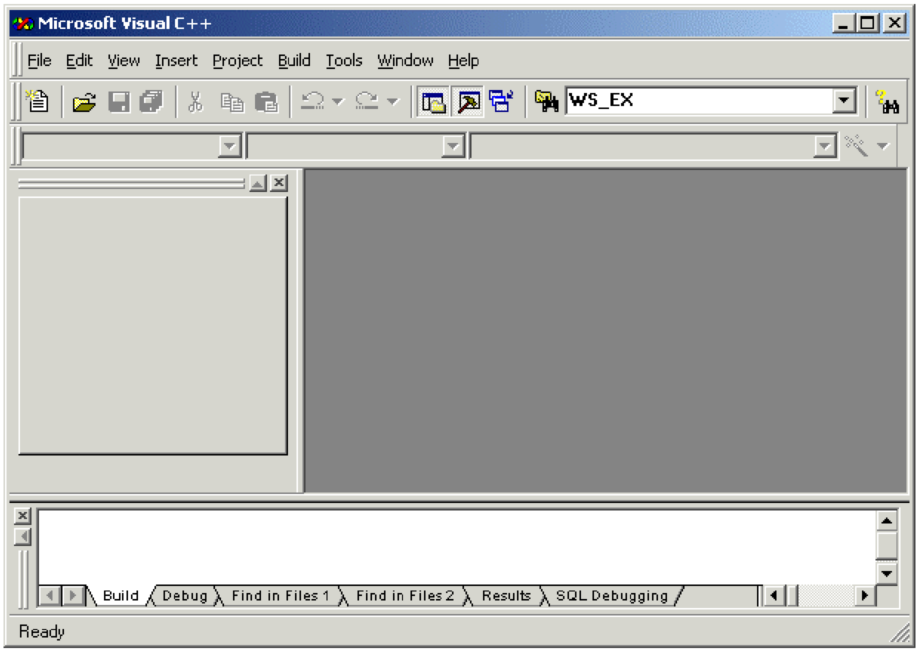


Figure 1: Microsoft Visual Studio IDE

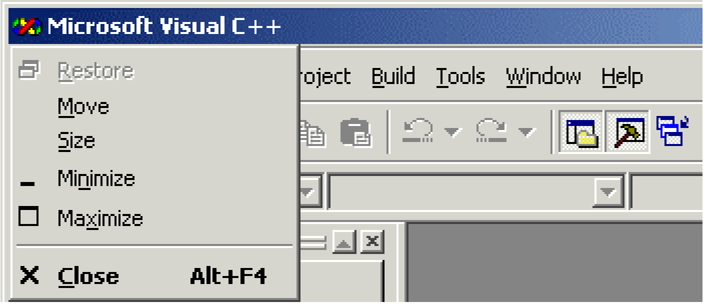
1.1.2 The Integrated Development Environment

After Microsoft Visual Studio has been opened, the screen you look at is called an Integrated Development Environment or IDE. The IDE is the set of tools you use to create a program. The IDE design of Microsoft Visual C++ 6 and Microsoft Visual Studio .Net is significantly different. While version 6 had its unique design as compared to Microsoft Visual Basic, the programming environments of Microsoft share the same look in the 2002 and 2003 releases.

1.1.3 The Title Bar

1. The system icon is used to identify the application that you are using. Almost every application has its own system icon. The system icon holds its own list of actions; for example, it can be used to move, minimize, maximize or close (when double-clicked) a window.

   To see its list of actions, click it

2. To see an example, while the system menu is displaying, click Minimize. To bring back the IDE, on the Task bar, click Microsoft Visual C++

3. The main section of the title bar displays Microsoft Visual C++. This starting title displays in the beginning until you decide to create a particular type of application, then the title changes. You will experience it once we start some programming assignments.

4. The main section of the title bar is also used to move, minimize, maximize the top section of the IDE, or to close Visual Studio. On the right section of the title bar, there are three system buttons with the following roles

Button		Role
_	_	Minimizes the window
□	□	Maximizes the window
🗗	🗗	Restores the window
☒	☒	Closes the window

1.1.4 The Main Menu

Under the title bar, there is a range of words located on a gray bar. This is called the menu or main menu.

In this ebook, the expression “Main Menu” refers to the menu on top of the IDE.
MSVC means Microsoft Visual C++
MSVC 6 means Microsoft Visual C++ 6.0
MSVC 7 means Microsoft Visual C++ 2003

To use a menu, you click one of its words and the menu expands.

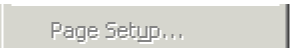
If an item is missing from the main menu, you can customize it.

1 Click File. There are four main types of menus you will encounter.

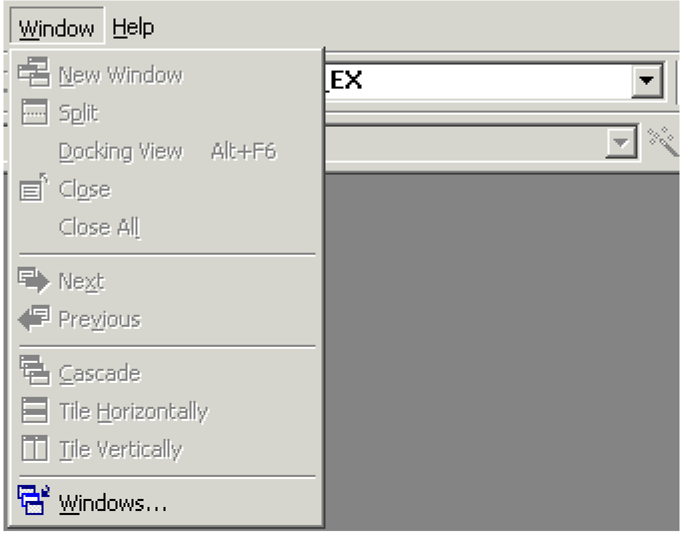
When clicked, the behavior of a menu that stands alone depends on the actions prior to clicking it. Under the File menu, examples include Close, Save All, or Exit. For example, if you click Close, Microsoft Visual Studio will find out whether the current file had been saved already. If it has been, the file would be closed; otherwise, you would be asked whether you want to save it before closing it

2 To see an example, click Exit.

3 Start Microsoft Visual C++ the same way we did earlier

4 

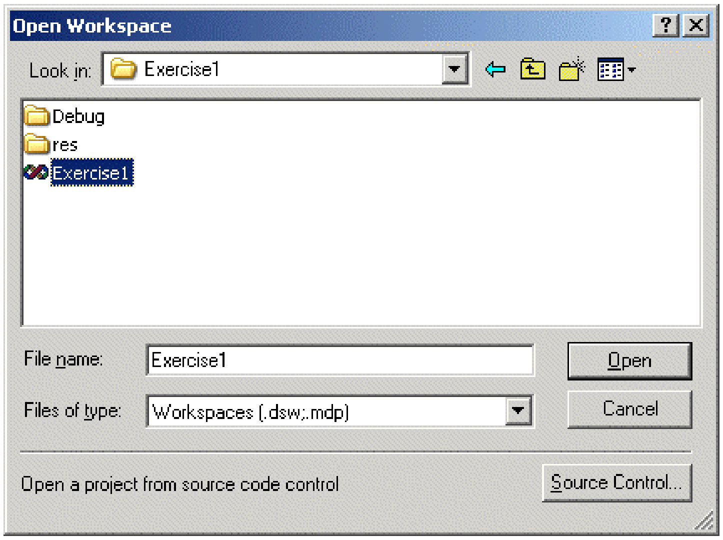
A menu that is disabled is not accessible at the moment. This kind of menu depends on another action or the availability of something else. To see an example, one the main menu, click Window:



5

A menu with three dots means that an intermediary action is required in order to apply its assigned behavior. Usually, this menu would call a dialog box where the user would have to make a decision.
As an example, on the main menu, position the mouse on File and click Open Workspace...

6 On the dialog box, locate the folder that has your downloaded exercises. Locate and display the Exercise1 folder in the Look In combo box:



Look in: Exercise1

File name: Exercise1

Files of type: Workspaces (.dsw;.mdp)

Open a project from source code control

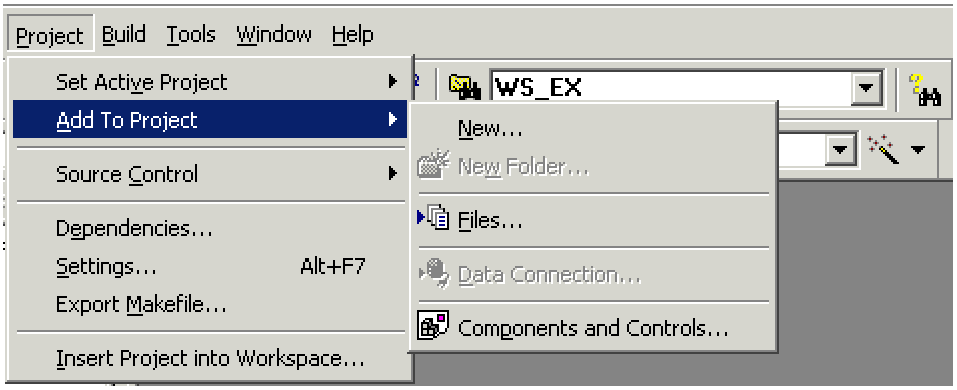
Figure 2: Open Workspace

7 Click Exercise1 and click Open.

8

A menu with an arrow holds a list of menu items under it. A menu under another menu is called a submenu. To use such a menu, you would position the mouse on it to display its submenu.

For example, on the main menu, click Project and position the mouse on Add To Project...



9 To dismiss the menu, click Project

10 Notice that, on the main menu (and any menu), there is one letter underlined on each word. Examples are F in File, E in Edit, V in View, etc. The underlined letter is called an access key. It allows you to access the same menu item using the keyboard. In order to use an access key, the menu should have focus first. The menu is given focus by pressing either the Alt or the F10 keys.

11 To see an example, press Alt

12 Notice that one of the items on the menu, namely File, has its borders raised. This means that the menu has focus.

13 Press t and notice that the Tools menu is expanded.

14 When the menu has focus and you want to dismiss it, you can press Esc.

For example, press Esc.

15 Notice that the Tools menu has collapsed but the menu still has focus.

16 Press f then press o. Notice that the Open dialog box displays.

17 To dismiss the Open dialog box, press Esc

18 On some menu items, there is a key or a combination of keys we call a shortcut. This key or this combination allows you to perform the same action on that menu using the keyboard.

If the shortcut is made of one key only, you can just press it. If the shortcut is made of two keys, press and hold the first one, while you are holding the first, press the second key once and release the first key. Some shortcuts are a combination of three keys.

To apply an example, press and hold Ctrl, then press o, and release Ctrl.

19 Notice that the Open dialog box opens. To dismiss it, press Esc

From now on, in this book,	
Press	Means
T	Press the T key
Alt, G	Press and release Alt. Then press G
Ctrl + H	Press and hold Ctrl. While you are still holding Ctrl, press H once. Then release Ctrl
Ctrl + Shift + E	Press and hold Ctrl. Then press and hold Shift. Then press E once. Release Ctrl and Shift

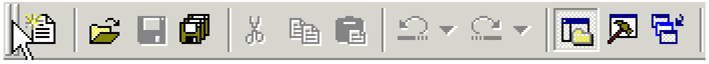
One of the differences between Microsoft Visual C++ 6 and Microsoft Visual Studio .Net is that, on version 6, the menu bar can be moved to any section on the IDE. To do this, you can click and hold on the small vertical grab bars on the left side of the File menu, then drag to any location of your choice. In the 2002 and 2003 versions, the menu bar cannot be moved.
In all versions, the main menu is customizable. This means that you can add and remove items from the menu.

1.1.5 The Toolbars

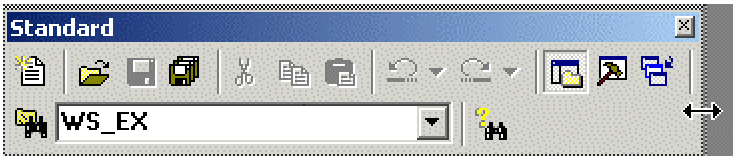
A toolbar is an object made of buttons. These buttons provide the same features you would get from the (main) menu, only faster. Under the main menu, the IDE is equipped with an object called the Standard toolbar. For example, to create a new project, on the main menu, you could click File -> New -> Project… On the other hand, the Standard toolbar is equipped with a button to perform the same action a little faster.

By default, the Standard toolbar is positioned under the main menu but you can position it anywhere else on the IDE. Like the menu, the toolbars can be customized.

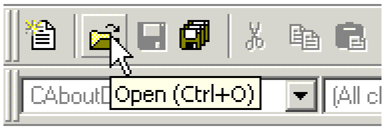
1. Click and drag the gripper on the Standard toolbar down and right:



2. Notice that the toolbar has moved.
3. Once moved, you can resize the toolbar. To do that, position the mouse on the right border of the toolbar. Click and drag in the left direction:



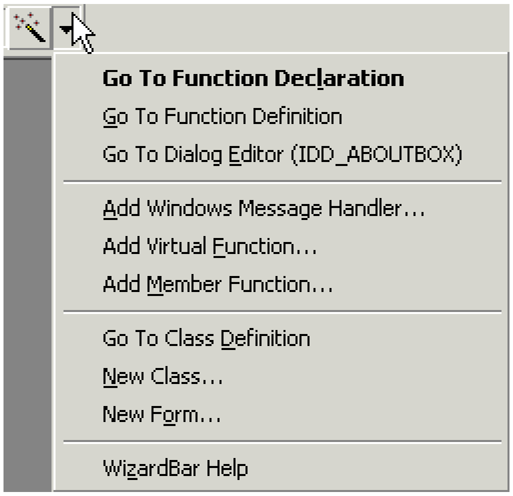
4. To restore the toolbar to its previous position, double-click its title bar.
5. You can get a list of the toolbars that are available if you right-click any button on any toolbar or menu.
   For example, right-click a toolbar and notice the list
6. To dismiss the menu, press Esc.
   In this book, every toolbar is referred to by its name
7. A toolbar is equipped with buttons that could be unfamiliar. Just looking at one is not obvious. The solution into knowing what a button is used for is to position the mouse on top of it. A tool tip will come up and display for a few seconds.
   As an example, position the mouse (do not click) on the second button from left on the Standard toolbar:



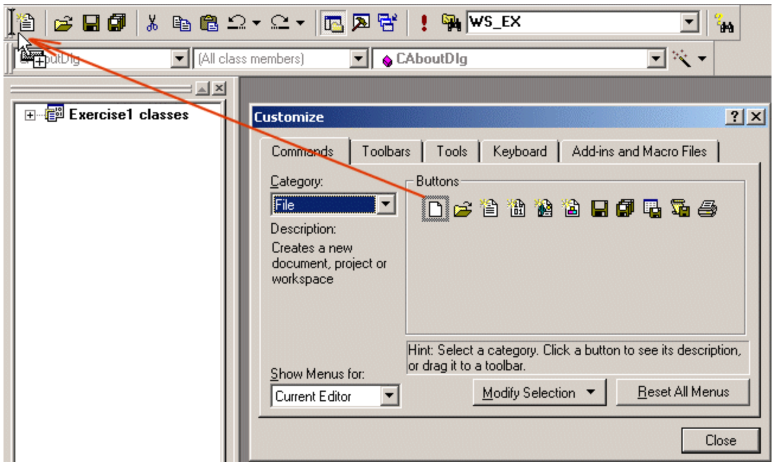
8. Without clicking, move the mouse to another button and to other buttons

From now on, each button on any toolbar will be named after its tool tip. This means that, if a tool tip displays "Hungry", its button will be called the Hungry Button. If a tool tip display "Exercises and Assignments", its button will be called the Exercises and Assignments button. If you are asked to click a button, position your mouse on different buttons until one displays the referred to name.

9. To use a toolbar's button, you click it. For example, click the Open button . Notice that the action calls the Open dialog box.
10. Press Esc to dismiss the New Project dialog box.
11. Some buttons present an arrow on their right side. This arrow represents a menu. To see an example, position the mouse on the Wizard Bar Actions button and click the arrow on the right side. Observe the menu:


Go To Function Declaration
Go To Function Definition
Go To Dialog Editor (IDD_ABOUTBOX)
Add Windows Message Handler...
Add Virtual Function...
Add Member Function...
Go To Class Definition
New Class...
New Form...
WizardBar Help

12. Press Esc to dismiss the menu.
13. To customize the Standard toolbar by adding buttons to it, right-click anything on the menu bar or the toolbar and click Customize...
14. On the Customize dialog box, click the Commands tab.
15. In the Categories list, select File
16. Under the Buttons combo box, click and drag the New button. Position it to be the most left button on the Standard toolbar



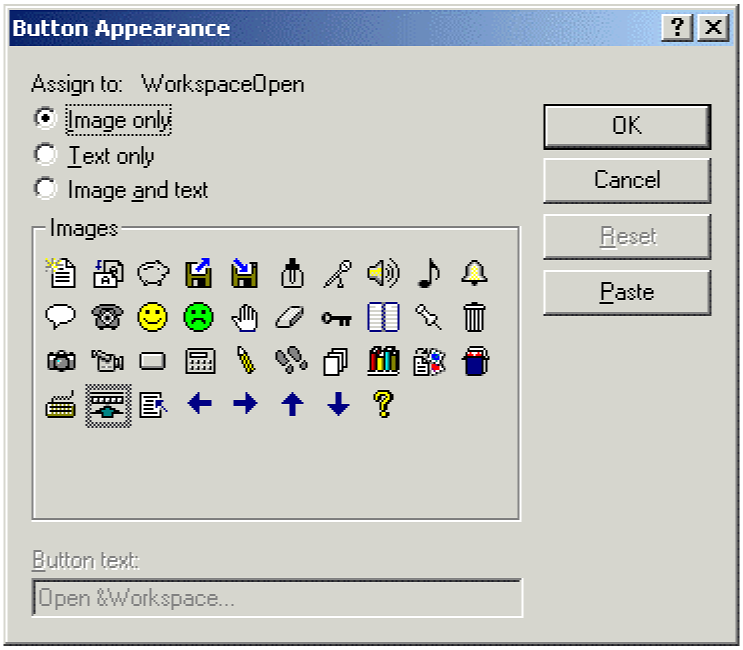
Description:
Creates a new document, project or workspace

Hint: Select a category. Click a button to see its description, or drag it to a toolbar.

Figure 3: Customizing the IDE Resources

17. Release the mouse
18. To add another toolbar button, while the Customize dialog box is still displaying, on the main menu, click File

19. Press and hold Ctrl. Click and hold Open Workspace… Then drag it and position it anywhere on the Standard toolbar
20. Release Ctrl and click File to dismiss its menu
21. On the Standard toolbar, click and drag Open Workspace… to position it to the right of the New button
22. On the Standard toolbar, right-click Open Workspace and click Button Appearance…
23. Click the Image Only radio button. In the Images section, click the icon on 2nd column – 4th row



24. Click OK on the Button Appearance dialog box
25. Click the Close button on the Customize dialog box

1.1.6 The Studio Windows

When creating your applications, you will use a set of windows that each accomplishes a specific purpose. In some windows, information is organized in a tree list equipped with + or – buttons. To expand a list, you can click its + button. To collapse a list, click its – sign

The workspace is divided into individual windows that, by default, are docked together as an ensemble:

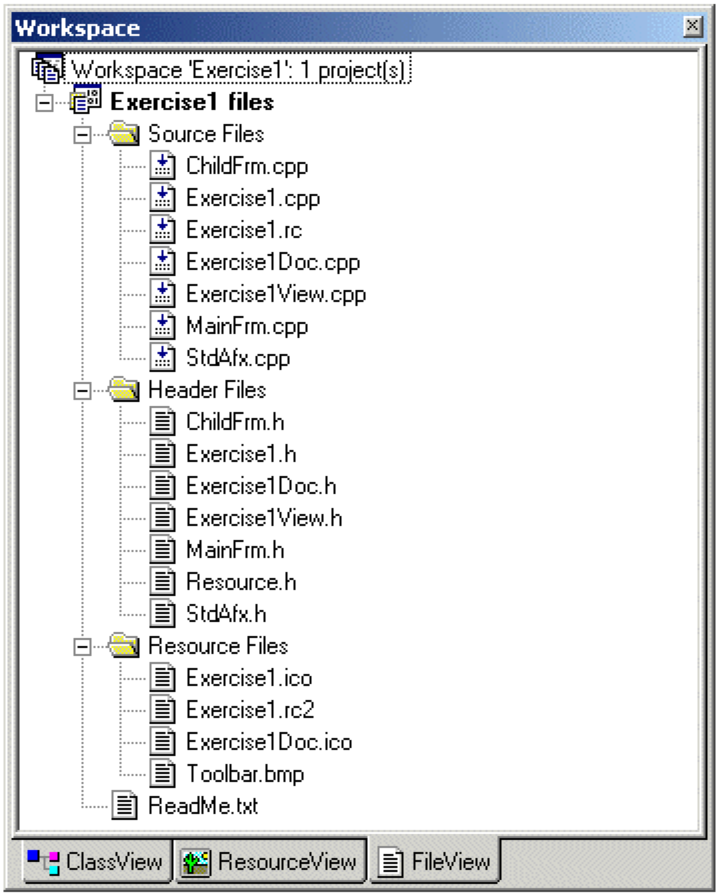


Figure 4: The Workspace Window

The FileView tab of is used to display the project name and its categories of files. The categories are organized as folders and include the Source Files, the Header Files and the Resource Files. They can also display other dependent files.

The Class View is used to display the classes, independent functions and variables used by the project:

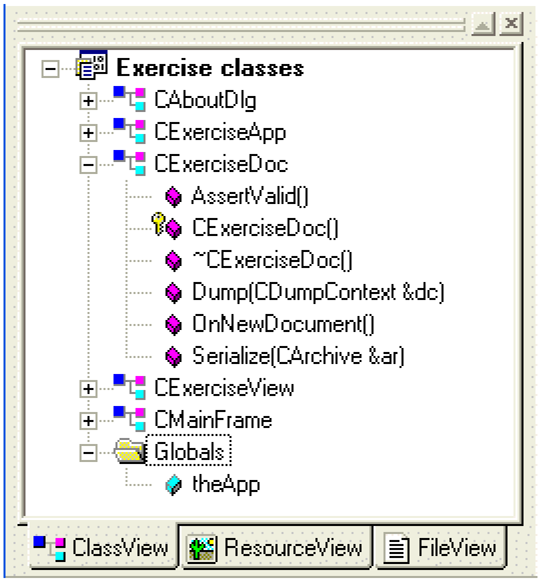


Figure 5: The Resource View

The Resource View displays an organized tree list of the resources used on a project:

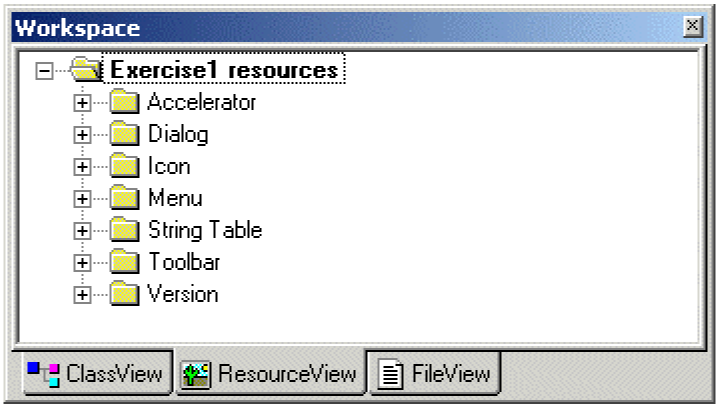
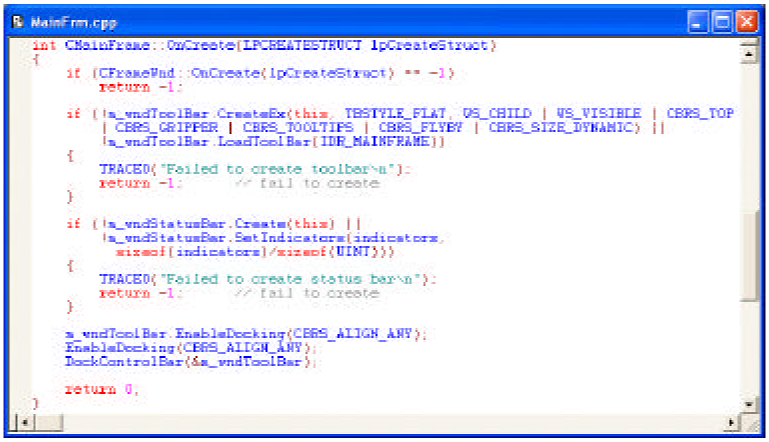


Figure 6: The ResourceView tab of the Worskpace

Practical Learning: Using the Studio Windows

1. In the Workspace, click the ClassView tab and, if necessary, click the + button of Exercise to expand it
2. Notice that the names of classes start with C
3. Double-click CMainFrame
4. Notice that this displays the contents of the CMainFrame class in the Code Editor
5. Also notice that the name of the file displays on the title bar, is MainFrm.h
6. To show the source file of the CMainFrame class, expand the class and double-click OnCreate to display its listing in the Code Editor:



```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
		| CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
		!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
	{
		TRACE0("Failed to create toolbar\n");
		return -1;      // fail to create
	}

	if (!m_wndStatusBar.Create(this) ||
		!m_wndStatusBar.SetIndicators(indicators,
		  sizeof(indicators)/sizeof(UINT)))
	{
		TRACE0("Failed to create status bar\n");
		return -1;      // fail to create
	}

	m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
	EnableDocking(CBRS_ALIGN_ANY);
	DockControlBar(&m_wndToolBar);

	return 0;
}
```

Figure 7: The Source Code Editor

1.2 Floatable and Dockable Windows

1.2.1 Description

An object, called a window, is referred to as dockable when it can assume different "gluable" or floating positions on the screen. This means that such a window can be moved and glued to another or it can simply be placed independently of other windows of the same application. Windows that are dockable indicate this by the presence of "grippers", which are 3-D lines or bars on the left side of the window.

To move a window, find its gripper(s). Click and drag it in the desired direction. To dock a window is to glue it either to one side of the IDE or to another window. To do this, drag its gripper(s) to the desired side or the necessary window and release the mouse.

Practical Learning: Docking and Floating Windows

1. Click and hold the mouse on top of the Workspace
   Drag to the center of the window

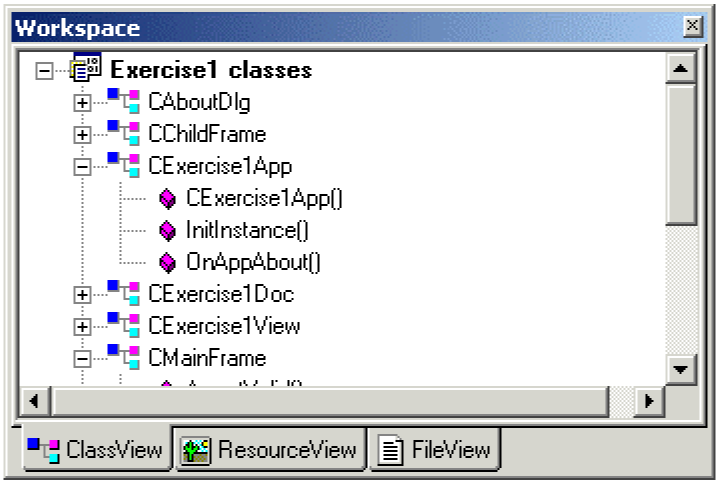


Figure 8: Moving a Dockable Window

2. To position the window to its previous position, double-click its title bar

1.3 Visual C++ Projects and Files

1.3.1 Creating a New Project

Microsoft Visual C++ allows creating various types of projects ranging from regular applications to complex libraries, from Win32 applications to communications modules or multimedia assignments.

Various options are available to create a project:

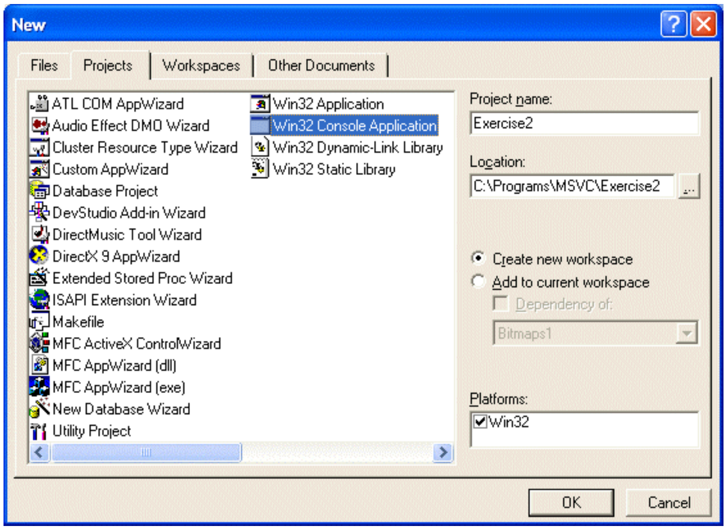
?? On the main menu, you can click File -> New…

?? The shortcut to create a new project is Ctrl + N

Any of these actions displays the New Project dialog box from where you can select the type of project you want.

Practical Learning: Creating a Microsoft Visual C++ Project

1. On the main menu, click File -> New…
2. In the New dialog box, click the Projects tab
3. In the list, click Win32 Console Application
4. In the Location edit box, replace the text with **C:\MSVC**
5. In the Name edit box, type **Exercise2**



Project name: Exercise2

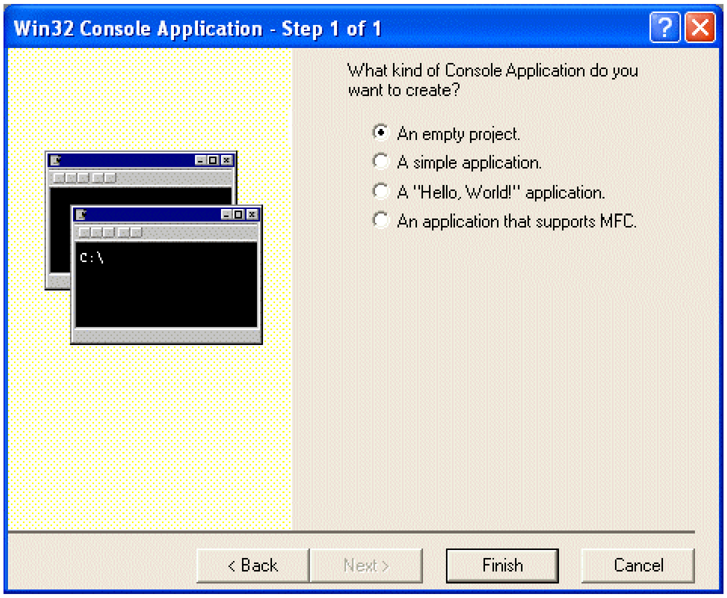
Location: C:\Programs\MSVC\Exercise2

● Create new workspace
○ Add to current workspace
□ Dependency of:
Bitmaps1

Platforms:
☑ Win32

Figure 9: Creating a New Project - Exercise1

6. Click OK
7. In the Win32 Console Application – Step 1 of 1, click An Empty Project



What kind of Console Application do you want to create?

● An empty project.
○ A simple application.
○ A "Hello, World!" application.
○ An application that supports MFC.

Figure 10: Win32 Application Wizard - Exercise1

8. Click Finish and click OK

1.3.2 Creating Files

Although the most popular files used in Visual C++ are header and source files, this IDE allows you to create various other types of files that are not natively C++ types.

To create a file, on the main menu, you can click File -> New… This would open the New dialog box from where you can select the desired type of file and click Open. By default, the new file would open in the Source Code Editor. If you want to open the file otherwise, after selecting it in the New dialog box, click the arrow of the Open As combo box

Practical Learning: Creating a C++ File

1. On the main menu, click File -> New…
2. Make sure the Files tab is selected. Click C++ Source File
3. In the Name edit box, type Main and press Enter
4. In the emp ty file, type the following:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    cout << "Welcome to College Park - Auto Parts\n";

    return 0;
}
```

5. To save everything, on the Standard toolbar, click the Save All button

1.3.3 Adding Existing Files to a Project

If a file had already been created and exists on a separate folder, drive, directory or project, you can import and add it to your application. When you decide to adding an existing file to a project, because Visual allows you a great level of flexibility on the types of files you can add, it is your responsibility to check that the file is “valid”, especially if the file was neither created using MSVC nor compiled in Visual C++.

If you copy a file from somewhere and paste it in the folder that contains your project, the file is still not considered as part of the project. You must explicitly add it. Before adding an existing file to your project:

?? On the main menu, you can click File -> Add Existing Item…

?? On the main menu, you can also click Project -> Add Existing Item…

?? In the Solution Explorer, you can right-click the name of the project, position your mouse on Add and click Add Existing Item…

Any of these actions would open the Add Existing Item dialog box. This requires you to locate the item and select it. Once the item is ready, to add it to the project, click the Open button. If you want to open it first, click the arrow of the Open button and specify how you want to open the file.

Practical Learning: Adding a File to a Project

1. On the main menu, click Project -> Add To Project -> Files…
2. Locate the folder that contains the exercises that accompany this book and display it in the Look In combo box.

3. In the list of files, click **Main.h** and click Open
4. In the FileView tab of the Workspace, if necessary, expand the Header Files node. Double-click Main.h to display it in the Source Code Editor

1.3.4 Adding Classes

To add a class to your project, you have various options. You can separately create a header file then create a source file. You can also import an existing header file and a source file from any valid path.

Microsoft Visual C++ makes it easy to create a C++ class. When doing this, a header and a source files are prepared. A (default) constructor and a destructor are added. You can delete or completely change the supplied constructor and/or the destructor. Also, Visual C++ includes the name of the header file in the source file so you would not have forgotten to do this.

Most classes in Visual C++ have their names starting with the letter C. Although this is only a suggestion, and any valid class name would work fine, to keep some harmony, in this book, we will follow this convention.

Note

In this book, we will not follow the famous "hungarian" naming convention because nobody seems to be able to follow or keep up with it, not even programmers from Microsoft.

To create a new class and add it to your project:

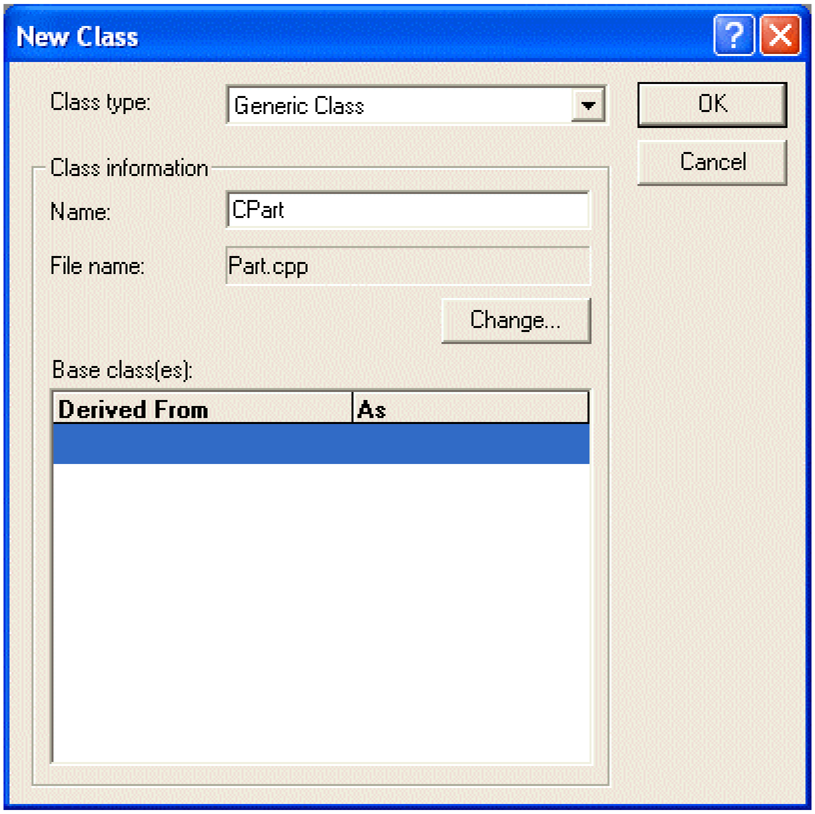
?? On the main menu, you can click Insert -> New Class …

?? In the ClassView tab of the Workspace, you can right-click the name of the project and click New Class…

Any of these actions would open the New Class dialog box. From here, you can specify the type of class you want to create and click Open. You will then be asked to provide a name for the class.

Practical Learning: Creating a Class

1. To create a new class, in the ClassView tab, right-click Exercise2 and click New Class…
2. In the New Class dialog box, in the Name edit box, type CPart



Class type: Generic Class

Name: CPart

File name: Part.cpp

Figure 11: Adding a C++ Class

3. Click OK

4. To save everything, on the Standard toolbar, click Save All

1.3.5 Opening Files

A file is a series of computer bits that are aligned in the computer memory as one entity to constitute a document. To open a file, simply display the Open dialog box and proceed as you would for any other computer application that can be used to open a file.

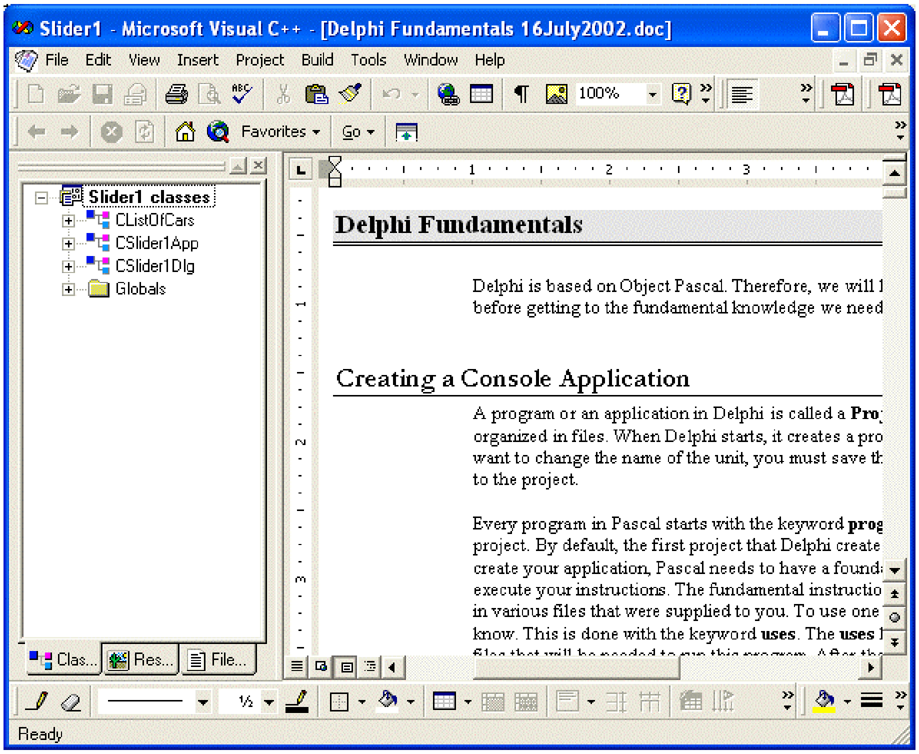
If you had previously created or opened a file, Visual C++ keeps a list of the most recently used (MRU) file under its File menu. To control the maximum number of files that can be listed:

?? If you are using MSVC 6, display the Options dialog box and access its Workspace property page. Then change the value of the Recent File List Contains edit box

?? If you are using MSVC 7, open the Options dialog box and, under the Environment node, click the General link. Then, change the value of the second Display line

Its ability to open various types of files allows you to view a word processing document, a spreadsheet or a presentation.

When opening a file that is not a "native" C++ or Visual C++ file, the main menu and toolbar of its parent application take over the top area of the Visual C++ environment. This gives you all the features of the application. For example, you can open a word processing document and use it as if you were working from the application in which the document was created. Here is a Microsoft Word document opened in Visual C++:


Slider1 - Microsoft Visual C++ - [Delphi Fundamentals 16July2002.doc]
File Edit View Insert Project Build Tools Window Help
100%
Favorites
Go
Slider1 classes
CListOfCars
CSlider1App
CSlider1Dlg
Globals
Clas...
Res...
File...
Ready

Delphi Fundamentals

Delphi is based on Object Pascal. Therefore, we will l
before getting to the fundamental knowledge we need

Creating a Console Application

A program or an application in Delphi is called a **Pro**
organized in files. When Delphi starts, it creates a pro
want to change the name of the unit, you must save th
to the project.

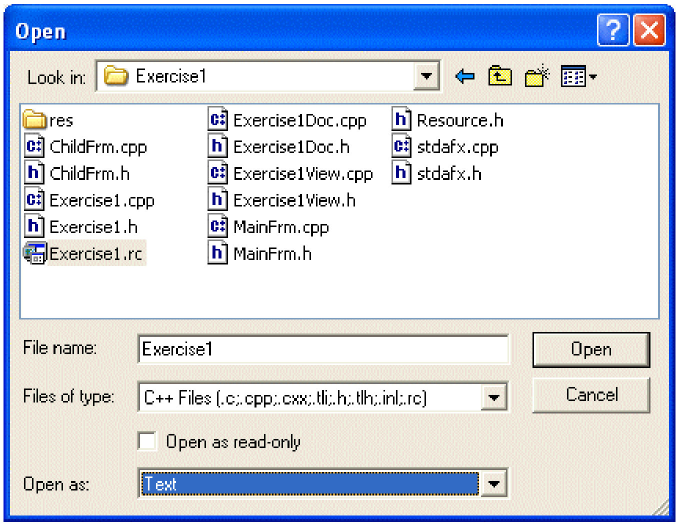
Every program in Pascal starts with the keyword **prog**
project. By default, the first project that Delphi create
create your application, Pascal needs to have a found
execute your instructions. The fundamental instructio
in various files that were supplied to you. To use one
know. This is done with the keyword **uses**. The **uses** l

Figure 12: File Opening in Visual Studio

This means that C++ files are not the only files you are able to view in Visual C++. Although you can open various types of document, this does not imply that any of them would display easily. Some of them may have rules to follow. That is why the Open dialog box of Visual C++ is equipped with the Open As combo box. For example, when opening a file that has the .rc extension, which is a resource file, you have the option of opening it automatically or as a text file and the result would be different depending on the selection option.

Practical Learning: Opening Files

1. To open a new file, on the main menu, click File -> Open…
2. Locate the exercises that accompany this book and display the Exercise1 folder in the Look In combo box
3. Make sure the Files of Type is set to C++ Files (.c;, cpp:, .cxx;, .tli;, .h;, .tlh;, .inl;, .rc) or All Files.
   Click Exercise1View.cpp and click Open
4. To open another file, on the Standard toolbar, click the Open button
5. Click Exercise1.rc
6. Click the arrow of the Open As combo box and select Text



File name: Exercise1

Files of type: C++ Files (.c;.cpp;.cxx;.tli;.h;.tlh;.inl;.rc)

Open as: Text

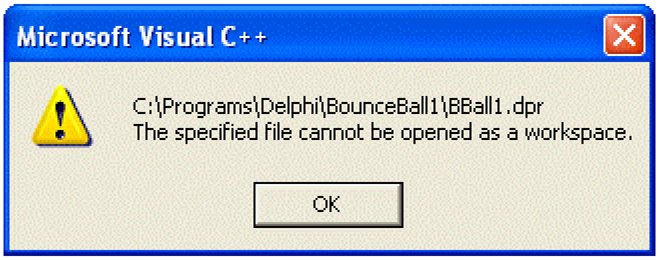
Figure 13: Opening a File With the Open Dialog Box

7. Click Open
8. Notice that the file was opened as text although it is a resource file, just like the previous one
9. To close the file, using the main menu, click Window -> Close
10. To close the project, on the main menu, click File -> Close Workspace
11. When asked whether you want to close all documents, click Yes. Also, agree to save the Main.cpp file

1.3.6 Opening Existing Projects

A project is made of various files and subject to the environment in which it was created. A project itself is a file but it is used to “connect” all the other files that compose its particular application.

There is a limit on the types of projects you can open in Visual C++. This is not an anomaly. Every computer-programming project is created using a particular environment and each environment configures and arranges its project as its judges it necessary. For example, the files necessary to create a Borland Delphi project are not the same for a Visual C++ project and Visual C++ would not need to use such a project. Therefore, any attempt to open an unrecognizable project would produce an error. Here is a message box resulting from trying to open a Delphi project:

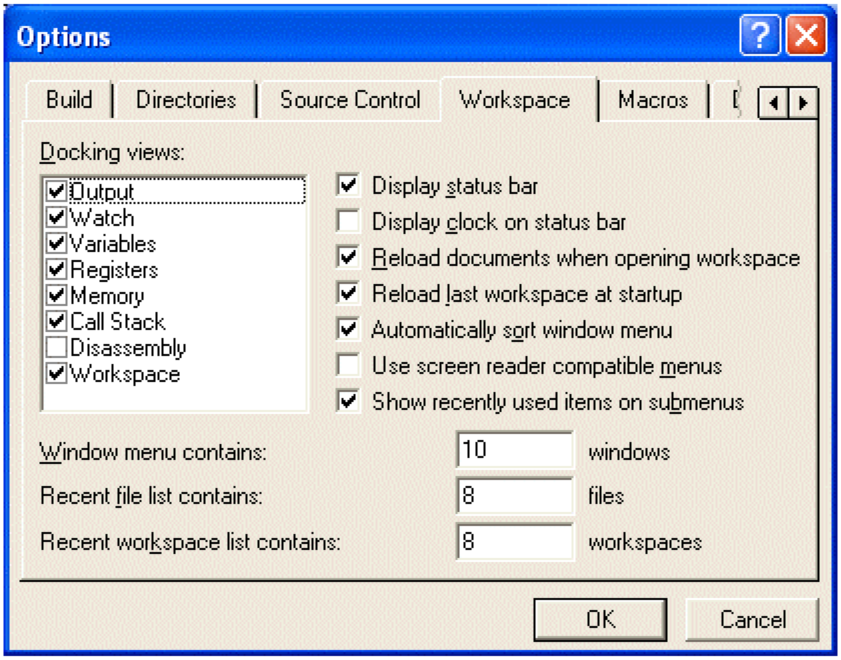


C:\Programs\Delphi\BounceBall1\BBall1.dpr
The specified file cannot be opened as a workspace.

Figure 14: A Microsoft Visual C++ Message Box

Visual C++ is configured to easily open its own Visual C++ projects that have the .dsw or .mdp extensions. Additionally, you can open a Win32 project that was created using Visual C++ (you may not be able to open a Win32 project created using Borland C++ Builder). You can also open Visual C++ and Win32 projects created with previous versions of Visual C++.

To open a project, if you have used it as one of the previous 8 projects, when Visual Studio opens and displays the Start Page, you should see a list of the last 8 projects used. You can then click it to open it. As done with files, Visual Studio keeps a list of the previous projects on its File menu. Therefore, to open one of the previously used projects, you can click File -> Recent Projects. If you see the project you are looking for, you can click it. If you write a lot of projects regularly, you may want to increase the list of files and/or projects that the file menu keeps. To do this, on the main menu, you can click Tools -> Options… In the Options dialog box, click the General node under Environment and set the number desired for the Display edit boxes:



Docking views:

☑ Output
☑ Watch
☑ Variables
☑ Registers
☑ Memory
☑ Call Stack
☐ Disassembly
☑ Workspace

☑ Display status bar
☐ Display clock on status bar
☑ Reload documents when opening workspace
☑ Reload last workspace at startup
☑ Automatically sort window menu
☐ Use screen reader compatible menus
☑ Show recently used items on submenus

Window menu contains: 10 windows

Recent file list contains: 8 files

Recent workspace list contains: 8 workspaces

Figure 15: The Options Dialog Box

Practical Learning: Opening a Project

1. To open a project, on the main menu, click File
2. Position the mouse on Recent Workspaces and click Exercise2

1.4 Getting Help

1.4.1 Online Help

While working on your projects, you will usually need to figure out how something can be done and sometimes why it is done like that. This is where you will need help. There are three primary types of help you can use: online, Microsoft, and others.

When Microsoft Visual C++ 6 gets installed, it asks you whether you want to install the MSDN library. This is because, besides the CDs that hold Visual C++ or Visual Studio installation files or the DVD, additional CDs are provided to you just for the help files. In some cases, the help files are packed in one an additional DVD. If you decide to install the MSDN library, it would be installed as an independent program but would be internally linked to Visual Studio. Because this library is particularly huge, you can/should keep it on the CDs or the DVD and access it only when needed. The help files that are installed with Visual Studio constitute what is referred to as Online Help.

Microsoft Visual Studio .Net installs its help files along with the programming environment and creates one to three tabs for them on the Solution Explorer window.

To get help on an MFC class while you are working in the Code Editor, click the class and press F1, the HTML Help window would display help on the topic. You can also access the online directly in the HTML Help window or the help tabs of MSVC .Net

Practical Learning: Getting Online Help

1. Open the Exercise application from the exercises that accompany this book
2. In the Class View, double-click CMainFrame to display its header file.
3. In the Code Editor window, click CFrameWnd and press F1
4. Notice that a description of the CFrameWnd class displays
5. On the main menu, click Help -> Index...
6. Type WinMain and press Enter

1.4.2 Other Help Types

Among the C++ programming environments, Microsoft Visual C++ is the most documented on the internet. The primary place of help is on the MSDN library web site. Besides the information you receive on the Visual C++ CDs or DVD, this Microsoft web site is updated regularly.

There many other sites dedicated to Visual C++. We also provide ours and a site that supports this book.

The other type of help available is through newsgroups, coworkers, or friends.

Chapter 2: Introduction to MFC

✍ The Microsoft Foundation Class Library

✍ Frames Fundamentals

✍ Message Boxes

2.1 The Microsoft Foundation Class Library

2.1.1 Introduction

The Microsoft Foundation Class (MFC) library provides a set of functions, constants, data types, and classes to simplify creating applications for the Microsoft Windows operating systems. To implement its functionality, the MFC is organized as a hierarchical tree of classes, the ancestor of which is **CObject**. Although you can create C++ classes for your applications, most of the classes you will use throughout this book descend directly or indirectly from **CObject**.

2.1.2 CObject, the Ancestor

The **CObject** class lays a valuable foundation that other classes can build upon. Using the rules of inheritance, the functionality of **CObject** can be transparently applied to other classes as we will learn little by little. Some of the features that **CObject** provides are: performing value streaming for saving or opening contents of files, controlling dynamic creation and destruction of its inherited classes, checking the validity of variables of classes, etc.

You will hardly, use **CObject** directly in your program. Instead, you may create your own classes that are based on **CObject**. An example would be:

```
class CStudent : public CObject
{
public:
	CStudent();
	char  *Make;
	char  *Model;
	int   Year;
	long   Mileage;
	int   Doors;
	double Price;
};
CObject Methods
```

When inheriting a class from **CObject**, your object can take advantage of the features of its parent **CObject**. This means that you will have available the functionality laid by its methods. Some of the methods of the **CObject** class are:

CObject(): This constructor allows you to use an instance of **CObject**. If you have created a class based on **CObject**, when you declared an instance of your object, the default **CObject** constructor is available and gives you access to the **CObject** methods.

CObject(const CObject &Src): If you want to copy a variable of your **CObject** derived class to use in your program, for example to assign it to another variable, you can use the inherited copy constructor. Since the object is derived from **CObject**, the compiler would make a copy of the variable on the member-by-member basis.

Serialize(): This method allows you to stream the values of the members of your objects.

AssertValid(): This method is used to check your class. Because an inherited class is usually meant to provide new functionality based on the parent class, when you create a class based on **CObject**, you should provide a checking process of your variables. In this case you should provide your own implementation of the **AssertValid()** method. You can use it for example to check that an **unsigned int** variable is always positive.

The objects of an applications send messages to the operating system to specify what they want. The MFC provides a special class to manage these many messages. The class is called **CCmdTarget**. We will come back to it when dealing with messages.

2.1.3 The Basic Application

To create a program, also called an application, you derive a class from the MFC's **CWinApp**. **CWinApp** stands for Class For A Windows Application.

Here is an example of deriving a class from **CWinApp**:

```
struct CSimpleApp : public CWinApp
{
};
```

Because the **CWinApp** class is defined in the AFXWIN.H header file, make sure you include that file where **CWinApp** is being used. To make your application class available and accessible to the objects of your application, you must declare a global variable from it and there must be only one variable of your application:

```
struct CSimpleApp : public CWinApp
{
};

CSimpleApp theApp;
```

The **CWinApp** class is derived from the **CWinThread** class. The **CWinApp** class provides all the basic functionality that an application needs. It is equipped with a method called **InitInstance().** To create an application, you must override this method in your own class. Its syntax is:

```
virtual BOOL InitInstance();
```

This method is used to create an application. If it succeeds, it returns **TRUE** or a non-zero value. If the application could not be created, it returns **FALSE** or 0. Here is an example of implementing it:

```
struct CSimpleApp : public CWinApp
{
	BOOL InitInstance() { return TRUE; }
};
```

Based on your knowledge of C++, keep in mind that the method could also have been implemented as:

```
struct CSimpleApp : public CWinApp
{
	BOOL InitInstance()
	{
```

```
                        return TRUE;
            }
};
```

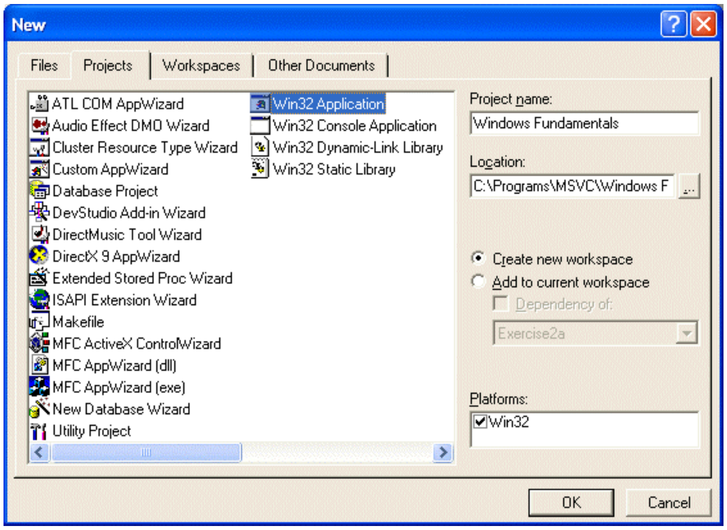
or:

```
struct CSimpleApp : public CWinApp
{
            BOOL InitInstance();
};

BOOL CSimpleApp::InitInstance()
{
            return TRUE;
}
```

Practical Learning: Building a Simple Application

1. Start Microsoft Visual Studio or Microsoft Visual C++
2. On the main menu, click File -> New...
3. In the Project tab, click Win32 Application

   Specify the Name to **Windows Fundamentals**



Project name: Windows Fundamentals

Location: C:\Programs\MSVC\Windows F

● Create new workspace

○ Add to current workspace

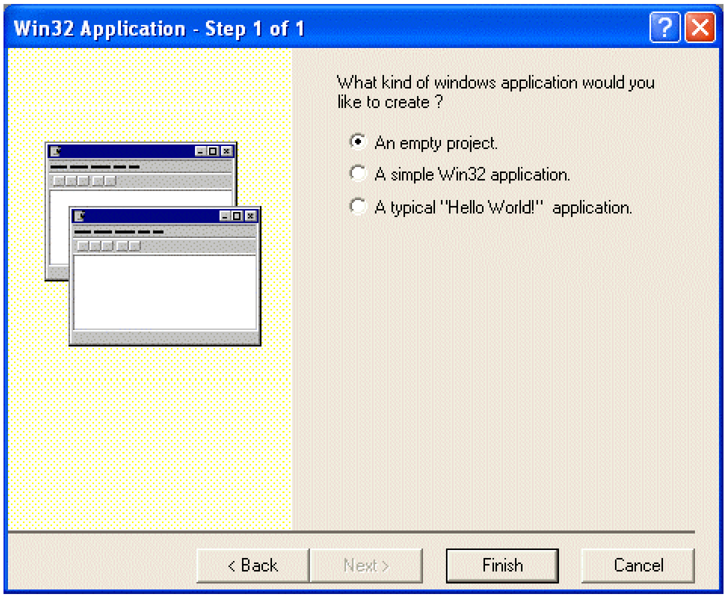
□ Dependency of: Exercise2a

Platforms: ☑ Win32

Figure 16: New Project - Windows Fundamentals

4. Click OK

5. In the Win32 Application – Step 1 of 1, click the An Empty Project radio button



What kind of windows application would you like to create ?

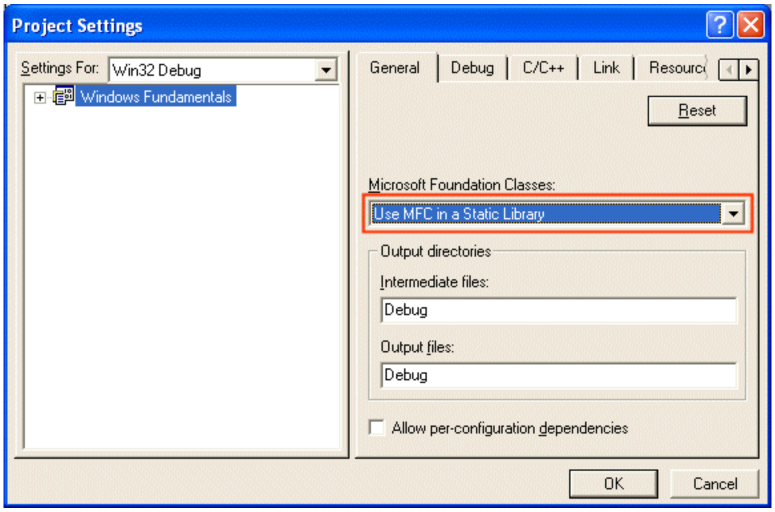
● An empty project.

○ A simple Win32 application.

○ A typical "Hello World!" application.

Figure 17: Win32 Application Wizard - Windows Fundamentals

6. Click Finish and click OK
7. To make sure that this application uses MFC, on the main menu, click Project -> Settings...
8. In Project Settings dialog box, in the General tab, in the Microsoft Foundation Classes combo box, select Use MFC In A Shared DLL **Use MFC in a Shared DLL**



Settings For: Win32 Debug

Windows Fundamentals

Microsoft Foundation Classes: Use MFC in a Static Library

Output directories

Intermediate files: Debug

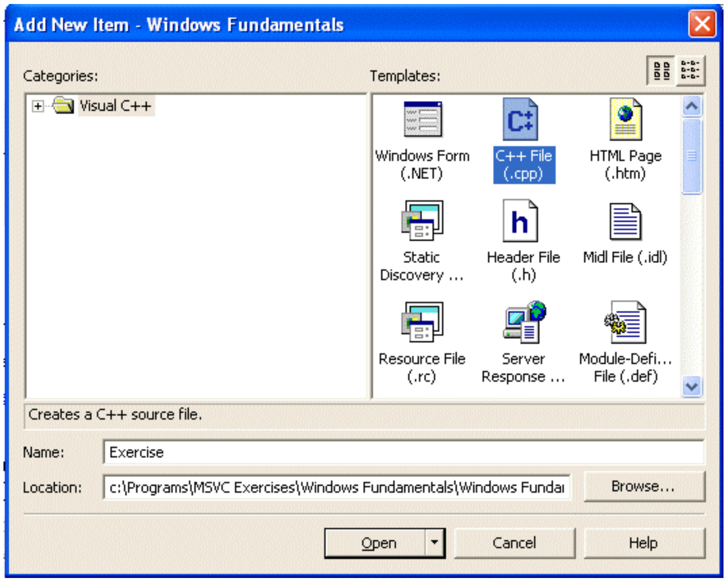
Output files: Debug

□ Allow per-configuration dependencies

Figure 18: Window Frame Property Pages

9. Click OK

10. Create a C++ Source File and Name it **Exercise**



Creates a C++ source file.

Name: Exercise

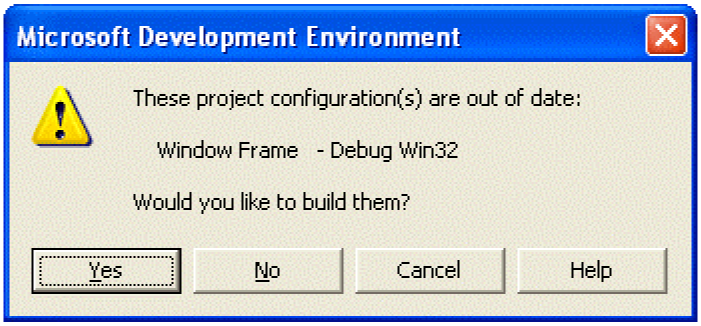
Location: c:\Programs\MSVC Exercises\Windows Fundamentals\Windows Fundai

Figure 19: ADd New Item - Windows Fundamentals

11. Click Open
12. To create the application, change the file as follows:

```
#include <afxwin.h>

struct CSimpleApp : public CWinApp
{
     BOOL InitInstance() { return TRUE; }
};

CSimpleApp theApp;
```

13. To test the application, press Ctrl + F5



These project configuration(s) are out of date:

Window Frame - Debug Win32

Would you like to build them?

Figure 20: Microsoft Development Environment - Message Box

14. When asked whether you would like to build the project, click Yes
15. Nothing will appear because we did not define an expected behavior

2.2 Frames Fundamentals

2.2.1 Introduction

The basic functionality of a window is defined in a class called **CWnd**. An object created from **CWnd** must have a parent. In other words, it must be a secondary window, which is a window called from another, existing, window.

The **CWnd** class gives birth to a class called **CFrameWnd**. This class actually describes a window or defines what a window looks like. The description of a window includes items such as its name, size, location, etc. To create a window using the **CFrameWnd** class, you must create a class derived from the **CFrameWnd** class.

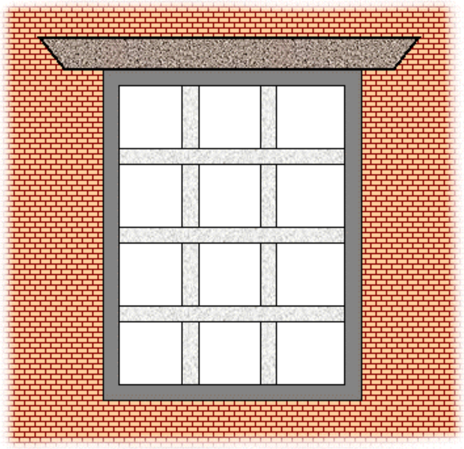


Figure 21: Window Illustration

As in real world, a Microsoft Windows window can be identified on the monitor screen because it has a frame. To create such a window, you can use the **CFrameWnd** class. **CFrameWnd** provides various alternatives to create a window. For example, it is equipped with a method called Create(). The Create() method uses a set of arguments that define a complete window.

The syntax of the **CFrameWnd::Create()** method is as follows:

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

As you can see, only two arguments are necessary to create a window, the others, though valuable, are optional.

As you should have learned from C++, every class in a program has a name. The name lets the compiler know the kind of object to create. The name of a class follows the conventions used for C++ names. That is, it must be a null-terminated string. Many classes used in Win32 and MFC applications are already known to the Visual C++ compiler. If the compiler knows the class you are trying to use, you can specify the *lpszClassName* as **NULL**. In this case, the compiler would use the characteristic of the **CFrameWnd** object as the class' name. Therefore, the **Create()** member function can be implemented as follows:

```
Create(NULL);
```

Every object in a computer has a name. In the same way, a window must have a name. The name of a window is specified as the *lpszWindowName* argument of the **Create()** method. In computer applications, the name of a window is the one that displays on the top section of the window. This is the name used to identify a window or an application. For example, if a window displays Basic Windows Application, then the object is called the "Basic Windows Application Window". Here is an example:

```
Create(NULL, "Basic Windows Application");
```

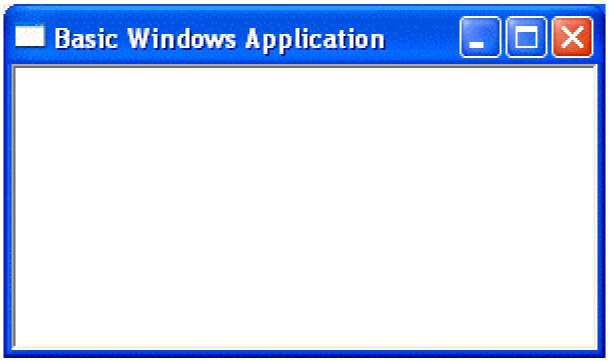


Figure 22: A Window Frame with Title Bar

In all of your windows, you should specify a window name. If you omit it, the users of your window would have difficulty identifying it. Here is an example of such a window:

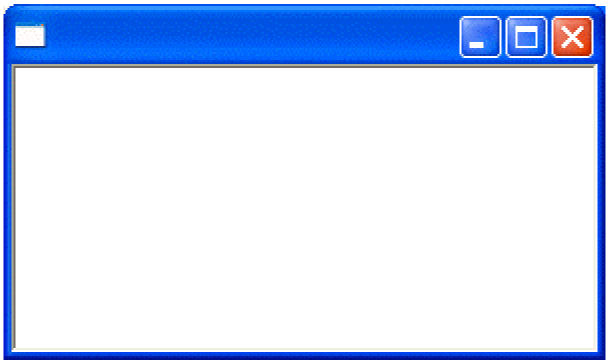


Figure 23: A Window Frame with no Caption

2.2.2 Reference to the Main Window

After creating a window, to let the application use it, you can use a pointer to the class used to create the window. In this case, that would be (a pointer to) **CFrameWnd**. To use the frame window, assign its pointer to the **CWinThread::m_pMainWnd** member variable. This is done in the **InitInstance()** implementation of your application.

At any time, to get a pointer to **m_pMainWnd** anywhere in your program, you can call the **AfxGetMainWnd()** function. Its syntax is:

```
CWnd* AfxGetMainWnd();
```

This function simply returns a pointer to **CWnd**. Because all MFC's window objects are based on the **CWnd** class, this function can give you access to the main class used for the application.

Practical Learning: Creating a Simple Window

1. To create a frame for the window, in the file, type the following:

```
#include <afxwin.h>

class CSimpleFrame : public CFrameWnd
{
public:
    CSimpleFrame()
    {
            // Create the window's frame
            Create(NULL, "Windows Application");
    }
};

struct CSimpleApp : public CWinApp
{
    BOOL InitInstance()
    {
            // Use a pointer to the window's frame for the application
            // to use the window
            CSimpleFrame *Tester = new CSimpleFrame ();
            m_pMainWnd = Tester;

            // Show the window
            m_pMainWnd->ShowWindow(SW_SHOW);
            m_pMainWnd->UpdateWindow();

            return TRUE;
    }
};

CSimpleApp theApp;
```

2. Save the file.
3. To implement the methods outside of their classes, change the file as follows:

```
#include <afxwin.h>

class CSimpleFrame : public CFrameWnd
{
public:
    CSimpleFrame();
};

CSimpleFrame::CSimpleFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application");
}

struct CSimpleApp : public CWinApp
{
    BOOL InitInstance();
};

BOOL CSimpleApp::InitInstance()
{
    // Use a pointer to the window's frame for the application
    // to use the window
    CSimpleFrame *Tester = new CSimpleFrame ();
    m_pMainWnd = Tester;

    // Show the window
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CSimpleApp theApp;
```

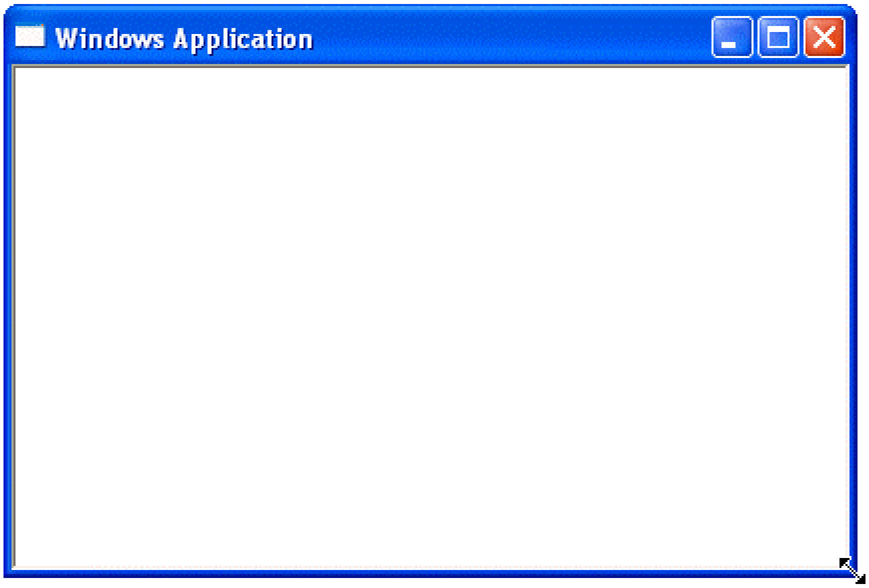


Figure 24: Windows Application

4. Test the program

5. To close the window, click its system Close button ☒ and return to MSVC

6. To provide a checking with the **CObject::AssertValid()** method and to diagnosis services using the **CObject::Dump()** method, change the file as follows:

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

class CSimpleFrame : public CFrameWnd
{
public:
    CSimpleFrame();

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

struct CSimpleApp : public CWinApp
```

```
{
    BOOL InitInstance();
};

CSimpleFrame::CSimpleFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application");
}

// Frame diagnostics

#ifdef _DEBUG
void CSimpleFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CSimpleFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}
#endif // _DEBUG

BOOL CSimpleApp::InitInstance()
{
    CSimpleFrame *Tester = new CSimpleFrame();
    m_pMainWnd = Tester;

    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CSimpleApp theApp;
```

7. Test the program and return to MSVC

2.2.3 Introduction to Macros

When creating objects that derive directly or indirectly from **CObject**, such as **CFrameWnd**, you should let the compiler know that you want your objects to be dynamically created. To do this, use the **DECLARE_DYNCREATE** and the **IMPLEMENT_DYNCREATE** macros. The **DECLARE_DYNCREATE** macro is provided in the class' header file and takes as argument the name of your class. An example would be **DECLARE_DYNCREATE**(CTheFrame). Before implementing the class or in its source file, use the **IMPLEMENT_DYNCREATE** macro, passing it two arguments: the name of your class and the name of the class you derived it from.

We mentioned that, to access the frame, a class derived from **CFrameWnd**, from the application class, you must use a pointer to the frame's class. On the other hand, if you want to access the application, created using the **CWinApp** class, from the windows frame, you use the **AfxGetApp()** global function.

Practical Learning: Creating a Simple Window

1. To use the dynamic macros, change the file as follows:

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

class CSimpleFrame : public CFrameWnd
{
public:
    CSimpleFrame();
    DECLARE_DYNCREATE(CSimpleFrame)

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

struct CSimpleApp : public CWinApp
{
    BOOL InitInstance();
};

IMPLEMENT_DYNCREATE(CSimpleFrame, CFrameWnd)

CSimpleFrame::CSimpleFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application");
}

    . . .

CSimpleApp theApp;
```

2. Test the application and return to MSVC.

2.2.4 Windows Styles

Windows styles are characteristics that control such features as its appearance, its borders, its minimized, its maximized, or its resizing states, etc.

After creating a window, to display it to the user, you can apply the **WS_VISIBLE** style to it. Here is an example:

```
Create(NULL, "Windows Application", WS_VISIBLE);
```

The top section of a window has a long bar called the title bar. This is a section we referred to above and saw that it is used to display the name of the window. This bar is added with the **WS_CAPTION** value for the style argument. Here is an example:

```
Create(NULL, "Windows Application", WS_CAPTION);
```

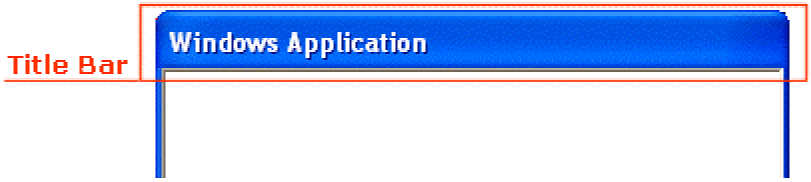


Figure 25: The Title Bar of a Frame

A window is identified by its size which can be seen by its borders. Borders are given to a window through the **WS_BORDER** value for the style. As you can see from the above window, using the **WS_CAPTION** style also gives borders to a window.

A shortcut to creating a window that has a caption and a border is done by using the **WS_OVERLAPPED** style:

```
Create(NULL, "Windows Application", WS_OVERLAPPED);
```

As you will see shortly, various styles can be applied to the same window. To combine styles, you use the bitwise OR operator represented by the beam symbol |. For example, if you have a style called WS_ONE and you want to combine it with the WS_OTHER style, you can use WS_ONE | WS_OTHER.

The title bar serves various purposes. For example, it displays the name of the window. In this case the window is called Window Application. The title bar also allows the user to move a window. To do this, the user would click and drag a section of the title bar while moving in the desired direction.

After using a window, the user may want to close it. This is made possible by using or adding the **WS_SYSMENU** style. This style adds a button with X that is called the System Close button ☒ or ☒. Here is an example:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU);
```

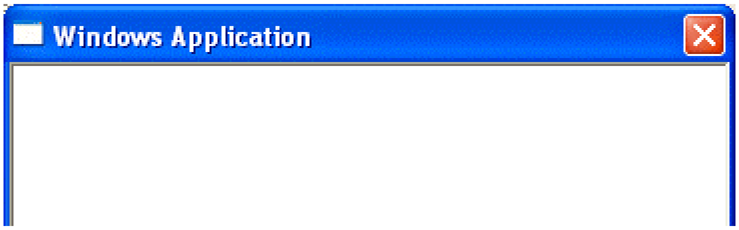


Figure 26: A Frame with System Menu

When a window displays, it occupies a certain area of the screen. To access other windows, for example to reveal a window hidden behind, the user can shrink the size of the window completely and make it disappear from the screen without closing the window. This is referred to as minimizing the window. The ability to minimize a window is made possible with the presence of a system button called Minimize ▁ or ▁. To allow this, you provide or add the **WS_MINIMIZEBOX** style:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU |
WS_MINIMIZEBOX);
```

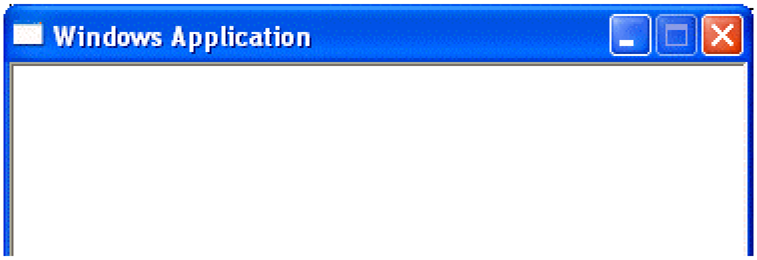


Figure 27: A Frame with a Minimize Button

As opposed to minimizing the window, the user can change its size to use the whole area of the screen. This is referred to as maximizing the window. This feature is provided through the system Maximize button ▣ or ▣. To allow this functionality, provide or add the **WS_MAXIMIZEBOX** style.

If you create a window with the **WS_SYSMENU** and the **WS_MINIMIZEBOX** style but not the **WS_MAXIMIZEBOX** style, the Maximize button would be disabled and the user would not be able to maximize the window:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU |
WS_MINIMIZEBOX);
```

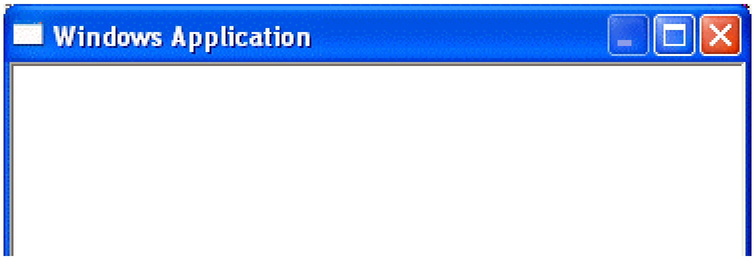


Figure 28: A Frame with a Maximize Button

On the other hand, if you do not want the user to be able to minimize the window, create a window with the **WS_SYSMENU** and the **WS_MINIMIZEBOX** without the **WS_MAXIMIZEBOX** styles:

```
Create(NULL, "Windows Application", WS_VISIBLE | WS_SYSMENU | WS_MAXIMIZEBOX);
```

Therefore, if you want to provide all three system buttons, provide their styles.

You can also control whether the window appears minimized or maximized when it comes up. To minimize the window at startup, apply the **WS_MINIMIZE** style. On the other hand, to have a maximized window when it launches, use the **WS_MAXIMIZE** style. This style also provides a window with borders. If you had not specified the **WS_CAPTION**, you can make sure that a window has borders using the **WS_BORDER** value.

One of the effects the user may want to control on a window is its size. For example, the user may want to narrow, enlarge, shrink, or heighten a window. To do this, the user would position the mouse on one of the borders, click and drag in the desired direction. This action is referred to as resizing a window. For the user to be able to change the size of a window, the window must have a special type of border referred to as a thick frame. To provide this border, apply or add the **WS_THICKFRAME** style:

```
Create(NULL, "Windows Application",
        WS_VISIBLE | WS_SYSMENU | WS_MINIMIZEBOX |
        WS_MAXIMIZEBOX |    WS_THICKFRAME);
```

Because many windows will need this functionality, a special style can combine them and it is called **WS_OVERLAPPEDWINDOW**. Therefore, you can create a resizable window as follows:

```
Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW);
```

Practical Learning: Building a Regular Window

1. To create a resizable window, change the file as follows:

```
CSimpleFrame::CSimpleFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW);
}
```

2. Test the program. When the window comes up, minimize, maximize, restore, and resize it:

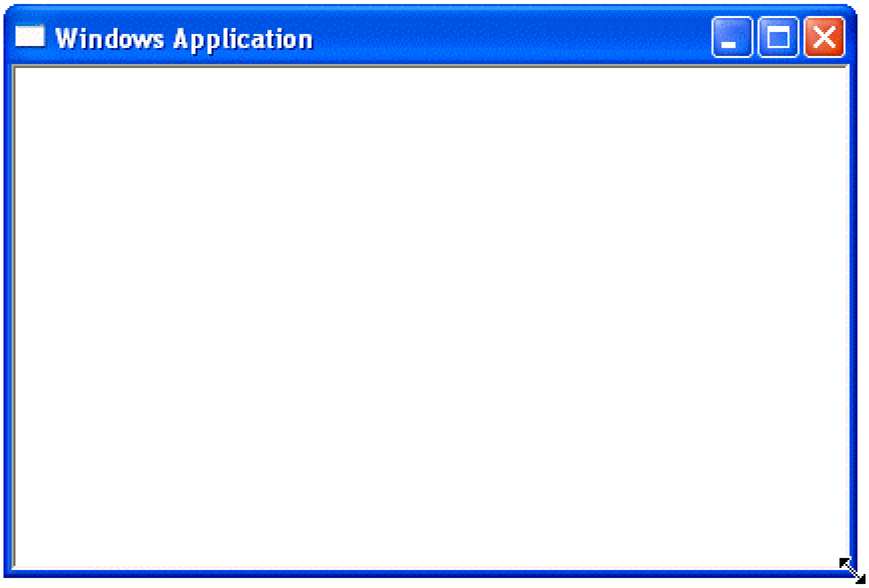


Figure 29: Windows Application - Resizing

3. To close the window, click its system Close button

2.2.5 Windows Location

To locate things that display on the monitor screen, the computer uses a coordinate system similar to the Cartesian's but the origin is located on the top left corner of the screen. Using this coordinate system, any point can be located by its distance from the top left corner of the screen of the horizontal and the vertical axes:

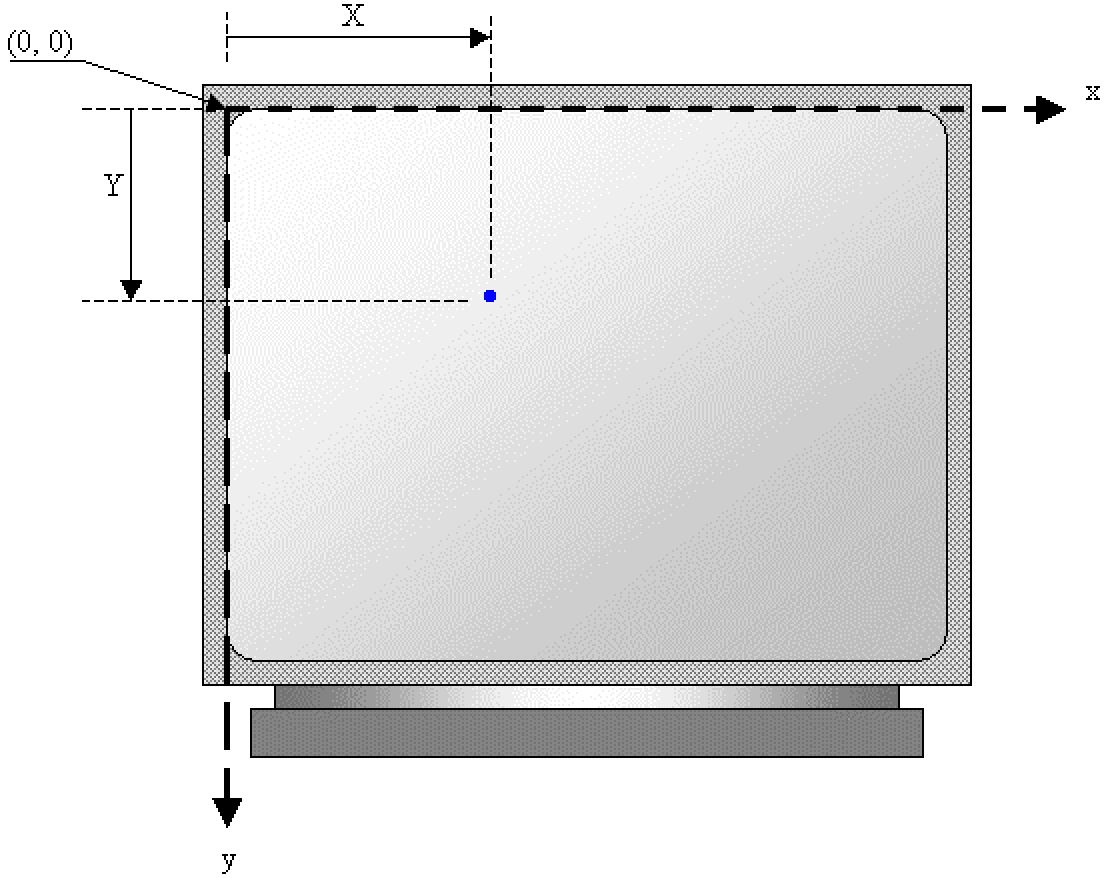


Figure 30: Illustration of Window Origin

To manage such distances, the operating system uses a point that is represented by the horizontal and the vertical distances. The Win32 library provides a structure called POINT and defined as follows:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT;
```

The x member variable is the distance from the left border of the screen to the point. The y variable represents the distance from the top border of the screen to the point.

Besides the Win32's **POINT** structure, the Microsoft Foundation Class (MFC) library provides the **CPoint** class. This provides the same functionality as the **POINT** structure. As a C++ class, it adds more functionality needed to locate a point.

The **CPoint::CPoint() default constructor** can be used to declare a point variable without specifying its location. If you know the x and y coordinates of a point, you can use the following constructor to create a point:

```
CPoint(int X, int Y);
```

2.2.6 Windows Size

While a point is used to locate an object on the screen, each window has a size. The size provides two measurements related to an object. The size of an object can be represented as follows:

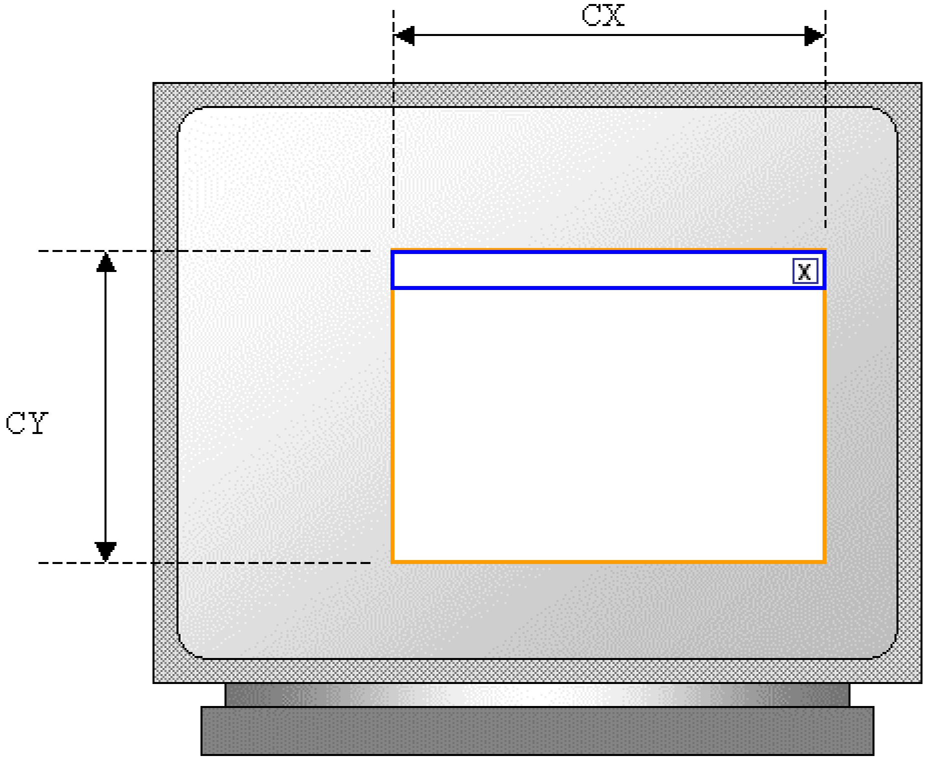


Figure 31: Illustration of Window Location

The width of an object, represented as CX, is the distance from its left to its right borders and is provided in pixels.

The height of an object, represented as CY is the distance from its top to its bottom borders and is given in pixels.

To represent these measures, the Win32 library uses the **SIZE** structure defined as follows:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

Besides the Win32's **SIZE** structure, the MFC provides the **CSize** class. This class has the same functionality as **SIZE** but adds features of a C++ class. For example, it provides five constructors that allows you to create a size variable in any way of your choice. The constructors are:

```
CSize();
CSize(int initCX, int initCY);
CSize(SIZE initSize);
CSize(POINT initPt);
CSize(DWORD dwSize);
```

The default constructor allows you to declare a **CSize** variable whose values are not yet available. The constructor that takes two arguments allows you to provide the width and the height to create a **CSize** variable. If you want another **CSize** or a **SIZE** variables, you can use the **CSize(SIZE initSize)** constructor to assign its values to your variable. You can use the coordinates of a **POINT** or a CPoint variable to create and initialize a **CSize** variable. When we study the effects of the mouse, we will know you can use the coordinates of the position of a mouse pointer. These coordinates can help you define a **CSize** variable.

Besides the constructors, the **CSize** class is equipped with different methods that can be used to perform various operations on **CSize** and **SIZE** objects. For example, you can add two sizes to get a new size. You can also compare two sizes to find out whether they are the same.

2.2.7 Windows Dimensions

When a window displays, it can be identified on the screen by its location with regards to the borders of the monitor. A window can also be identified by its width and height. These characteristics are specified or controlled by the *rect* argument of the **Create()** method. This argument is a rectangle that can be created through the Win32 **RECT** structure or the MFC's **CRect** class. First, you must know how Microsoft Windows represents a rectangle.

A rectangle is a geometric figure with four sides or borders: top, right, bottom, and left. A window is also recognized as a rectangle. Therefore, it can be represented as follows:

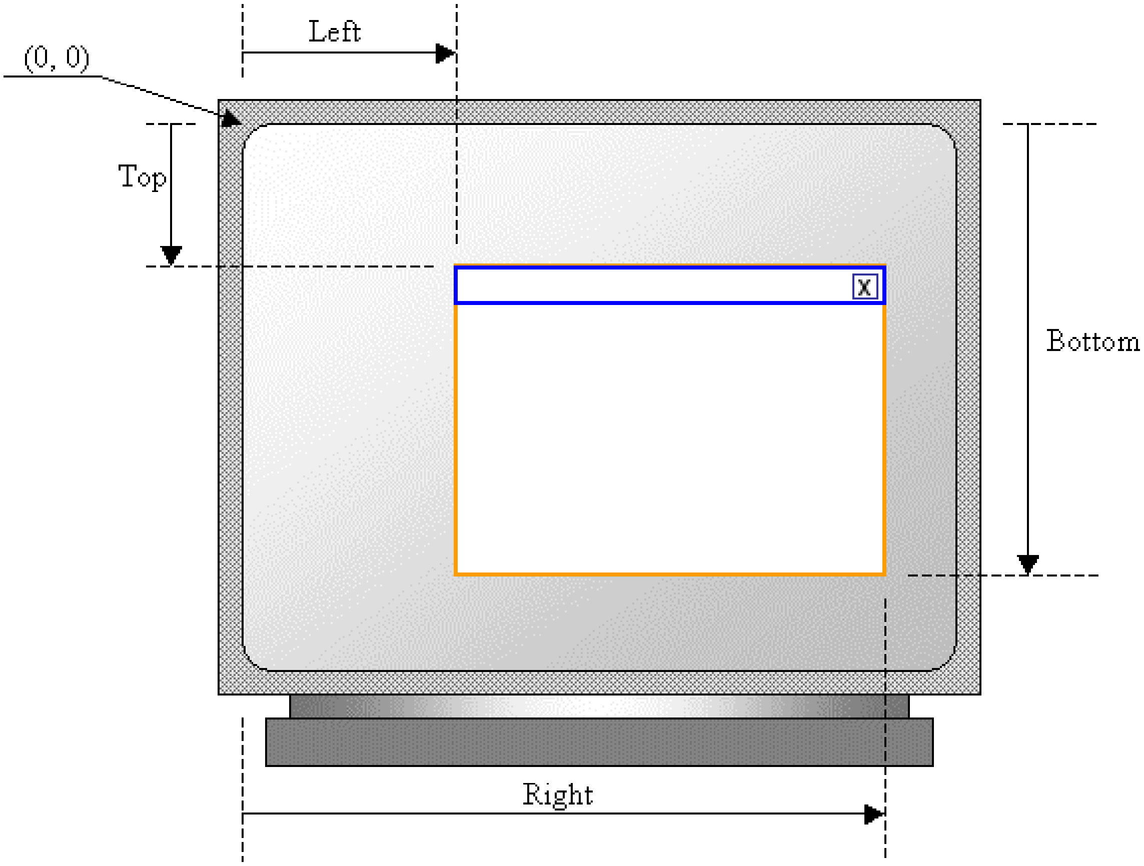


Figure 32: Illustration of Window Origin and Location

Microsoft Windows represents items as a coordinate system whose origin (0, 0) is located on the top-left corner of the screen. Everything else is positioned from that point. Therefore:

?? the distance from the left border of the screen to left border of a window represents the left measurement

?? the distance from the top border of the screen to the top border of a window is the top measurement.

?? the distance from the left border of the screen to the right border of a window represents the right measurement

?? the distance from the top border of the screen to the bottom border of a window is the top measurement.

To represent these distances, the Win32 API provides a structure called RECT and defined as follows:

```
typedef struct _RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT;
```

This structure can be used to control the location of a window on a coordinate system. It can also be used to control or specify the dimensions of a window. To use it, specify a natural numeric value for each of its member variables and pass its variable as the rect argument of the **Create()** method. Here is an example:

```
RECT Recto;

Recto.left   = 100;
Recto.top    = 120;
Recto.right  = 620;
Recto.bottom = 540;

Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW, Recto);
```

Besides the Win32 **RECT** structure, the MFC provides an alternate way to represent a rectangle on a window. This is done as follows:

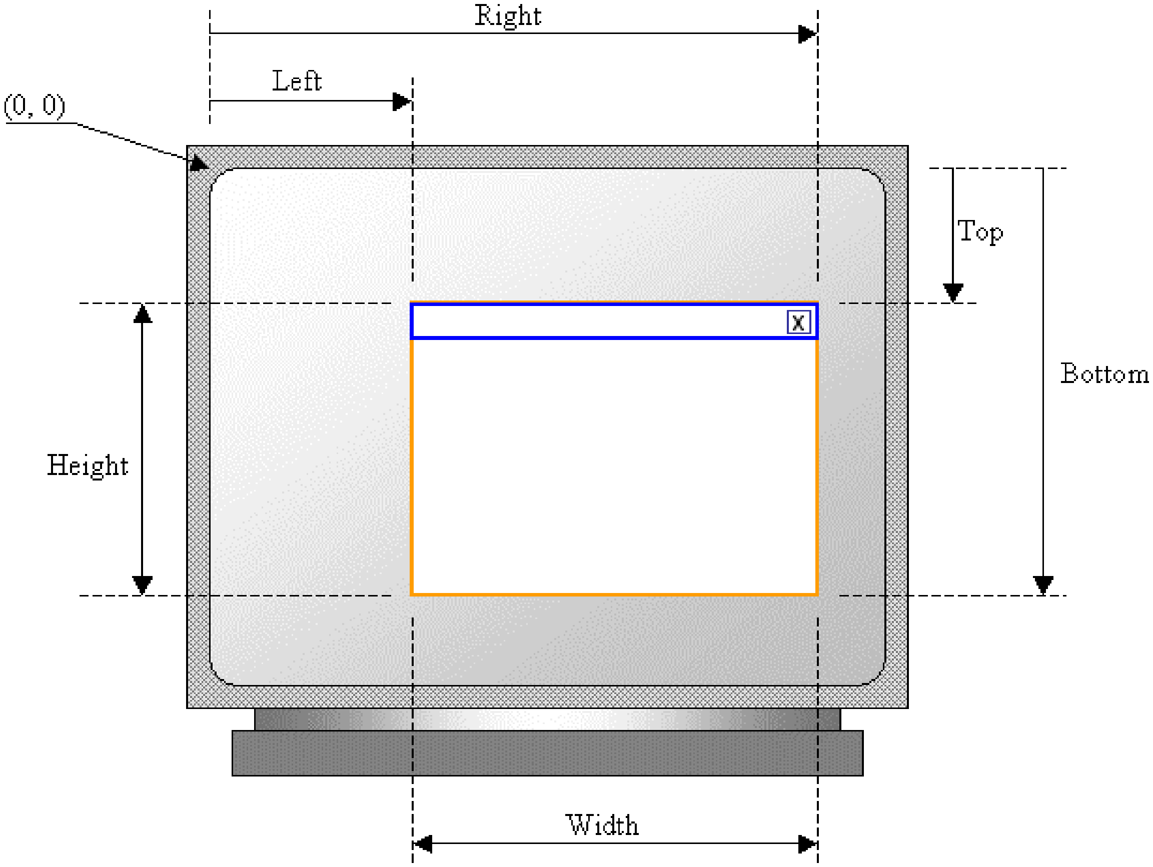


Figure 33: Illustration of Window Location, Origin, and Size

Besides the left, top, right, and bottom measurements, a window is also recognized for its width and its height. The width is the distance from the left to the right borders of a rectangle. The height is the distance from the top to the bottom borders of a rectangle. To recognize all these measures, the Microsoft Foundation Classes library provides a class called **CRect**. This class provides various constructors for almost any way of representing a rectangle. The CRect class provides the following constructors:

```
CRect();
CRect(int l, int t, int r, int b);
CRect(const RECT& srcRect);
CRect(LPCRECT lpSrcRect);
CRect(POINT point, SIZE size);
CRect(POINT topLeft, POINT bottomRight);
```

The default constructor is used to declare a **CRect** variable whose dimensions are not known. On the other hand, if you know the left, the top, the right, and the bottom measures of the rectangle, as seen on the **RECT** structure, you can use them to declare and initialize a rectangle. In the same way, you can assign a **CRect**, a **RECT**, a pointer to **CRect**, or a pointer to a **RECT** variable to create or initialize a CRect instance.

If you do not know or would not specify the location and dimension of a window, you can specify the value of the rect argument as *rectDefault*. In this case, the compiler would use its own internal value for this argument.

Practical Learning: Controlling a Window's Size

1. To specify the location and the size of the window, change the MainFrm.cpp file as follows:

```
CSimpleFrame::CSimpleFrame()
{
	// Create the window's frame
	Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
		CRect(120, 100, 700, 480));
}
```

2. Test the application. After viewing the window, close it and return to your programming environment.

2.2.8 Windows Parents

Many applications are made of different windows, as we will learn eventually. When an application uses various windows, most of these objects depend on a particular one. It could be the first window that was created or another window that you designated. Such a window is referred to as the parent window. All the other windows depend on it directly or indirectly.

If the window you are creating is dependent of another, you can specify that it has a parent. This is done with the *pParentWnd* argument of the **CFrameWnd::Create()** method. If the window does not have a parent, pass the argument with a **NULL** value.

Practical Learning: Specifying a Window's Parent

1. To specify that the current window does not have a parent, change the **Create()** method as follows:

```
CSimpleFrame::CSimpleFrame()
{
	// Create the window's frame
	Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
	       CRect(120, 100, 700, 480), NULL);
}
```

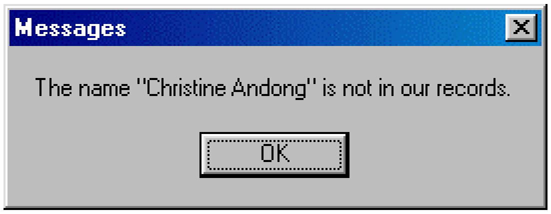
2. Test the application and return to Visual Studio

2.3 Message Boxes

2.3.1 Definition

A message box is a bordered rectangular window that displays a short message to the user. The message can be made of one sentence, various lines, or a few paragraphs. The user cannot change the text but can only read. A message box is created either to inform the user about something or to request his or her decision about an issue. When a dialog box displays from an application, the user must close it before continuing using the application. This means that, as long as the message box is displaying, the user cannot access any other part of the application until the message box is first dismissed.

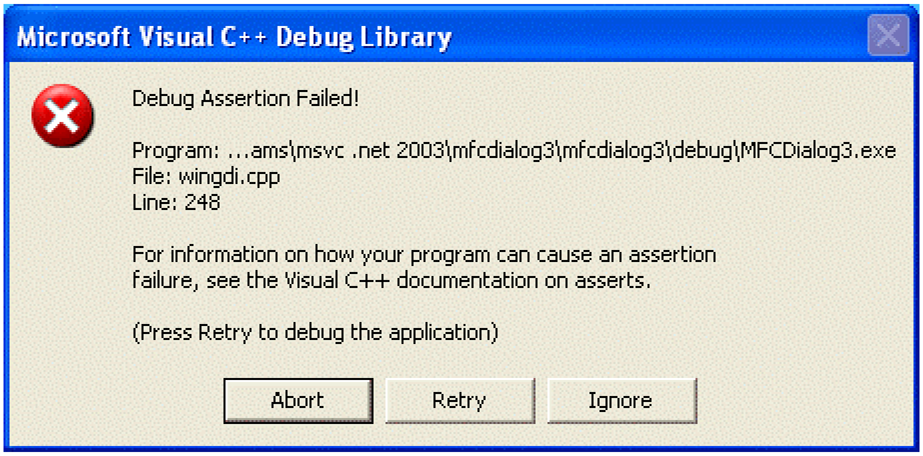
The simplest message box is used to display a message to the user. This type of message box is equipped with only one button, labelled OK. Here is an example:



The name "Christine Andong" is not in our records.

Figure 34: A Simple Message Box

A more elaborate or detailed message box can display an icon and/or can have more than one button.



Debug Assertion Failed!

Program: ...ams\msvc .net 2003\mfcdialog3\mfcdialog3\debug\MFCDialog3.exe
File: wingdi.cpp
Line: 248

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)

Figure 35: An Elaborate Message Box

2.3.2 Message Box Creation

To create a message box, the Win32 library provides a global function called **MessageBox**. Its syntax is:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

If you want to use this version from Win32, because it is defined outside of MFC, you should start it with the scope access operator "::" as follows:

```
::MessageBox(...);
```

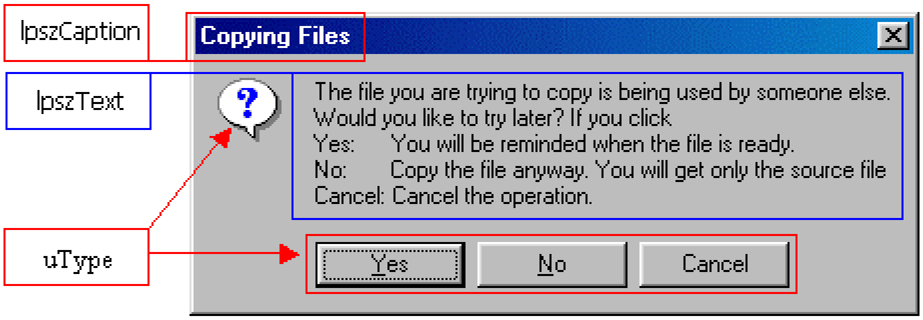
As we will see shortly, the Win32's **MessageBox()** function requires a handle (not necessarily difficult to provide but still…), to make the creation of a message a little easier, the **CWnd** class provides its own version of the **MessageBox()** function. Its syntax is:

```
int MessageBox(LPCTSTR lpszText, LPCTSTR lpszCaption = NULL, UINT nType = MB_OK);
```

To still make it easier, Visual C++ (actually a concept called the Framework) provides a function called **AfxMessageBox** also used to create a message box. Although this function also is global, its scope is limited to MFC applications (this means that you can use the Win32's global **MessageBox()** function with any compiler used on the Microsoft Windows operating systems but you cannot use the MFC's global **AfxMessageBox()** function with any compiler other Visual C++). The syntaxes of the **AfxMessageBox()** function are:

```
int AfxMessageBox(LPCTSTR lpszText, UINT nType = MB_OK, UINT nIDHelp = 0);
int AFXAPI AfxMessageBox(UINT nIDPrompt, UINT nType = MB_OK,
                                    UINT nIDHelp = (UINT) -1);
```

Based on the above functions, a message box can be illustrated as follows:



The file you are trying to copy is being used by someone else.
Would you like to try later? If you click
Yes: You will be reminded when the file is ready.
No: Copy the file anyway. You will get only the source file
Cancel: Cancel the operation.

Figure 36: Components of a Message Box

2.3.3 Message Box Implementation

As seen above, you have the choice among three functions to create a message. There is no valid reason that makes one of them better than the other. They do exactly the same thing. The choice will be based on your experience and factors beyond our understanding.

If you decide to use the Win32's **MessageBox()** function, you must specify the handle to the application that created the message box. As we will learn eventually when we study controls, you can get a handle to a (**CWnd**-derived) control with a call to **m_hWnd**. For example, if a button on a dialog box initiates the message box, you can start this function as follows:

::MessageBox(**m_hWnd**, …);

We also saw that you can get a pointer to the main window by calling the MFC's global **AfxGetMainWnd()** function. This function only points you to the application. To get a handle to the application, you can call the same **m_hWnd** member variable. In this case, the message box can be started with:

::MessageBox(**AfxGetMainWnd()->m_hWnd**, …);

If you are creating a message but do not want a particular window to own it, pass this *hWnd* argument as **NULL**.

Practical Learning: Using a Message Box

1. Create a new Win32 Project name it **MsgBox**
2. Create it as a **Windows Application Empty Project**
3. Specify that you want to **Use MFC in a Shared DLL**
4. Create a C++ file and name it **Exercise**
5. In the empty file, initialize the application as follows:

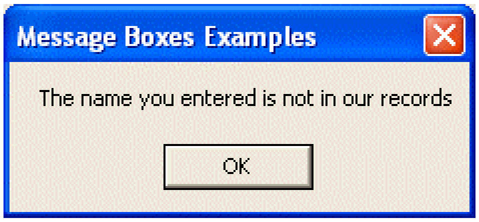
```
#include <afxwin.h>

class CExerciseApp : public CWinApp
{
public:
	BOOL InitInstance();
};

CExerciseApp theApp;

BOOL CExerciseApp::InitInstance()
{
	return TRUE;
}
```

6. Save All

2.3.4 The Box' Message

For all these functions, the *Text* argument a null-terminated string. It specifies the text that would be displayed to the user. This argument is required for all these functions. Here is an example that use the **CWnd::MessageBox()** function:

```
void Whatever()
{
		// TODO: Add your control notification handler code here
		MessageBox("The name you entered is not in our records");
}
```

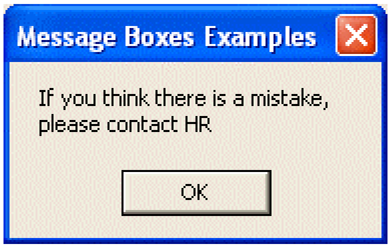


The name you entered is not in our records

Figure 37: A Simple Message Box with MessageBox()

If you want to display the message on various lines, you can separate sections with the new line character '\n'. Here is an example:

```
void Whatever()
{
        // TODO: Add your control notification handler code here
        MessageBox("If you think there is a mistake,\nplease contact HR");
}
```



If you think there is a mistake,
please contact HR

Figure 38: A Message's Box on Various Lines

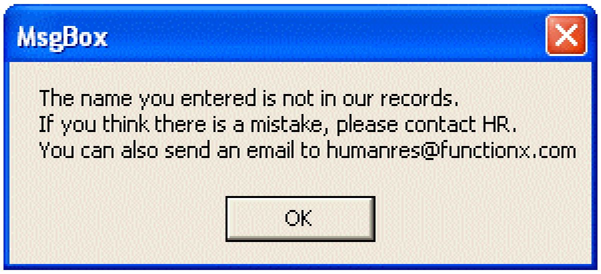
If the message you are creating is too long to fit on one line, you can separate lines by ending each with a double-quote and starting the next line with a new qouble-quote. As long as you have not closed the function, the string would be considered as one entity. You can also use string editing and formatting techniques to create a more elaborate message. This means that you can use functions of the C string library to create your message.

Practical Learning: Displaying a Message

1. To prepare a message for the user, in the **InitInstance()** member function, call the **AfxMessageBox()** function as follows:

```
#include <afxwin.h>

class CExerciseApp : public CWinApp
{
public:
    BOOL InitInstance();
};

CExerciseApp theApp;

BOOL CExerciseApp::InitInstance()
{
    CWnd *Msg = new CWnd;
    Msg->MessageBox("The name you entered is not in our records.\n"
                                "If you think there is a mistake, please contact HR.\n"
                                "You can also send an email to humanres@functionx.com");

        return TRUE;
}
```

2. Test the application



The name you entered is not in our records.
If you think there is a mistake, please contact HR.
You can also send an email to humanres@functionx.com

Figure 39: Creating a Multiline Message Box

3. Click OK to close the message box and return to MSVC.

2.3.5 The Message' Title

The caption of the message box is the text that displays on its title bar. If you create a message box using the **CWnd::MessageBox()** method or the **AfxMessageBox()** function, which allow you to specify only one argument, namely the text to display to the user, which is the value of the *Text* argument, the title bar of the message box would display the name of the application that "owns" the message box. If you want to display a more meaningful title, specify the second argument.

The *Caption* argument is a null-terminated string that would display on the title bar of the message box. Here is an example:

```
void CDialog1Dlg::OnBnClickedBtnMsg1()
{
        // TODO: Add your control notification handler code here
        MessageBox("Due to an unknown internal error, this application will now close.",
                      "Regular Warning");
}
```

Although the *Caption* argument is optional for the **CWnd::MessageBox()** method and the **AfxMessageBox()** function, it is required for the Win32's **MessageBox()** function. Because it is in fact a pointer, you can pass it as **NULL**. In this case, the title bar of the message box would display Error. This caption may not be friendly on most application and could appear freightening to the user. Therefore, unless you are in a hurry, you should strive to provide a friendly and more appropriate title.

Practical Learning: Displaying a Message's Caption

1. To display a caption for the message box, change the MessageBox() call as follows:

```
BOOL CExerciseApp::InitInstance()
{
        CWnd *Msg = new CWnd;
        Msg->MessageBox("The name you entered is not in our records.\n"
                        "If you think there is a mistake, please contact HR.\n"
                        "You can also send an email to humanres@functionx.com",
                         "Failed Logon Attempt");

    return TRUE;
}
```

2. Test the application and return to MSVC

2.3.6 Message Box Options

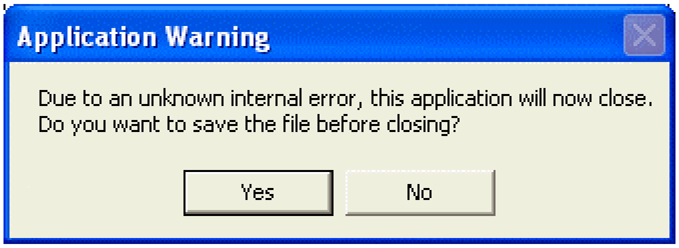
The *uType* argument is used to provide some additional options to the message box. First, it is used to display one or a few buttons. The buttons depend on the value specified for the argument. If this argument is not specified, the message box displays (only) OK. Otherwise, you can display a message box with a combination of selected buttons.

To display one or more buttons, the *uType* argument uses a constant value that controls what button(s) to display. The values and their buttons can be specified as follows:

Constant	Buttons
MB_OK	OK
MB_OKCANCEL	OK Cancel
MB_ABORTRETRYIGNORE	Abort Retry Ignore
MB_YESNOCANCEL	Yes No Cancel
MB_YESNO	Yes No
MB_RETRYCANCEL	Retry Cancel
MB_HELP	Help

Here is an example:

```
BOOL Whatever()
{
    MessageBox("Due to an unknown internal error, this application will now close.\n"
               "Do you want to save the file before closing?",
                "Application Warning",
                MB_YESNO);
}
```


Application Warning

Due to an unknown internal error, this application will now close.
Do you want to save the file before closing?

Figure 40: A Message Box with Various Buttons

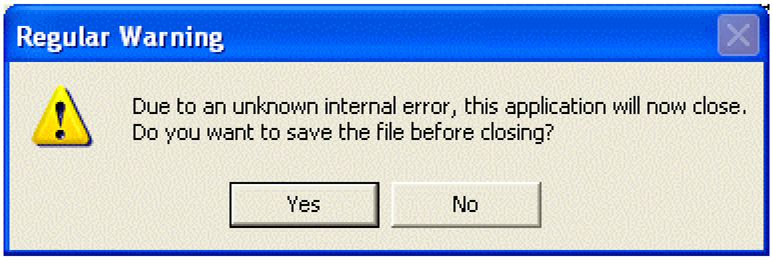
Besides the buttons, the message box can also display an icon that accompanies the message. Each icon is displayed by specifying a constant integer. The values and their buttons are as follows:

Value	Icon	Suited when

MB_ICONEXCLAMATION MB_ICONWARNING	⚠	⚠	Warning the user of an action performed on the application
MB_ICONINFORMATION MB_ICONASTERISK	i	i	Informing the user of a non-critical situation
MB_ICONQUESTION	?	?	Asking a question that expects a Yes or No, or a Yes, No, or Cancel answer
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	✖	✖	A critical situation or error has occurred. This icon is appropriate when informing the user of a termination or deniability of an action

The icons are used in conjunction with the buttons constant. To combine these two flags, use the bitwise OR operator “|”. Here is an examp le:

```
void CExerciseDlg::OnMsgBox()
{
	// TODO: Add your control notification handler code here
	MessageBox("Due to an unknown internal error, this application will now close.\n"
	           "Do you want to save the file before closing?",
	           "Regular Warning",
	            MB_YESNO | MB_ICONWARNING);
}
```



Regular Warning

Due to an unknown internal error, this application will now close.
Do you want to save the file before closing?

Figure 41: A Message Box with an Icon

When a message box is configured to display more than one button, the operating system is set to decide which button is the default. The default button has a thick border that sets it apart from the other button(s). If the user presses Enter, the message box would behave as if the user had clicked the default button. Fortunately, if the message box has more than one button, you can decide what button would be the default.

To specify the default button, add one of the following constants to the *uType* combination:

Value	If the message box has more than one button, the default button would be
MB_DEFBUTTON1	The first button
MB_DEFBUTTON2	The second button
MB_DEFBUTTON3	The third button
MB_DEFBUTTON4	The fourth button

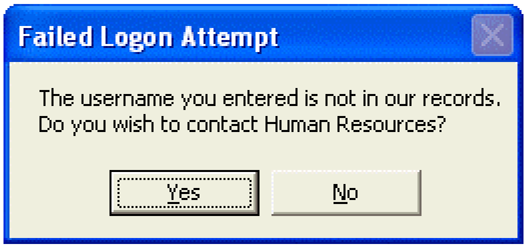
To specify the default button, use the bitwise OR operator "|" to combine the constant integer of the desired default button with the button's constant and the icon.

Practical Learning: Using a Message Box

1. To display an combination of buttons on the message box, change the **InitInstance**() member function as follows:

```
BOOL CExerciseApp::InitInstance()
{
	::MessageBox(NULL,
	             "The username you entered is not in our records.\n"
	             "Do you wish to contact Human Resources?",
	             "Failed Logon Attempt",
	             MB_YESNO);

	return TRUE;
}
```

2. Test the application



The username you entered is not in our records.
Do you wish to contact Human Resources?

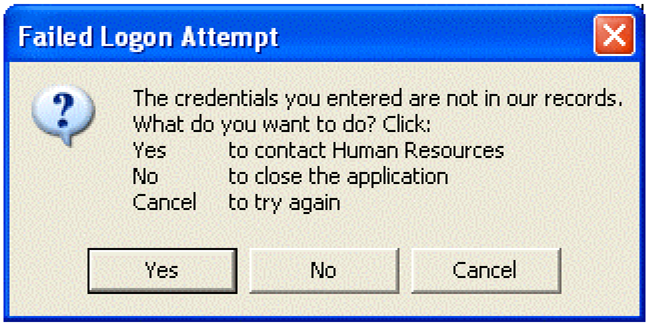
Figure 42: Creating a Message Box

3. Close it and return to MSVC

4. To display the message box with an icon, change **InitInstance()** as follows:

```
BOOL CExerciseApp::InitInstance()
{
	CWnd *Msg = new CWnd;

	Msg->MessageBox("The credentials you entered are not in our records.\n"
	                "What do you want to do? Click:\n"
	                 "Yes\tto contact Human Resources\n"
	                "No\tto close the application\n"
	                "Cancel\tto try again\n",
	                "Failed Logon Attempt",
	                MB_YESNOCANCEL | MB_ICONQUESTION);

	return TRUE;
}
```

5. Test the application



Failed Logon Attempt

The credentials you entered are not in our records.
What do you want to do? Click:
Yes to contact Human Resources
No to close the application
Cancel to try again

Figure 43: Creating an Elaborate Message Box

6. Close it and return to MSVC

2.3.7 The Message's Return Value

After using the message box, the user must close it by clicking a button on it. Clicking OK usually means that the user acknowledged the message. Clicking Cancel usually means the user is changing his or her mind about the action performed previously. Clicking Yes instead of No usually indicates that the user agrees with whatever is going on.

In reality, the message box only displays a message and one or a few buttons. The function used to create the message box returns a natural number that you can use as you see fit. The return value itself is a registered constant integer and can be one of the following:

Displayed Button(s)	If the user clicked	The return value is
OK	OK	IDOK
OK Cancel	OK	IDOK
OK Cancel	Cancel	IDCANCEL
Abort Retry Ignore	Abort	IDABORT
Abort Retry Ignore	Retry	IDRETRY
Abort Retry Ignore	Ignore	IDIGNORE
Yes No Cancel	Yes	IDYES
Yes No Cancel	No	IDNO
Yes No Cancel	Cancel	IDCANCEL
Yes No	Yes	IDYES
Yes No	No	IDNO
Retry Cancel	Retry	IDRETRY
Retry Cancel	Cancel	IDCANCEL

Here is an example:

```
void CExerciseDlg::OnMsgBox()
{
        // TODO: Add your control notification handler code here
        int Answer;

        Answer = MessageBox("Due to an unknown internal error, "
                            "this application will now close.\n"
                            "Do you want to save the file before closing?",
                            "Regular Warning",
                            MB_YESNO | MB_ICONWARNING | MB_DEFBUTTON2);

        if( Answer == IDNO )
                return;
}
```

Chapter 3: Windows Resources

✍ Introduction to Resources

✍ Icons

✍ Menu Fundamentals

☞ Toolbars

☞ Accelerators

☞ Version Information

☞ Cursors

☞ String Tables

3.1 Introduction to Resources

3.1.1 Introduction

A resource is a text file that allows the compiler to manage such objects as pictures, sounds, mouse cursors, dialog boxes, etc. Microsoft Visual C++ makes creating a resource file particularly easy by providing the necessary tools in the same environment used to program, meaning you usually do not have to use an external application to create or configure a resource file.

Although an application can use various resources that behave independently of each other, these resources are grouped into a text file that has the .rcextension. You can create this file manually and fill out all necessary parts but it is advantageous to let Visual C++ created it for you. To do this, you add or create one resource at a time when designing it. After saving a resource, it is automatically added to the .rc file. To make your resource recognizable to the other files of the program, you must also create a header file usually called resource.h. This header file must provide a constant integer that identifies each resource and makes it available to any part that needs it. This also means that most, if not all, of your resources will be represented by an identifier.

Because resources are different entities, they are created one at a time. They can also be imported from existing files. Most resources are created by selecting the desired one from the Add Resource dialog box:

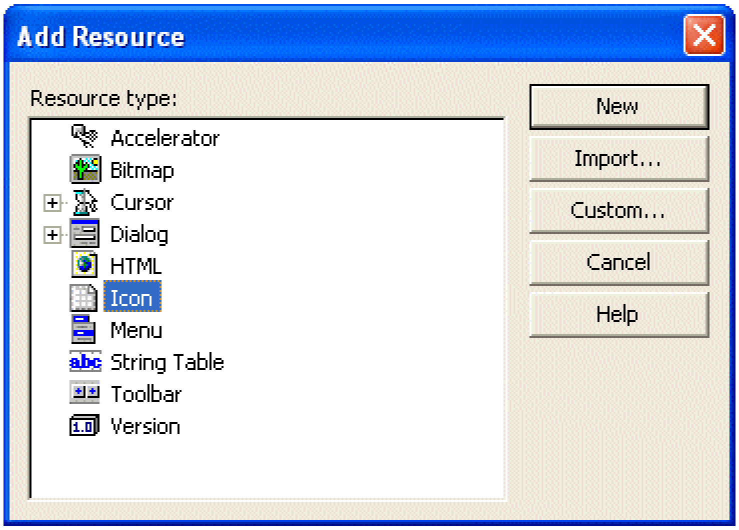


Figure 44: Add Resource - Icon

The Add Resource dialog box provides an extensive list of resources to accommodate almost any need. Still, if you do not see a resource you need and know you can use it, you can add it manually to the .rc file before executing the program.

3.1.2 Converting a Resource Identifier

An identifier is a constant integer whose name usually starts with ID. Although in Win32 programming you usually can use the name of a resource as a string, in MFC applications, resources are usually referred to by their identifier. To make an identifier name (considered a string) recognizable to an MFC (or Win32) function, you use a macro called **MAKEINTERESOURCE**. Its syntax is:

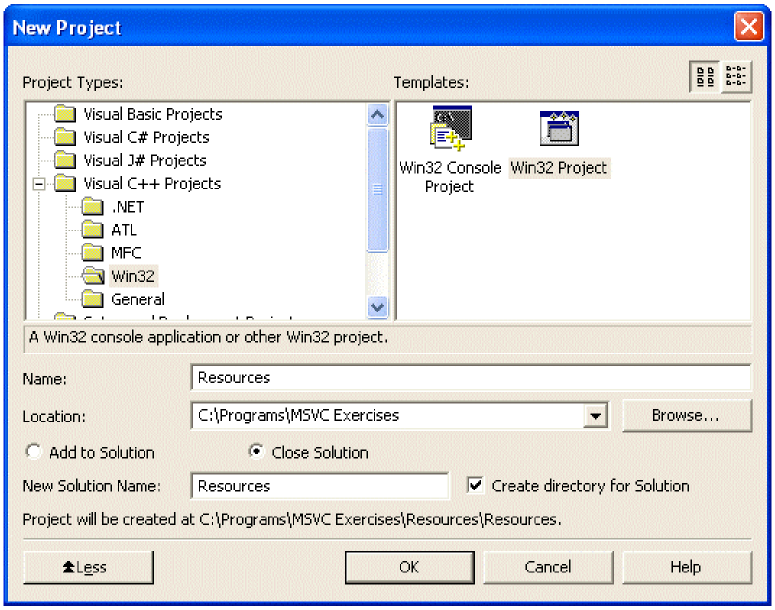
```
LPTSTR MAKEINTRESOURCE(WORD IDentifier);
```

This macro takes the identifier of the resource and returns a string that is given to the function that called it.

In the strict sense, after creating the resource file, it must be compiled to create a new file that has the extension .res. Fortunately, Visual C++ automatically compiles the file and links it to the application.

Practical Learning: Creating an Application

1. Start Microsoft Visual Studio or Microsoft Visual C++
2. Create a new Win32 Project named **Resources**



A Win32 console application or other Win32 project.

Name: Resources

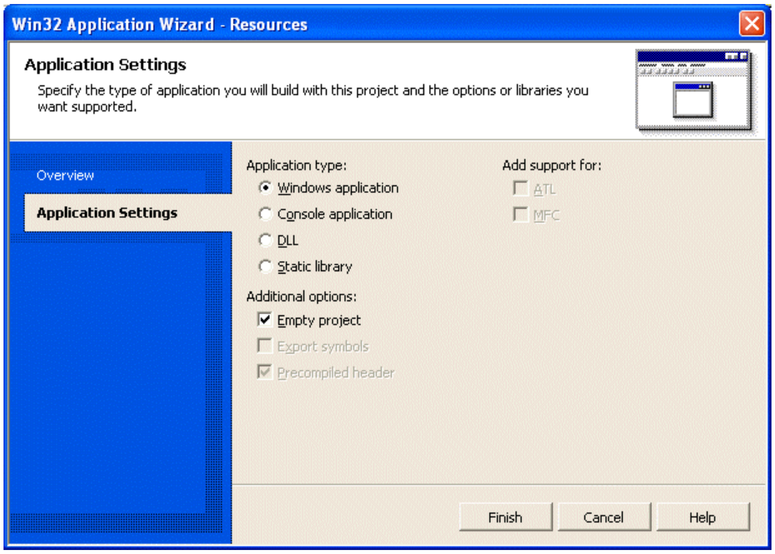
Location: C:\Programs\MSVC Exercises

○ Add to Solution ● Close Solution

New Solution Name: Resources ☑ Create directory for Solution

Project will be created at C:\Programs\MSVC Exercises\Resources\Resources.

3. Click OK and specify the Windows Application as an Empty Project



Win32 Application Wizard - Resources

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:

● Windows application
○ Console application
○ DLL
○ Static library

Add support for:

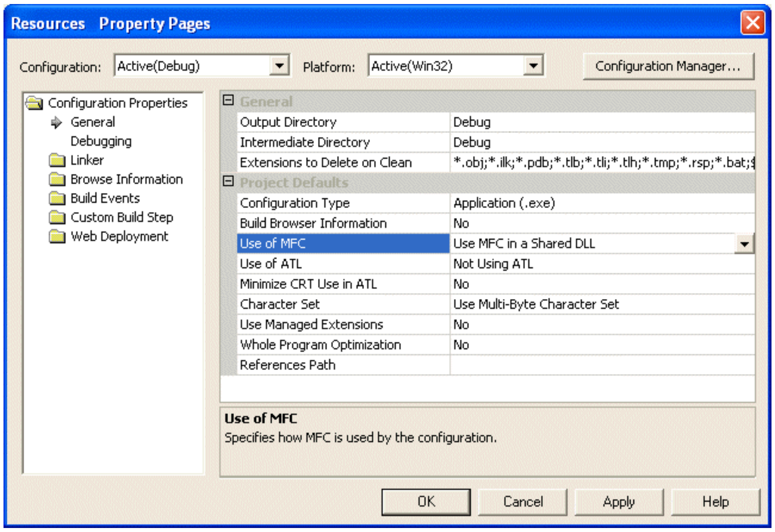
□ ATL
□ MFC

Additional options:

☑ Empty project
□ Export symbols
☑ Precompiled header

4. Click Finish

5. In the Solution Explorer, Resource View, or Class View, right-click the Resources node and click Properties



Resources Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

General	
Output Directory	Debug
Intermediate Directory	Debug
Extensions to Delete on Clean	*.obj;*.ilk;*.pdb;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.bat;$
Project Defaults	
Configuration Type	Application (.exe)
Build Browser Information	No
Use of MFC	Use MFC in a Shared DLL
Use of ATL	Not Using ATL
Minimize CRT Use in ATL	No
Character Set	Use Multi-Byte Character Set
Use Managed Extensions	No
Whole Program Optimization	No
References Path	

Use of MFC
Specifies how MFC is used by the configuration.

6. Specify that you want to **Use MFC as a Shared DLL** for your application and click OK

7. Add a new item as a C++ file and name it **Exercise**

8. Change its content as follows:

```
#include <afxwin.h>

class CResApp : public CWinApp
{
public:
	BOOL InitInstance();
};

BOOL CResApp::InitInstance()
{
	return TRUE;
}
```

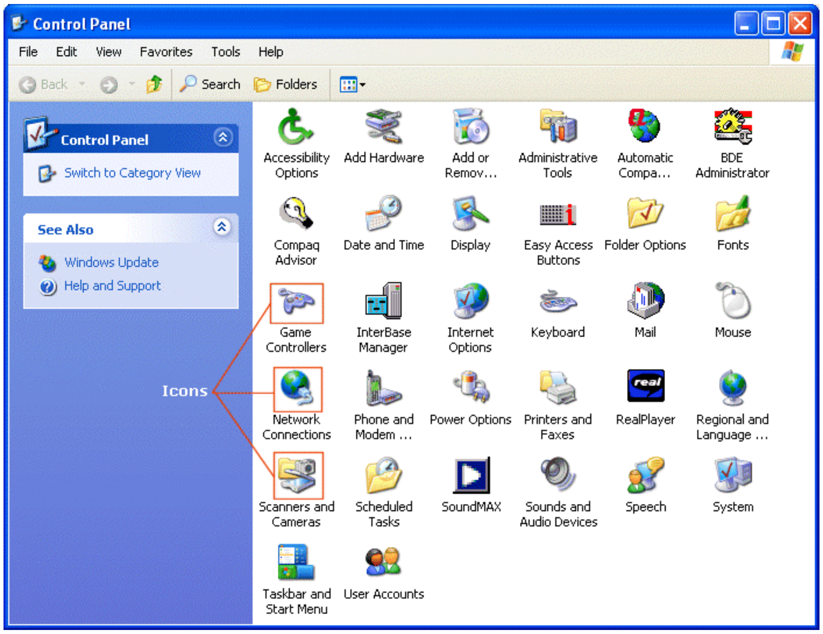
```
CResApp theApp;
```

9. Save All

3.2 Icons

3.2.1 Icons Overview

An icon is a small picture used on a window. It is used in two main scenarios. On a window's frame, it display on the left side of the window name on the title bar. In Windows Explorer, on the Desktop, in My Computer, or in the Control Panel windows, an icon is used to represent an application:

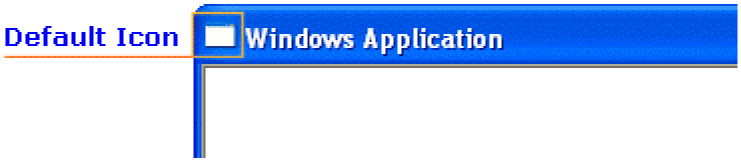


On a window, the icon is a 16x16 pixels size of picture to accommodate the standard height of the title bar. In window displays such as Windows Explorer or My Computer, the applications can be represented as tiles or as a list. Each display uses a different size of icon. Therefore, an icon is created in two sizes that share the same name but the operating system can manage that concept. This means that you will sometimes create two designs for one icon, a 16x16 pixel and a 32x32 pixel.

An icon is a graphical object made of two categories of colors. One category represents the artistic side of the design. The other is the color that would be used as background so that if the icon is positioned on top of another picture, the background color would be used as the transparent color to show the sections that are not strictly part of the icon. In reality, Microsoft Windows imposes the color that is used as background on an icon. You will find what the icon is.

3.2.2 Icons Design

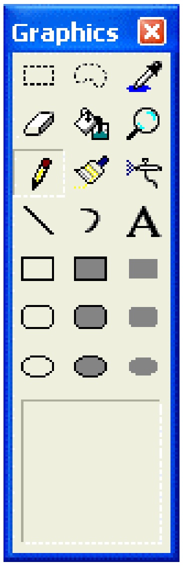
So far, we were not specifying an icon for our window. Therefore, Visual C++ was using a default icon for the title bar:



This icon is 16x16 pixels. If you want to display a custom icon on the title bar, you can design your own 16x16 pixel icon as a resource.

If you want your application to display a big icon in the thumbnails or tiles views of Windows Explorer or My Computer, design an icon with a 32x32 pixel dimension and using the same name as the 16x16 pixel icon.

To create an icon or add a new one, you can right-click the project name in the Solution Explorer, position the mouse on Add and click Add Resource. This displays the Add Resource dialog box where you can double-click Icon.

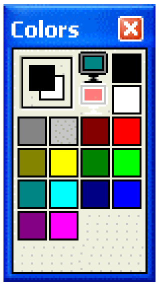


To design an icon, you use a set of tools that allow you to draw lines, curves, points, etc. These tools are available from the Graphics toolbar. Each tool is called to by the text of its tool tip. To know the name of a tool, position the mouse on it for a few seconds.

Tool	Name	Description
⬚	Rectangle Selection	Used as an outline to select a rectangular area
◌	Irregular Selection	Used to draw freehand selection but a rectangle is drawn from the starting selection to the end
🖉	Select Color	Used to select a color
⌫	Erase	Used to erase color on an area
🪣	Fill	Fills a section with a color, only the same pixels with the same color are filled
🔍	Magnify	Zooms an area of the picture
✏	Pencil	Draws lines
🖌	Brush	Used to draw predefined lines, small squares or large lines
⁂	Airbrush	Assigns a color to random pixels on the picture
╲	Line	Used to draw a line
)	Curve	Draws a curve in a multiple step process
A	Text	Allows drawing text
□	Rectangle	Used to draw a rectangular shape specifying only the border color
▣	Outlined Rectangle	Draws a rectangle with one color for the border (pen) and another color for the inside (brush)
■	Filled Rectangle	Used to draw a rectangle filling it with an interior color while the border color is ignored
▢	Round Rectangle	Used to draw a round rectangular shape specifying only the border color

▢	Outlined Round Rectangle	Draws a round rectangle with one color for the border (pen) and another color for the inside (brush)
■	Filled Round Rectangle	Used to draw a round rectangle filling it with an interior color while the border color is ignored
	Ellipse	Used to draw an ellipse or a circle
⬭	Outlined Ellipse	Draws an ellipse or a circle with one color for the border (pen) and another color for the inside (brush)
⬬	Filled Ellipse	Used to draw an ellipse or a circle filling it with an interior color while the border color is ignored

To use a particular tool, click it to select. Then click on the icon.



To design your icon, you can choose colors from the Colors toolbar. Because an icon uses a background color, the default is set as green. The default icon is represented by a small monitor icon.

To use a color, first select a tool on the Graphics toolbar, then click the desired color and draw.

After creating an icon, you can use it for your application. There are two solutions you can use. The default icon used by MSVC is called **AFX_IDI_STD_FRAME**. Therefore, after designing your icon, you can simply set its name as this default.

Practical Learning: Creating an Icon

1. To create an icon, on the main menu, click Project -> Insert Resource...
2. In the Add Resource dialog box, click Icon

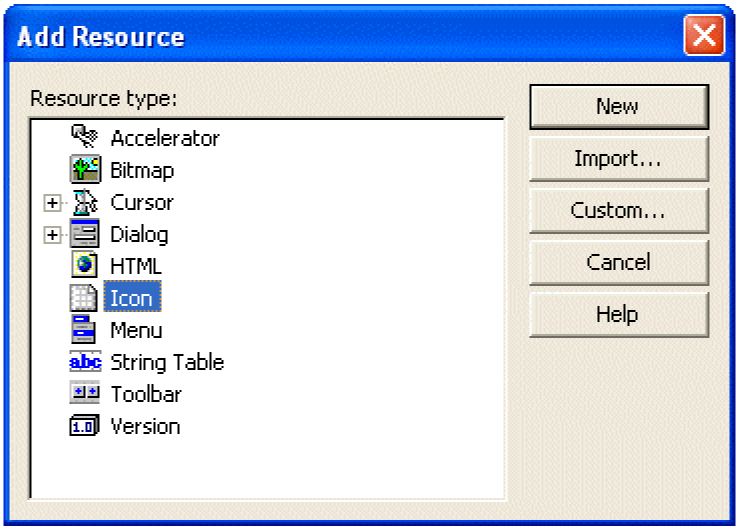
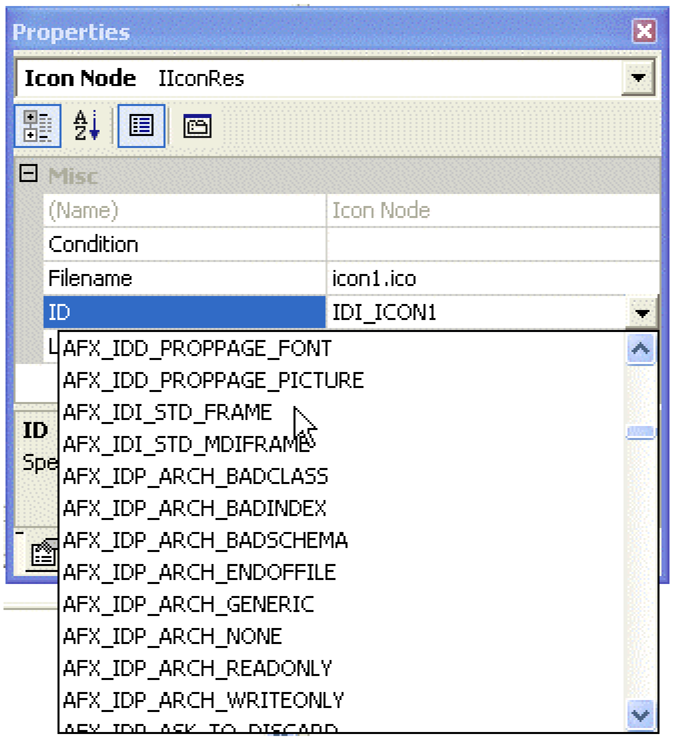


Figure 45: Add Resource - Icon

3. Click New
4. In the Resource View, right-click IDI_ICON1 and click Properties.
5. In the Properties window, click the arrow of the ID combo box and select **AFX_IDI_STD_FRAME**



Properties

Icon Node IIconRes

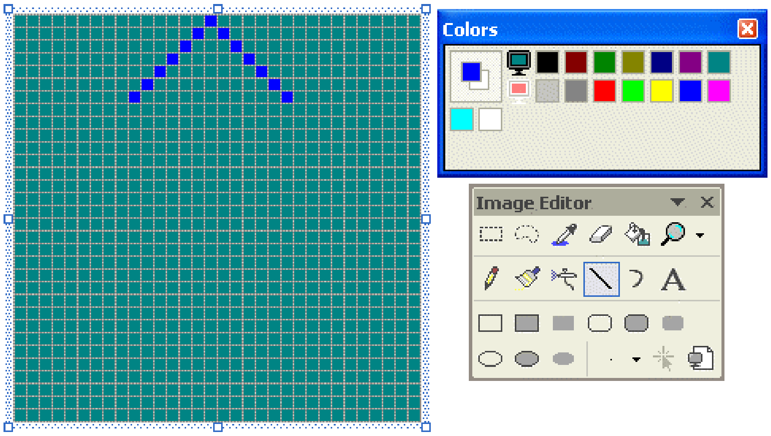
Misc

(Name)	Icon Node
Condition	
Filename	icon1.ico
ID	IDI_ICON1

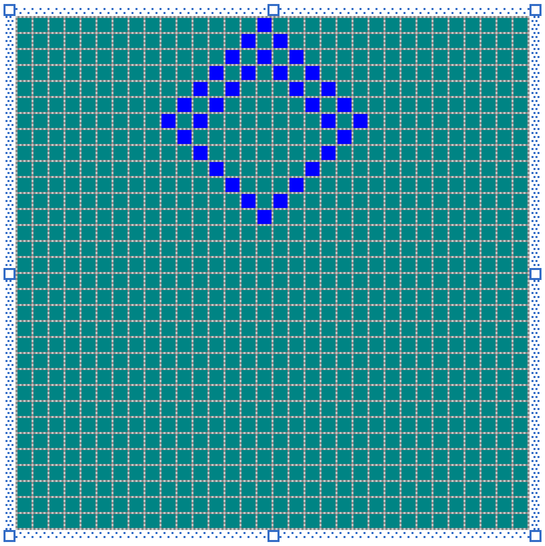
AFX_IDD_PROPPAGE_FONT
AFX_IDD_PROPPAGE_PICTURE
AFX_IDI_STD_FRAME
AFX_IDI_STD_MDIFRAME
AFX_IDP_ARCH_BADCLASS
AFX_IDP_ARCH_BADINDEX
AFX_IDP_ARCH_BADSCHEMA
AFX_IDP_ARCH_ENDOFFILE
AFX_IDP_ARCH_GENERIC
AFX_IDP_ARCH_NONE
AFX_IDP_ARCH_READONLY
AFX_IDP_ARCH_WRITEONLY
AFX_IDP_ASK_TO_DISCARD

ID
Spe

6. On the Properties window, click Filename. Type **diamond** and press Enter
7. On the Image Editor toolbar, click the Line Tool button
8. In the Colors window, click the blue color (7th column, 2nd row)
9. In the empty drawing area, count 15 pixels from the top left to the right. On the 16th box, click and drag right and down for an angle of 45° for 7 boxes. Release the mouse
10. Click the same top blue box and drag left and down at 45° for 7 boxes:

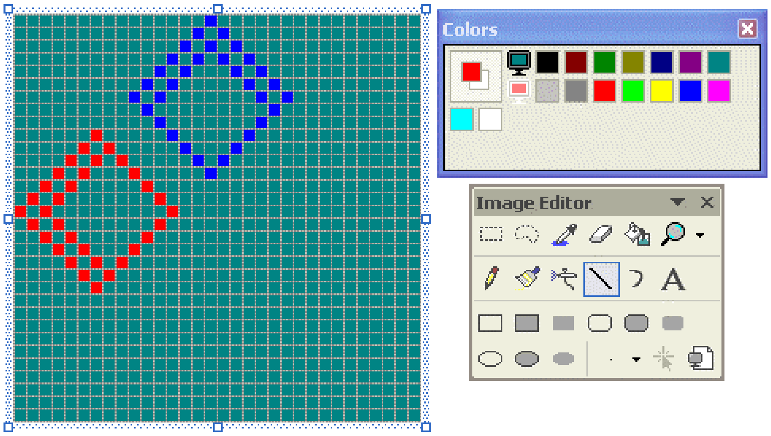


11. Draw a diamond under the first one as follows:

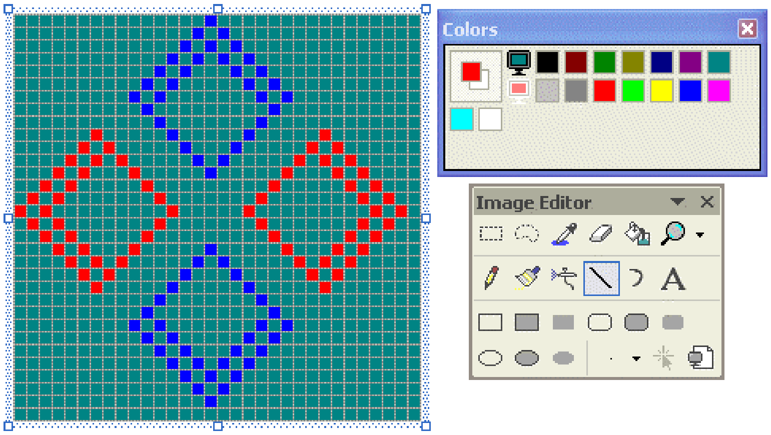


12. On the Colors window, click the red color (3rd column, 2nd row).

13. Draw a reverse graphic with regard to the above dialog as follows:

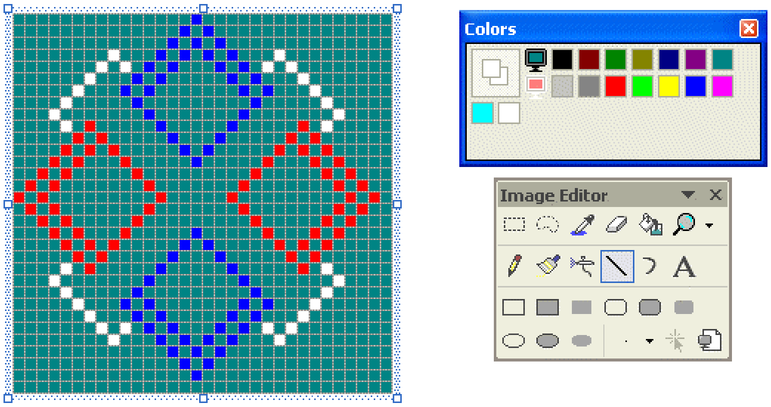


14. Using the blue and the red colors, design the other diamonds as follows:

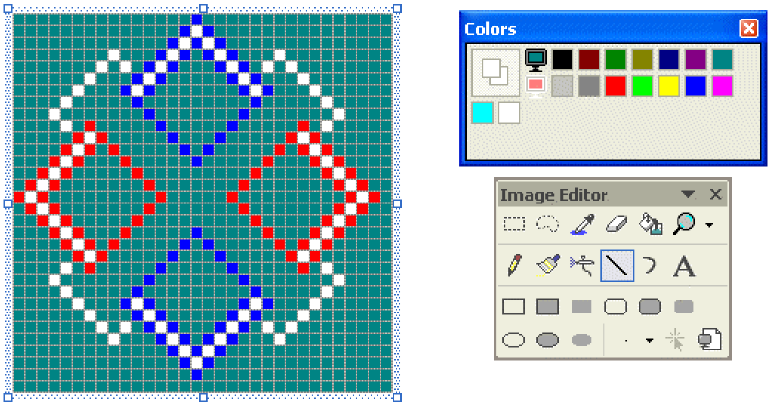


15. On the Colors window, click the white button

16. Using the Line Tool, draw four borders as follows:

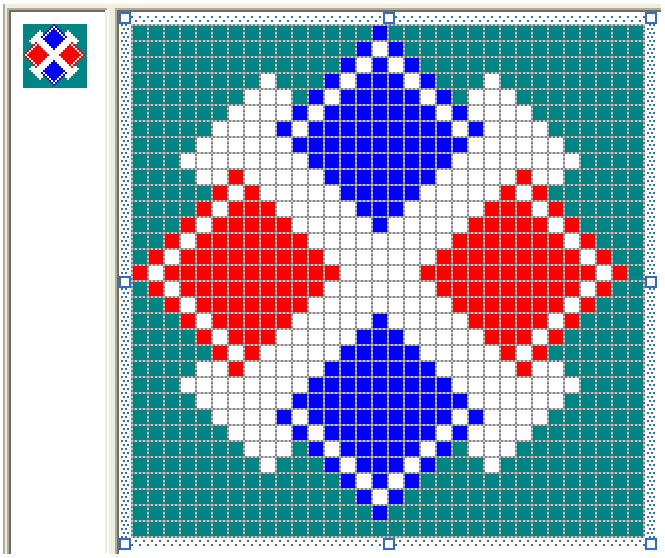


17. Still using the Line Tool and the white color, draw new white lines as follows:



18. On the Image Editor toolbar, click the Fill Tool button .

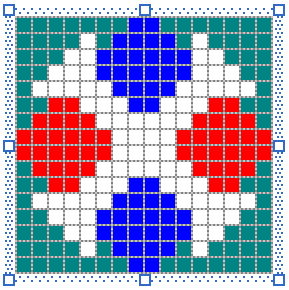
19. Using the white, red, and blue colors, fill the icon as follows:



20. To create the smaller equivalent icon, on the main menu, click Image -> New Image Type...

21. Make sure that 16x16, 16 Colors is selected and click OK.

22. Using the same above approach, design the icon as follows:

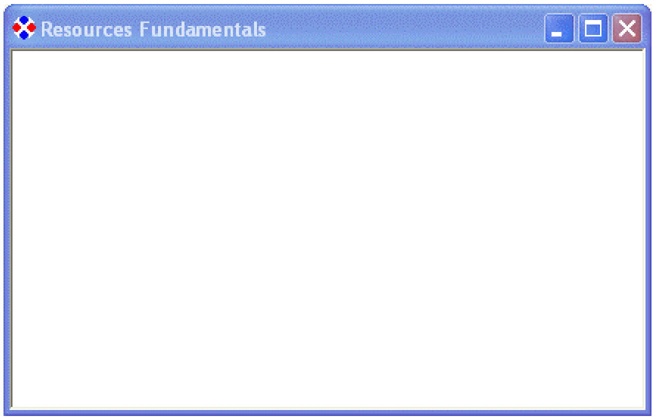


23. To save the icon, click the system Close button ☒ of the window that is displaying the icon

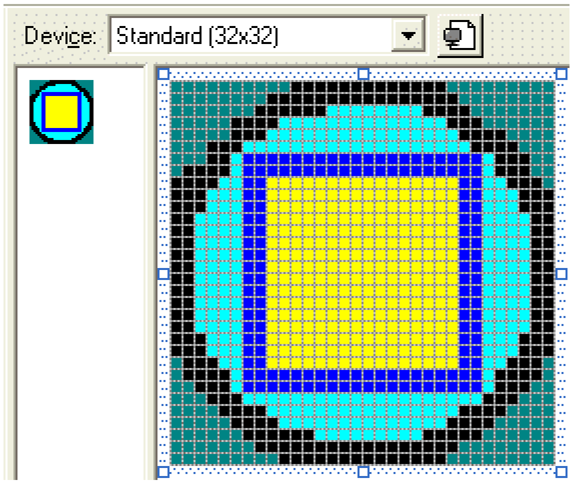
24. Change the Exercise.cpp file as follows:

```
#include <afxwin.h>
#include "ResourceDlg.h"

class CResApp: public CWinApp
{
public:
      BOOL InitInstance();
};

class CResFrame : public CFrameWnd
{
public:
      CResFrame()
      {
                  Create(NULL, "Resources Fundamentals");
      }
};

BOOL CResApp::InitInstance()
{
      m_pMainWnd = new CResFrame;
      m_pMainWnd->ShowWindow(SW_SHOW);
      m_pMainWnd->UpdateWindow();

      return TRUE;
}

CResApp theApp;
```

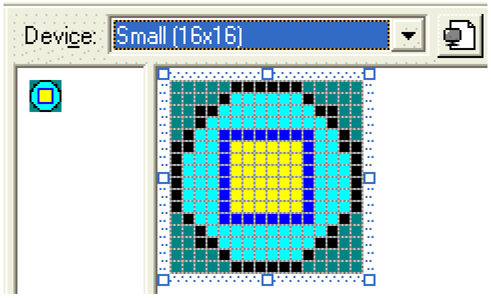
25. Test the application:



26. Close the window and return to MSVC
27. Create another icon identified as IDI_APP_ICO and design it follows:



Device: Standard (32x32)



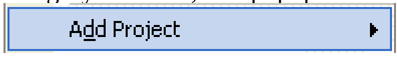
Device: Small (16x16)

28. Save All

3.3 Menu Fundamentals

3.3.1 Overview

A menu is a list of actions the user can perform on an application. Each item of the list is primarily a word or a group of words on a line. Different menu items are used for different reasons. For example, some menu items simply display a word or a group of words. Some other items display a check mark. This indicates that the item toggles the availability or disappearance of an object.

When a menu item is only meant to lead to a sub-menu, such a menu item is call a popup menu. There are two types of popup menus. If the menu displays on top of a window, which is the type of menu under the title bar, the word on top, which represents a category of menu, is a popup menu. If a menu item is equipped with an arrow in its right 

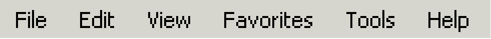
, which means the menu item has a submenu, such a menu item is also a popup menu. Popup menus are used only to represent a submenu. No inherent action is produced by clicking them, except that, when placed on top, such menu items allow opening the submenu.

To create menus that belong to a group, menu items are separated by a horizontal line called a separator. Separators are created differently in MSVC 6 and MSVC 7.

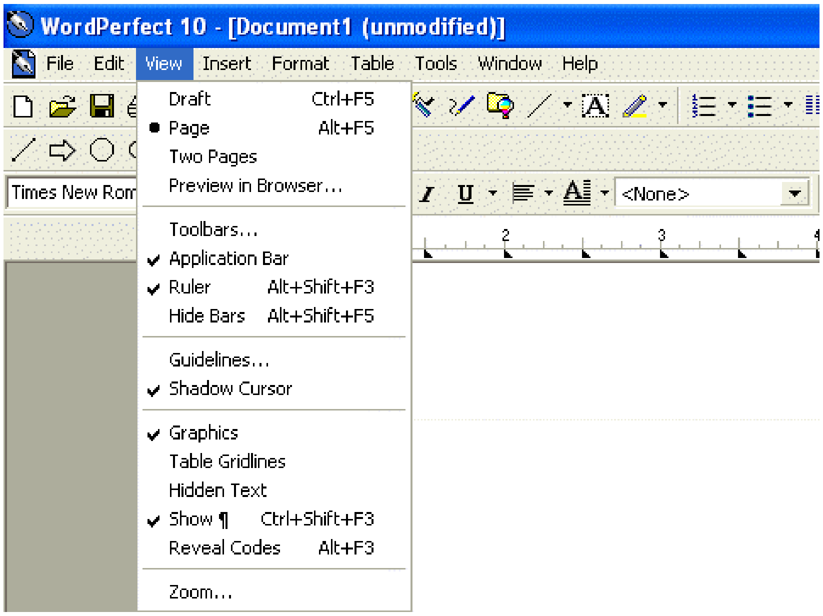
There are two primary types of menus in most applications: a main menu and a popup menu.

3.3.2 The Main Menu

A menu is considered a main menu, when it carries most or all of the actions the user can perform on an application. Such a menu is positioned on the top section of the main window in which it is used. A main menu is divided in categories of items and each category is represented by a word. Here is an example:



On the Visual Studio IDE, the categories of menus are File , Edit, View, Project, etc. To use a menu, the user first clicks one of the words that displays on top. Upon clicking, the menu expands and displays a list of items that belong to that category. Here is an example where the View menu of WordPerfect was clicked and got expanded:



There is no strict rule on how a menu is organized, only suggestions. For example, actions that are related to file processing, such as creating a new file, opening an existing file, saving a file, printing the open file, or closing the file usually stay under a category called File. In the same way, actions related to viewing things can be listed under a View menu.

3.3.3 Main Menu Design

There are two ways you can create a main menu. You can use the Win32 approach in which case you would create or open your .rc file and create a section as follows:

```
IDR_MAINFRAME MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New",        IDM_FILENEW
        MENUITEM "&Open",       IDM_FILEOPEN
        MENUITEM SEPARATOR
        MENUITEM "E&xit",       IDM_FILEEXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",      IDM_HELPABOUT
    END
END
```

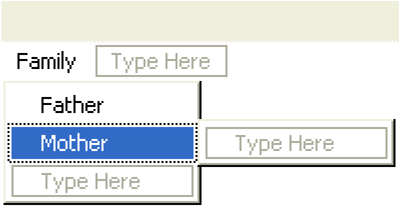
If you create your file manually, you must also remember to create or edit the resource.h file in order to specify an identifier for each menu. The alternative, which we will use, is to "visually" create the menu in Visual Studio. When doing this, the studio itself would update the resource.h as items are added or removed.

To create a menu, first add a resource of type Menu. To create a popup menu that would display on top of the main menu, click the item on top and type the desired string in the Caption field of the Properties window. Such a popup menu item would not use a specify identifier.

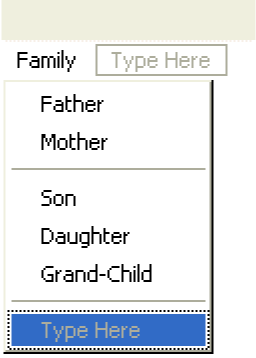
To create a menu item, click the line under the popup menu, provide an identifier and a caption. The arrow for the popup menu is readily available so you can use or ignore it.

Practical Learning: Creating a Main Menu

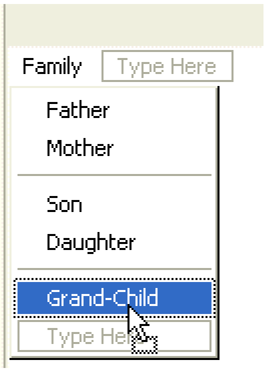
1. On the main menu, click Project -> Add Resource...
2. On the Add Resource dialog box, double-click Menu
3. In the Resource View, click IDR_MENU1 to select it and change its identifier to **IDR_MENU_RES**
4. In the main window, click the top box (in MSVC .Net, it displays Type Here), type **Family** and press Enter
5. Click the item under Family. Type **Father** and press Enter
6. Type **Mother** and press Enter



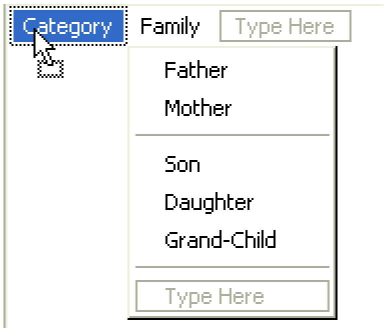
7. To add a separator, click the item under mother, type - and press Enter
8. Complete the menu as follows (remember to add the lower separator):

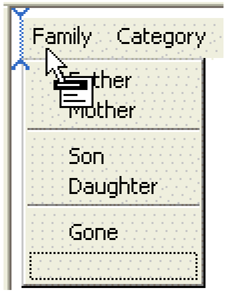


9. To move the Grand-Child item and position it under the lower separator, click and hold the mouse on Grand-Child, then drag in the bottom direction until the selection is in the desired position:



10. Release the mouse.
11. To create another main menu item, click the box on the right side of Family, type **Category** and press Enter
12. Click the item under Category, type **Parent** and press Enter.
13. Type **Child**and press Enter
14. To move the Category menu and position it to the left side of Family, click and drag Category in the left direction





15. When it is positioned to the left of Family, release the mouse. Notice that the popup menu and its submenus moved.
16. To create a new main menu item, click the box on the right side of Family, type **Job Functions** and press Enter
17. Click the box under Job Functions and type **Level**
18. While Level is still selected, click the box on the right side of Level
19. Type **Executive** and press Enter
20. Complete the popup menu as follows:

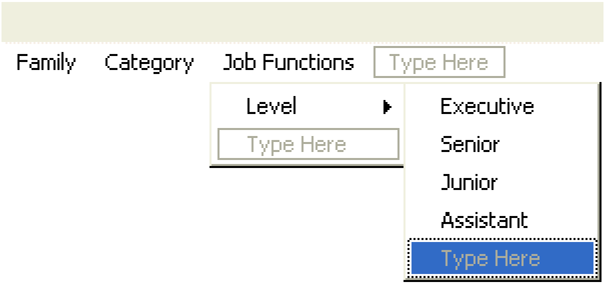
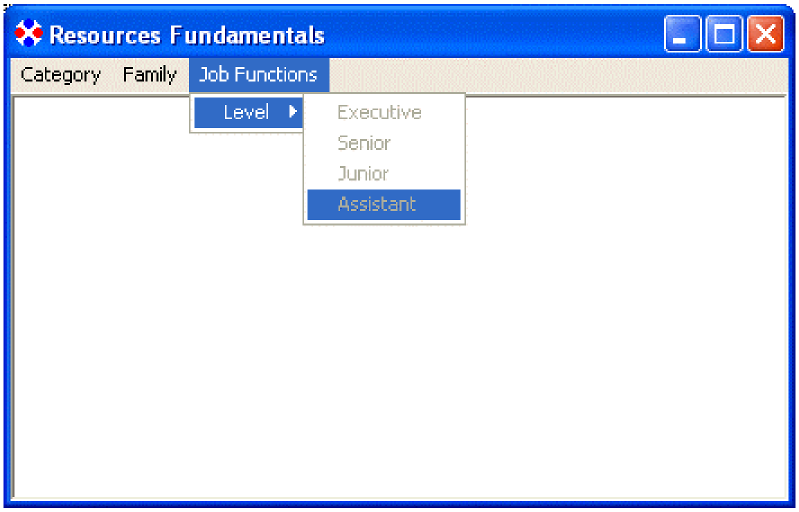


Figure 46: Simple Menu

21. To use the new menu, open the Exercise.cpp file and change the CFrameWnd::Create() method as follows:

```
#include <afxwin.h>
#include "resource.h"

class CResApp: public CWinApp
{
public:
	BOOL InitInstance();
};

class CResFrame : public CFrameWnd
{
public:
	CResFrame()
	{
		Create(NULL, "Resources Fundamentals",
			WS_OVERLAPPEDWINDOW, CRect(200, 120, 640, 400),
			NULL,
			MAKEINTRESOURCE(IDR_MENU_RES));
	}
};

BOOL CResApp::InitInstance()
{
	m_pMainWnd = new CResFrame;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}

CResApp theApp;
```

22. Test the application



23. Close the window and return to MSVC

3.4 Toolbars

3.4.1 Overview

A toolbar is a Windows control that allows the user to perform some actions on a form by clicking a button instead of using a menu. A toolbar provides a convenient group of buttons that simplifies the user's job by bringing the most accessible actions as buttons so that, instead of performing various steps to access a menu, a button on a toolbar can bring such common actions closer to the user.

Toolbars usually display under the main menu. They can be equipped with buttons but sometimes their buttons or some of their buttons have a caption. Toolbars can also be equipped with other types of controls

3.4.2 Creating a Toolbar

To create a toolbar, from the Add Resource dialog box, click Toolbar and click New.

A toolbar is only a container and does not provide much role by itself. To make a toolbar efficient, you should equip it with the necessary controls.

The most common control used on a toolbar is a button. After adding a new toolbar, it is equipped with a gray button. You can simply start designing that button as you see fit. Once you start designing a button, a new one is added. You can keep designing the buttons until you get as many buttons as you need. If you design a button but do not need it anymore, to delete it, drag it away from the toolbar. The space between two buttons is called a separator. To put a separator between two buttons, drag one away from the other just a little.

3.5 Accelerators

3.5.1 Access Keys

An access key is a letter that allows the user to perform a menu action faster by using the keyboard instead of the mouse. This is usually faster because the user would not need to position the mouse anywhere, which reduces the time it takes to perform the action.

The most basic accelerator keys allow the user to access each item of the menu with a key. To do this, the user must first give focus to the menu. This is done by pressing the F10 function key or Alt. Once the menu has focus, you can provide a unique letter that the user can press to activate the menu. Each main popup menu must have a unique letter that serves as access key. The letter is underlined to show that it the access key for that particular menu.

As a suggestion, when creating the access keys, use the first letter of the menu item, as in File, Edit, or View. If you have another menu item that starts with a letter already used, as Format after File, use the next letter that has not been used already. This can result in File, Edit, View, Format, Insert, Efficiency. Only items in the same category should follow this rule. The menu items under a popup menu use access keys that are independent of another category. This means that, under File, you can use a submenu that uses the letter E as access key even though Edit on top is using it.

To use access keys, the user press F10 or Alt and presses the underlined letter, which opens the menu category. Then the user can click the desired underlined letter in the displayed list.

To create an access key, type an ampersand "&" on the left of the menu item.

Practical Learning: Creating Access Keys

1. In the Resource View, double-click the identifier of the menu to display the menu.
2. Right-click Category and click Properties
3. In the Caption box, click to left of Category, type **&** and press Enter
4. Complete the menu with the following captions:

&Category	&Family	&Job Functions
&Parent	&Father	&Level
&Child	&Mother	&Executive
	&Son	&Senior
	&Daughter	&Junior
	&Grand-child	&Assistant

5. Test the application and return to MSVC

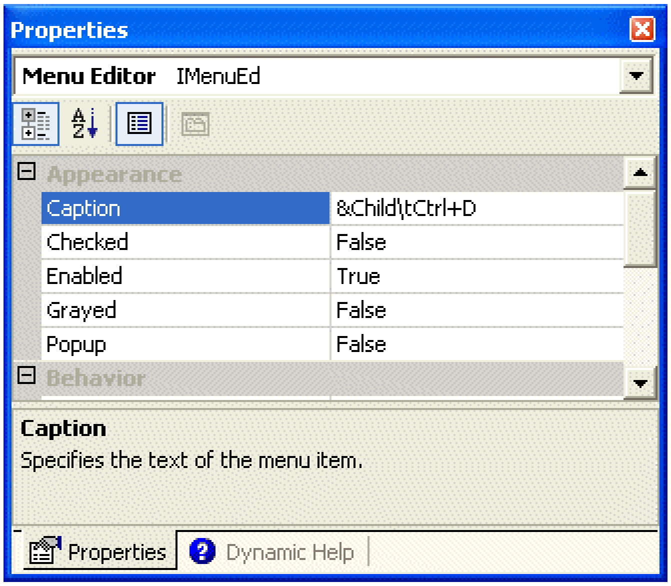
3.5.2 Shortcuts

A shortcut key is a key or a combination of keys the user presses to perform an action that would otherwise be done on a menu item. Most shortcuts are made of the Ctrl key simultaneously pressed with a letter key. Examples are Ctrl + N, Ctrl + O, or Ctrl + D. Some applications, such as Adobe Photoshop or Macromedia Flash, use a shortcut made of only one key.

To create a shortcut, on the right side of the string that makes up a menu caption, type \t followed by the desired combination.

Practical Learning: Creating Shortcut Keys

1. In the Resource View, click Category, right-click Parent, and click Properties
2. In the Caption box, click to right side of Parent, type \tCtrl+R
3. In the same way, set the shortcut of Child to Ctrl+D



Caption	&Child\tCtrl+D
Checked	False
Enabled	True
Grayed	False
Popup	False

Caption
Specifies the text of the menu item.

4. Test the application and return to MSVC

3.5.3 Accelerator Table

An Accelerator Table is a list of items where each item of the table combines an identifier, a (shortcut) key, and a constant number that specifies the kind of accelerator key. Each item is a variable of the **ACCEL** class whose syntax is:

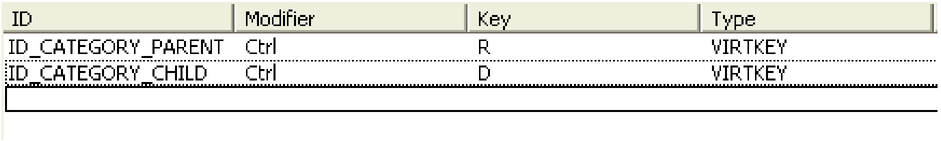
```
typedef struct tagACCEL {
    BYTE   fVirt;
    WORD   key;
    WORD   cmd;
} ACCEL, *LPACCEL;
```

Using a variable or variables of type **ACCEL**, you can create as many items as needed. Then declare an **HACCEL** variable. **HACCEL** is a handle to a **ACCEL** and is used to pass the **ACCEL** table to the program. This would allow the compiler to translate the items of the accelerator to keys that the program can use.

Just like the other resources, an accelerator table can be created manually in a .rc file but MSVC simplifies the process by allowing you to "visually" create the table.

Practical Learning: Creating an Accelerator Table

1. On the main menu of Visual Studio, click Insert -> Resource... or Project -> Add Resource...
2. In the Add Resource dialog box, double-click Accelerator
3. In the Properties window, click the arrow of the ID combo box and select ID_CATEGORY_PARENT
4. Click the Key box and type R
5. Make sure the Ctrl check box is checked or that the Ctrl field is set to True. Also, make sure the Type is set to **VIRTKEY**
6. In the same way, create an accelerator item for the Child menu:



ID	Modifier	Key	Type
ID_CATEGORY_PARENT	Ctrl	R	VIRTKEY
ID_CATEGORY_CHILD	Ctrl	D	VIRTKEY

7. To use the accelerator, change the program as follows:

```
class CResFrame : public CFrameWnd
{
public:
    HACCEL m_hAccel;

    CResFrame()
    {
            m_hAccel = ::LoadAccelerators(AfxGetInstanceHandle(),
            MAKEINTRESOURCE (IDR_ACCELTEST));

            Create(NULL, "Resources Fundamentals",
                  WS_OVERLAPPEDWINDOW,
                 CRect(200, 120, 640, 400), NULL,
                  MAKEINTRESOURCE(IDR_TEST));
    }
};
```

8. Test the application and return to MSVC

3.6 Version Information

3.6.1 Overview

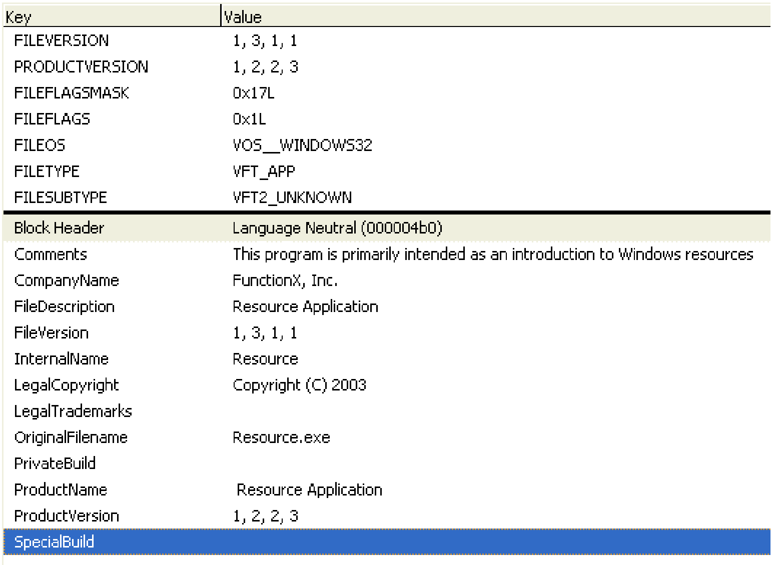
The version of a computer program allows to have some information about the product such as it official name, a release number, the person or company that owns its copyright, the year of the current release, etc.

3.6.2 The Version Information Editor

To create the pieces of information that make up the version information, you can use the Version Information Editor.

Practical Learning: Creating Version Information

1. On the main menu of MSVC, click Project -> Add Resource
2. In the Add Resource dialog box, click **Version** and click New. Notice that a complete file is generated for you
3. To change the first key, double-click the top 1, 0, 0, 1 and edit it to display 1, 3, 1, 1
4. Double-click **PRODUCTVERSION** and type 1, 2, 2, 3
5. Double-click **Comments** and type **This program is primarily intended as an introduction to Windows resources**
6. To add a new string, right-click in the window and click New Version Info Block



Key	Value
FILEVERSION	1, 3, 1, 1
PRODUCTVERSION	1, 2, 2, 3
FILEFLAGSMASK	0x17L
FILEFLAGS	0x1L
FILEOS	VOS__WINDOWS32
FILETYPE	VFT_APP
FILESUBTYPE	VFT2_UNKNOWN
Block Header	Language Neutral (000004b0)
Comments	This program is primarily intended as an introduction to Windows resources
CompanyName	FunctionX, Inc.
FileDescription	Resource Application
FileVersion	1, 3, 1, 1
InternalName	Resource
LegalCopyright	Copyright (C) 2003
LegalTrademarks	
OriginalFilename	Resource.exe
PrivateBuild	
ProductName	Resource Application
ProductVersion	1, 2, 2, 3
SpecialBuild	

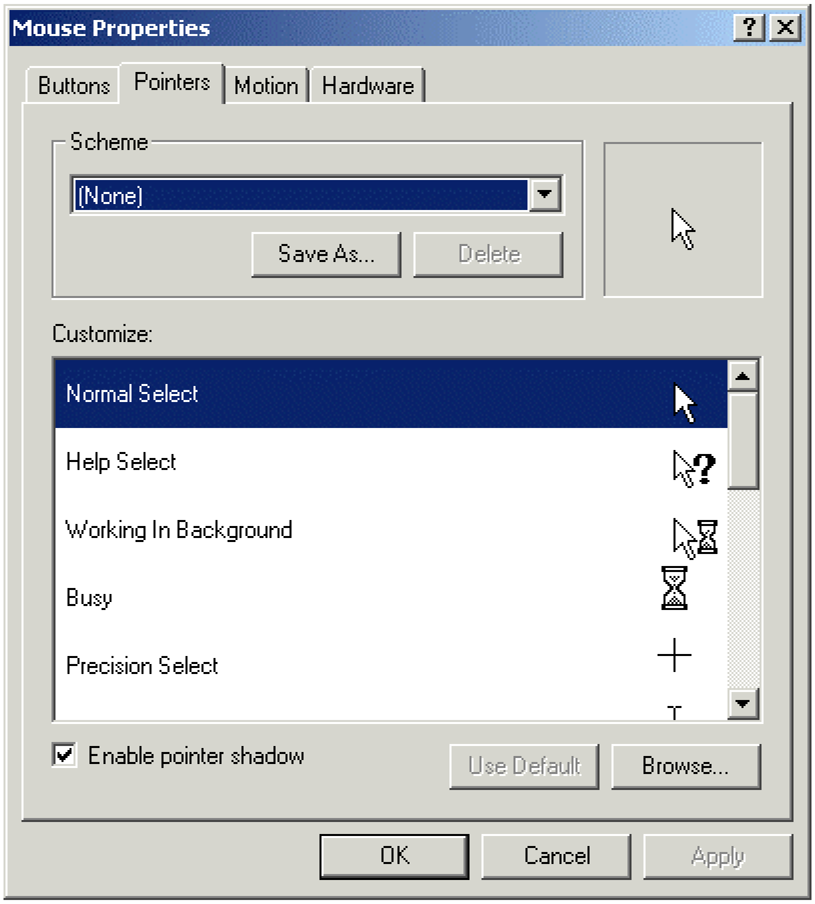
Figure 47: Version Table

7. Save All

3.7 Cursors

3.7.1 Overview

A cursor is a small picture that represents the position of the mouse on a Windows object. Because Windows is a graphic operating system, when it installs, it creates a set of standard or regularly used cursors. These can be seen by opening the Control Panel window and double-clicking the Mouse icon. This opens the Mouse Properties dialog box where you can click the Pointers tab to see a list of standard cursors installed by Windows:



3.7.2 Creating and Using Cursors

Microsoft Windows installs a wide array of cursors for various occasions. Like all other resources, a cursor is identified by a constant integer that can be communicated to other files that need it.

Essentially, a cursor uses only two colors, black and white. This is because a cursor is only used as an indicator of the presence or position of the mouse pointer on the screen. Based on this (limitation), you ought to be creative. The minimumyou can give a cursor is a shape. This can be a square, a rectangle, a circle, an ellipse, a triangle, or any shape of your choice. You can make the cursor fully black by painting it with that color. If you decide to make the cursor completely white, make sure you draw borders.

Between the black and white colors, two gray degrees are provided to you. In reality these two colors are used to give transparency to the cursor so the background can be seen when the mouse passes over a section of the document.

After designing a cursor, you should define its hot spot. A cursor's hot spot is the point that would be used to touch the specific point that the mouse must touch to perform the action expected on the mouse. The hot spot must be an area, namely a spot, on the cursor but it could be anywhere on that cursor. You should specify the hot spot in the most intuitive section of the cursor. In other words, the user should easily identify it since it is not otherwise visible.

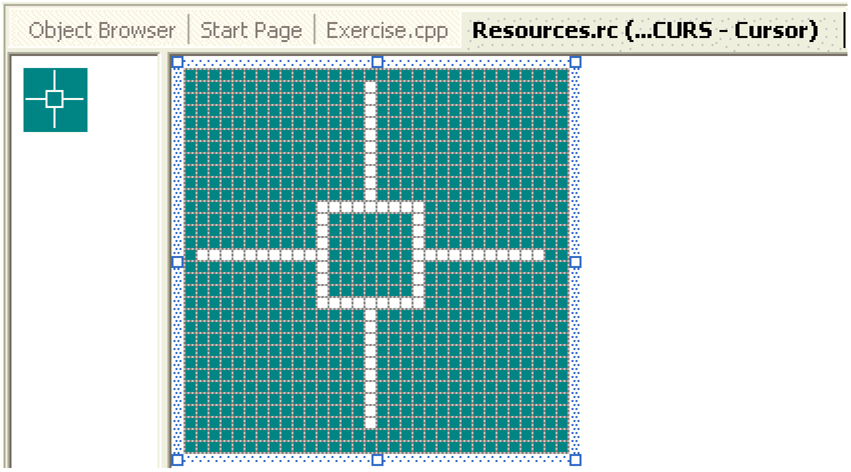
If you do not want to show a cursor, you can use the Wind32 API **ShowCursor()** function. Its syntax is:

```
int ShowCursor(BOOL bShow);
```

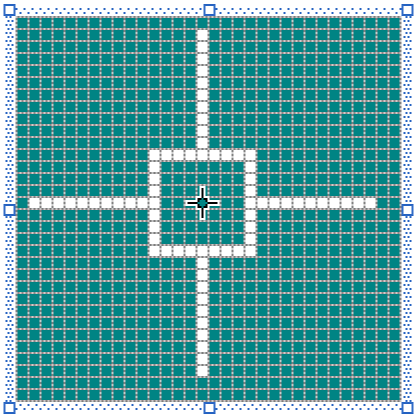
To hide a cursor, specify the argument as **FALSE**.

Practical Learning: Creating a Cursor

1. To create a new cursor, on the main menu, click Project -> Add Resource
2. In the Insert Resource or Add Resource dialog box, click Cursor and click New
3. In the Resource View, right-click the name of the cursor and click Properties
4. Change its ID to **IDC_APP_CURS** and its Filename to appcurs.cur
5. On the Toolbox, click the Line tool and select the black color
6. Design the cursor as follows:



7. To define the hot spot, on the toolbar of the editor, click the Set Hot Spot button . Click the middle of the square:



8. Save All
9. Close the Cursor Properties window.

3.8 The String Table

3.8.1 Description

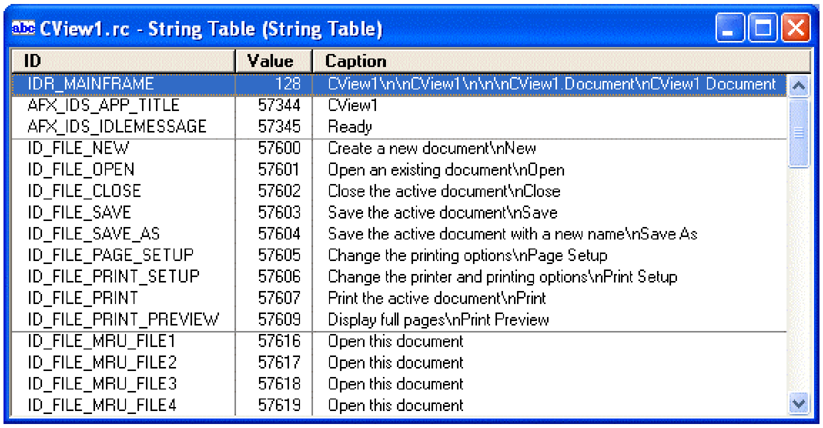
A string table is a list of all object strings that are part of an application. It allows any part of the program to refer to that table when a common string is needed. The advantage of a string table is that it can be used as a central place when to store or retrieve strings that any other objects of the application may need. These can include the titles or captions of Windows controls, the formatting strings used inside of functions or controls.

3.8.2 Creating and Using a String Table

When you install Microsoft Visual C++, it also installs a lot of strings that are readily available for your applications. Most of the time, you will also need to create additional or your own strings. To create a string table, from the Add Resource dialog box, click String Table and click New. Each item of the table is made of three sections: an identifier, a constant natural number, and a caption.

To add a new item, you can right-click in the String Table window and click New String. You can also double-click the last empty row in the window. You can also press Insert. Either case, the String Properties window would display.

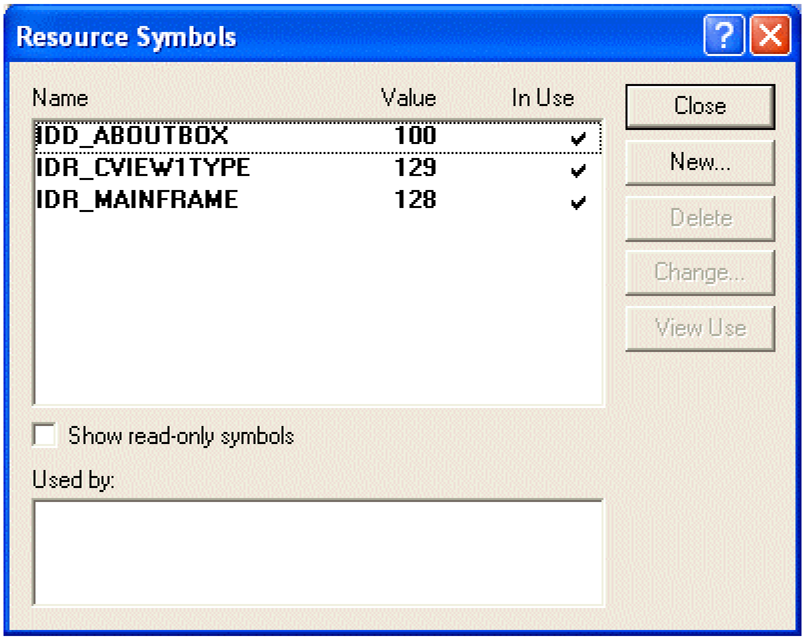
You can either type a new ID or select an existing ID from the ID combo box. Then type the associated string in the Caption box. Continually, you would have created a String Table:



ID	Value	Caption
IDR_MAINFRAME	128	CView1\n\nCView1\n\n\nCView1.Document\nCView1 Document
AFX_IDS_APP_TITLE	57344	CView1
AFX_IDS_IDLEMESSAGE	57345	Ready
ID_FILE_NEW	57600	Create a new document\nNew
ID_FILE_OPEN	57601	Open an existing document\nOpen
ID_FILE_CLOSE	57602	Close the active document\nClose
ID_FILE_SAVE	57603	Save the active document\nSave
ID_FILE_SAVE_AS	57604	Save the active document with a new name\nSave As
ID_FILE_PAGE_SETUP	57605	Change the printing options\nPage Setup
ID_FILE_PRINT_SETUP	57606	Change the printer and printing options\nPrint Setup
ID_FILE_PRINT	57607	Print the active document\nPrint
ID_FILE_PRINT_PREVIEW	57609	Display full pages\nPrint Preview
ID_FILE_MRU_FILE1	57616	Open this document
ID_FILE_MRU_FILE2	57617	Open this document
ID_FILE_MRU_FILE3	57618	Open this document
ID_FILE_MRU_FILE4	57619	Open this document

The identifier, ID, is the same type of ID used when creating the resources earlier. In fact, most of the IDs used on a string table are shared among resources, as we will learn when reviewing the CFrameWnd::LoadFrame() method.

The value column contains a constant LONG integer for each identifier. You will not need to specify this number when using the String Table window; Visual C++ automatically and independently creates and manages those numbers. If you want to create a string and specify your own number, on the main menu of MSVC 6, you can click View -> Resource Symbols...



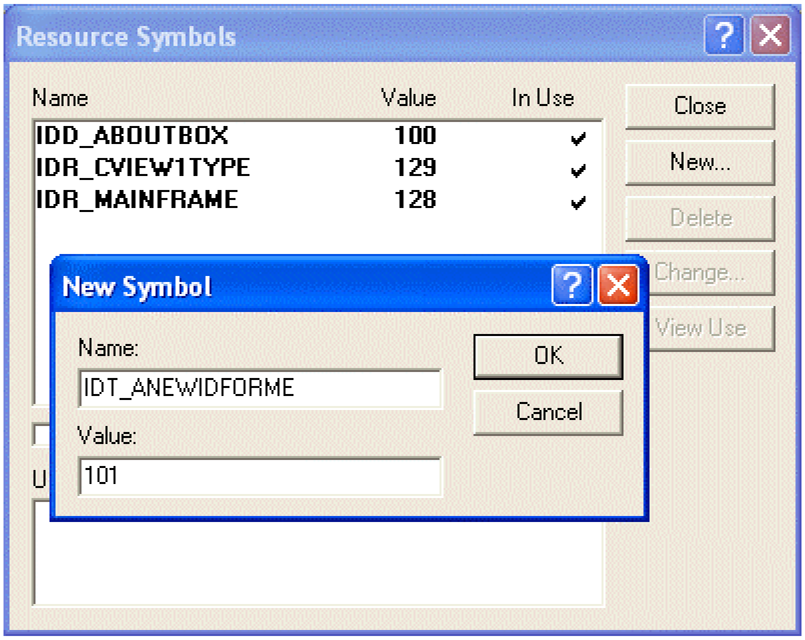
Name	Value	In Use
IDD_ABOUTBOX	100	✓
IDR_CVIEW1TYPE	129	✓
IDR_MAINFRAME	128	✓

□ Show read-only symbols

Used by:

Figure 48: Resource Symbols

In the Resource Symbols window, to create a new string, click the New... button to display the New Symbol dialog box:



Name	Value	In Use
IDD_ABOUTBOX	100	✓
IDR_CVIEW1TYPE	129	✓
IDR_MAINFRAME	128	✓

Name: IDT_ANEWIDFORME

Value: 101

Figure 49: New Symbol

You can then type an IDentifier in the Name edit box and the desired number in the Value edit box. Over all, you should only use numbers between 101 and 127 and avoid numbers over 57344.

To edit an item from the String Table window, double-click the row needed to display the String Properties window with the selected item.

3.9 Other Techniques of Creating Windows

3.9.1 Window Registration and Standard Resources

The **CFrameWnd::Create()** method we have used so far provides a quick mechanism to create a window. Unfortunately, it is not equipped to recognize custom resources. In reality, this method is only used to create a window, not to receive resources. As we saw in its syntax, its first argument, *lpszClassName* is used to get a window that has been built, that is, a window whose resources have been specified. If you do not specify these resources, that is, if you pass the *lpszClassName* argument as **NULL**, the compiler would use internal default values for the window. These default are: a window that can be redrawn when the user moves or resizes it, a white background for the main area of the window, the arrow cursor, and the Windows "flat" icon, called **IDI_APPLICATION**.

To use your own resources, you must first create then register them. To register the resources, you can call the **AfxRegisterWndClass()** global function. Its syntax is:

```
LPCTSTR AFXAPI AfxRegisterWndClass(UINT nClassStyle,
                              HCURSOR hCursor = 0,
                              HBRUSH hbrBackground = 0,
                              HICON hIcon = 0);
```

As you can see, this function is used to register, the window style, a cursor, a color for the background, and an icon. After registering the window's style or the style and the resources, this function returns a (constant) string that can be passed as the first argument of the CFrameWnd::Create() method.

Practical Learning: Using Standard Resources

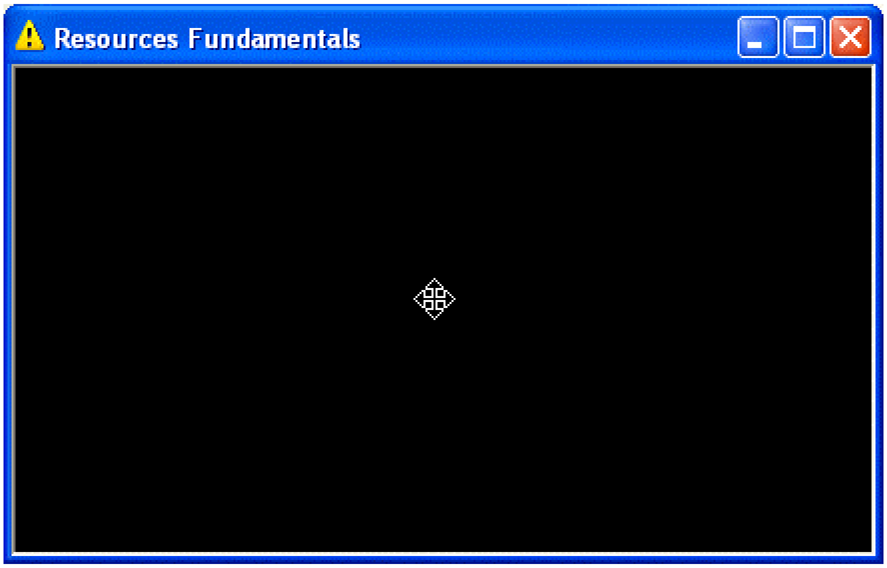
1. To use the AfxRegisterWndClass() function with default values, change the Exercise.cpp file as follows:

```
class CResFrame : public CFrameWnd
{
public:
    CResFrame()
    {
      const char *RWC = AfxRegisterWndClass(NULL, NULL,
                                 (HBRUSH)::GetStockObject(WHITE_BRUSH),
                                 NULL);
      Create(RWC, "Resources Fundamentals", WS_OVERLAPPEDWINDOW,
                           CRect(200, 120, 640, 400), NULL);
    }
};
```

2. Test the application
3. After viewing the window, close it and return to MSVC
4. To use a standard cursor and a standard icon, change the Exercise.cpp file as follows:

```
CResFrame()
{
    HCURSOR hCursor;
    HICON hIcon;

    hCursor = AfxGetApp()->LoadStandardCursor(IDC_SIZEALL);
    hIcon   = AfxGetApp()->LoadStandardIcon(IDI_EXCLAMATION);

    const char *RWC = AfxRegisterWndClass(CS_VREDRAW | CS_HREDRAW,
                                        hCursor,
                                              (HBRUSH)GetStockObject(BLACK_BRUSH),
                                          hIcon);
    Create(RWC, "Resources Fundamentals", WS_OVERLAPPEDWINDOW,
           CRect(200, 120, 640, 400), NULL);
}
```

5. Test the application



6. After viewing the window, close it and return to MSVC

3.9.2 Window Registration and Custom Resources

If the standard cursors and/or icons are not enough, you can create your own. To create your own cursor, display and select Cursor from the Add Resource dialog box. A starting but empty cursor would be displayed. Design the cursor to your liking.

To use a custom cursor, you can retrieve its identifier and pass it to the **CWinApp::LoadCursor()** method. It is overloaded as follows:

```
HCURSOR LoadCursor(LPCTSTR lpszResourceName) const;
HCURSOR LoadCursor(UINT nIDResource) const;
```

To use a custom icon, you can pass its identifier to the **CWinApp::LoadIcon()** method overloaded as follows:

```
HICON LoadIcon(LPCTSTR lpszResourceName) const;
HICON LoadIcon(UINT nIDResource) const;
```

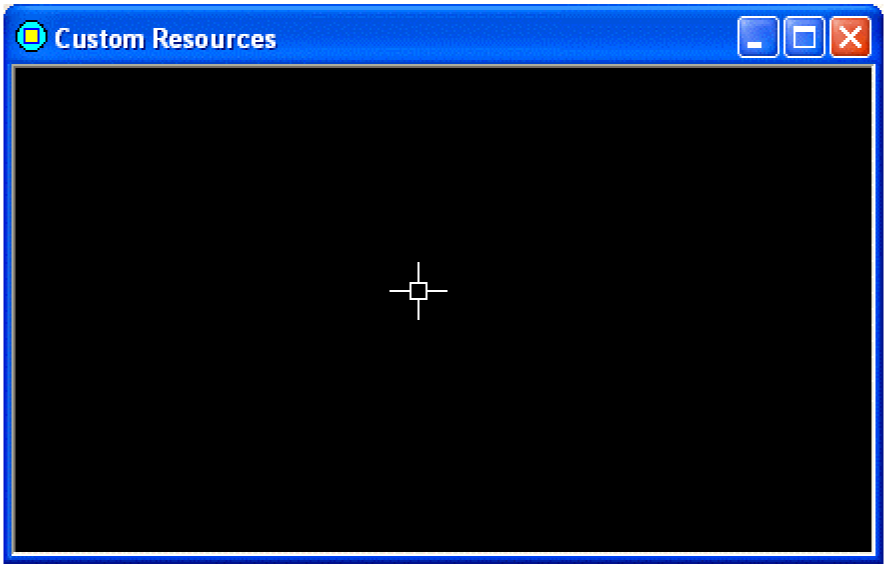
When calling one of these methods, you can simply pass its identifier as argument. You can also specify the resource identifier as a **constant** string. To do this, pass the name of the icon to the **MAKEINTRESOURCE** macro that would convert its identifier to a string.

Practical Learning: Using Custom Resources

1. To use a custom cursor and a custom icon, change the Exercise.cpp file as follows:

```
#include <afxwin.h>
#include "resource.h"

class CResFrame : public CFrameWnd
{
public:
    CResFrame()
    {
      HCURSOR hCursor;
      HICON hIcon;

      hCursor = AfxGetApp()->LoadCursor(IDC_APP_CURS);
      hIcon   = AfxGetApp()->LoadIcon(IDI_APP_ICO);

      const char *RWC = AfxRegisterWndClass(CS_VREDRAW | CS_HREDRAW,
                                            hCursor,
                                            (HBRUSH)GetStockObject(BLACK_BRUSH),
                                            hIcon);
      Create(RWC, "Custom Resources", WS_OVERLAPPEDWINDOW,
                 CRect(200, 120, 640, 400), NULL);
    }
};

class CResApp: public CWinApp
{
public:
    BOOL InitInstance()
    {
      m_pMainWnd = new CResFrame;
      m_pMainWnd->ShowWindow(SW_SHOW);
      m_pMainWnd->UpdateWindow();

      return TRUE;
    }
};

CResApp theApp;
```

2. Test the application:



3. After viewing the window, close it and return to MSVC.

3.9.3 Frame Loading

So far, in order to create a window, we learned to use the **CFrameWnd::Create()** method. Because that method does not recognize resources, we resolved to use the **AfxRegisterWndClass()** function to register the necessary resources before calling the **CFrameWnd::Create()** method. The MFC library provides another, simpler, technique to create a window. This is done using the **CFrameWnd::LoadFrame()** method.

To create a window with one simple call, first create the necessary resources, namely an accelerator table, a menu, an icon, a string table (and possible a toolbar). The only rule to follow is that all of these resources must have the same name. As a habit, the common name used for resources is IDR_MAINFRAME.

The syntax of the **LoadFrame()** method is:

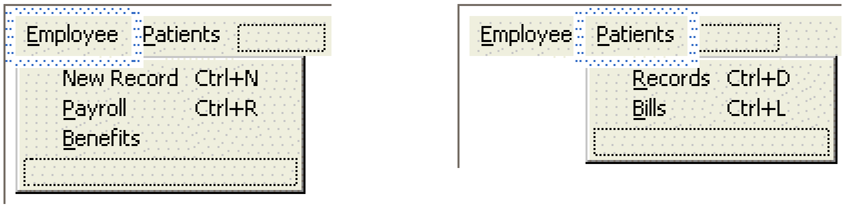
```
BOOL LoadFrame(UINT nIDResource,
               DWORD dwDefaultStyle = WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
               CWnd* pParentWnd = NULL,
               CCreateContext* pContext = NULL );
```

The only required argument to the **LoadFrame()** method is the identifier shared by the resources. Like the **Create()** method, you can use **LoadFrame()** to specify the style of the window. Its characteristics are specified as the *dwDefaultStyle* argument. If this window has a parent, you can specify it using the *pParentWnd* argument.

Practical Learning: Loading a Frame

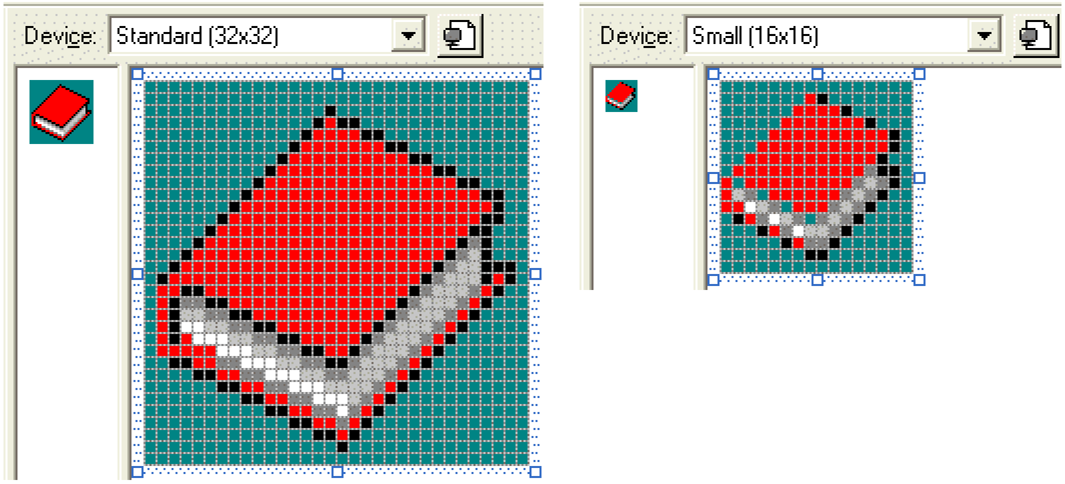
1. Create a new project named **Frame Loader** and stored in **C:\Programs\MSVC Exercises**
   Make sure you create it as a Windows Application with an Empty Project
2. Access the application's settings or properties and specify that you want to use MFC As A Shared DLL

3. Add a new menu resource as follows:

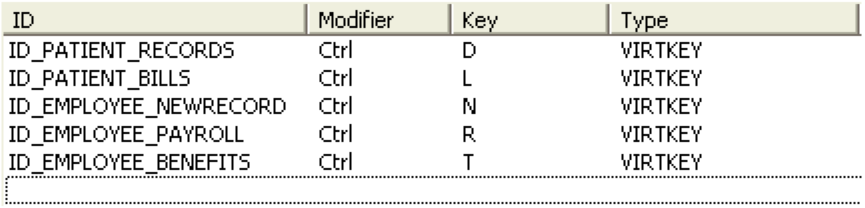


4. Change the ID of the menu from IDR_MENU1 to IDR_MAINFRAME and save the resource as FrmLoad.rc
If using MSVC 6, add the .rc file to the project (Project -> Add to Project -> File, Form Loader.rc)

5. Create a new Icon identified as **IDR_MAINFRAME** and design it as follows:

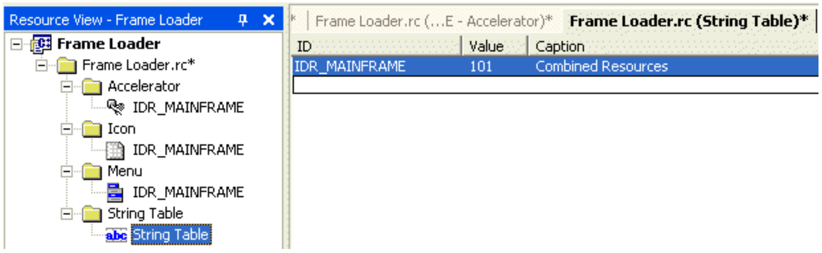


6. Create a new accelerator table identified as **IDR_MAINFRAME** as follows:



ID	Modifier	Key	Type
ID_PATIENT_RECORDS	Ctrl	D	VIRTKEY
ID_PATIENT_BILLS	Ctrl	L	VIRTKEY
ID_EMPLOYEE_NEWRECORD	Ctrl	N	VIRTKEY
ID_EMPLOYEE_PAYROLL	Ctrl	R	VIRTKEY
ID_EMPLOYEE_BENEFITS	Ctrl	T	VIRTKEY

7. Create a String Table and add a string identified as **IDR_MAINFRAME** with a Caption as Combined Resources


Resource View - Frame Loader
Frame Loader.rc (...E - Accelerator)*
Frame Loader.rc (String Table)*

```
Frame Loader
  Frame Loader.rc*
    Accelerator
      IDR_MAINFRAME
    Icon
      IDR_MAINFRAME
    Menu
      IDR_MAINFRAME
    String Table
      String Table
```

ID	Value	Caption
IDR_MAINFRAME	101	Combined Resources

8. Create a new C++ source file and name it Main

9. In the Main.cpp file, create the application as follows:

```
#include <afxwin.h>
#include "resource.h"

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ()
    {
            LoadFrame(IDR_MAINFRAME);
    }
};

class CMainApp: public CWinApp
{
public:
    BOOL InitInstance()
    {
            m_pMainWnd = new CMainFrame ;
            m_pMainWnd->ShowWindow(SW_SHOW);
            m_pMainWnd->UpdateWindow();

            return TRUE;
    }
};

CMainApp theApp;
```

10. Test the application.

Chapter 4: Messages and Events

4.1 Introduction to Messages

4.1.1 Overview

Some of your applications will be made of various objects. Most of the time, more than one application is running on the computer. These two scenarios mean that the operating system is constantly asked to perform some assignments. Because there can be so many requests presented unpredictably, the operating system leaves it up to the objects to specify what they want, when they want it, and what behavior or result they expect.

The Microsoft Windows operating system cannot predict what kinds of requests one object would need to be taken care of and what type of assignment another object would need. To manage all these assignments and requests, the objects send messages, one message at a time, to the operating system. For this reason, Microsoft Windows is said to be a message-driven operating system.

The messages are divided in various categories but as mentioned already, each object has the responsibility to decided what message to send and when. Therefore, most of the messages we will review here are part of a window frame. Others will be addressed when necessary.

Once a control has composed a message, it must send it to the right target which could be the operating system. In order to send a message, a control must create an event. It is also said to fire an event. To make a distinction between the two, a message's name usually starts with WM_ which stands for Window Message. The name of an event usually starts with On which indicates an action. Remember, the message is what needs to be sent. The event is the action of sending the message.

4.1.2 A Map of Messages

For the compiler to manage messages, they should be included in the class definition. The list of messages starts on a section driven by, but that ends with, the **DECLARE_MESSAGE_MAP** macro. The section can be created as follows:

```
#include <afxwin.h>

class CSimpleFrame : public CFrameWnd
{
public:
	CSimpleFrame();

	DECLARE_MESSAGE_MAP()
};
```

The **DECLARE_MESSAGE_MAP** macro should be provided at the end of the class definition. The actual messages (as we will review them shortly) should be listed just above the **DECLARE_MESSAGE_MAP** line. This is just a rule. In some circumstances, and for any reason, you may want, or have, to provide one message or a few messages under the **DECLARE_MESSAGE_MAP** line.

To implement the messages, you should/must create a table of messages that your program is using. This table uses two delimiting macros. Its starts with a **BEGIN_MESSAGE_MAP** and ends with an **END_MESSAGE_MAP** macros. The **BEGIN_MESSAGE_MAP** macro takes two arguments, the name of your class and the MFC class you derived your class from. An example would be:

```
BEGIN_MESSAGE_MAP(CSimpleFrame, CFrameWnd)
```

Like the **DECLARE_MESSAGE_MAP** macro, **END_MESSAGE_MAP** takes no argument. Its job is simple to specify the end of the list of messages. The table of messages can be created as follows:

```
#include <afxwin.h>
#include "resource.h"

class CMainFrame : public CFrameWnd
{
public:
	CMainFrame ();

	DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
	LoadFrame(IDR_MAINFRAME);
}

class CMainApp: public CWinApp
{
public:
	BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)

END_MESSAGE_MAP()

BOOL CMainApp::InitInstance()
{
	m_pMainWnd = new CMainFrame ;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}

CMainApp theApp;
```

There various categories of messages the operating system receives. Some of them come from the keyboard, some from the mouse, and some others from various other origins. For example, some messages are sent by the application itself while some other messages are controlled by the operating.

Practical Learning: Creating a Map of Messages

1. Create a new and empty Win32 Project located in C:\Programs\MSVC Exercises and set its the Name to **Messages1**
2. Specify that you want to **Use MFC in a Shared DLL**
3. Create a C++ file and name it **Exercise**
4. To create a frame for the window, in the Exercise.cpp file, type the following:

```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ();

protected:

    DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
           CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
    BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)

END_MESSAGE_MAP()

BOOL CExerciseApp::InitInstance()
{
    m_pMainWnd = new CMainFrame ;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CExerciseApp theApp;
```

5. Test the application and return to MSVC

4.2 Windows Messages

4.2.1 Window Creation

WM_CREATE: When an object, called a window, is created, the frame that creates the objects sends a message identified as **ON_WM_CREATE.** Its syntax is:

```
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
```

This message calls the **CFrameWnd::Create()** method to create a window. To do that, it takes one argument, which is a pointer to a **CREATESTRUCT** class. The **CREATESTRUCT** class provides all the information needed to create a window. It is defined as follows:

```
typedef struct tagCREATESTRUCT {
   LPVOID    lpCreateParams;
   HANDLE    hInstance;
   HMENU     hMenu;
   HWND      hwndParent;
   int       cy;
   int       cx;
   int       y;
   int       x;
   LONG      style;
   LPCSTR    lpszName;
   LPCSTR    lpszClass;
   DWORD     dwExStyle;
} CREATESTRUCT;
```

This class provides the same types of information as the **WNDCLASS** object. When sending the **OnCreate()** message, the class is usually created without your intervention but when calling it, you should check that the window has been created. This can be done by checking the result returned by the **OnCreate()** message from the parent class. If the message returns 0, the window was created. If it returns -1, the class was not created or it would simply be destroyed. This can be done as follows:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	// Create the window and make sure it doesn't return -1
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
		return -1;
	// else is implied
	return 0;
}
```

To use this message, in the class definition, type its syntax. In the message table, type the name of the message **ON_WM_CREATE():**

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
	ON_WM_CREATE()
END_MESSAGE_MAP()
```

Practical Learning: Creating a Window

1. To create an **ON_WM_CREATE** message, change the file as follows:

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

```
class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ();

protected:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
        CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
    BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // Call the base class to create the window
    if( CFrameWnd::OnCreate(lpCreateStruct) == 0)
    {
            // If the window was successfully created, let the user know
            MessageBox("The window has been created!!!");
           // Since the window was successfully created, return 0
            return 0;
    }
    // Otherwise, return -1
    return -1;
}

BOOL CExerciseApp::InitInstance()
{
    m_pMainWnd = new CMainFrame ;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CExerciseApp theApp;
```

2. Test the application and return to MSVC

4.2.2 Window's Showing State

WM_SHOWWINDOW: After creating a window, it needs to be displayed. Also, if the window was previously hidden, you can decide to show it. On the other hand, if a window is displaying, you may want to hide it, for any reason you judge necessary. To take any of these actions, that is, to show or hide a window, you must send the **ON_WM_SHOWWINDOW** message. The syntax of this message is:

```
afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
```

When using this message, *bShow* is a Boolean argument determines what state to apply to the window. If it is **TRUE**, the window needs to be displayed. If it is **FALSE**, the window must be hidden.

nStatus is a positive integer that can have one of the following values:

Value	Description
SW_PARENTCLOSING	If the window that sent this message is a frame, the window is being minimized. If the window that sent this message is hosted by another window, the window is being hidden.
SW_PARENTOPENING	If the window that sent this message is a frame, the window is being restored. If the window that sent this message is hosted by another window, the window is displaying
0	The message was sent from a CWnd::ShowWindow() method

Practical Learning: Showing a Window

1. To maximize the window at startup, change the program as follows:

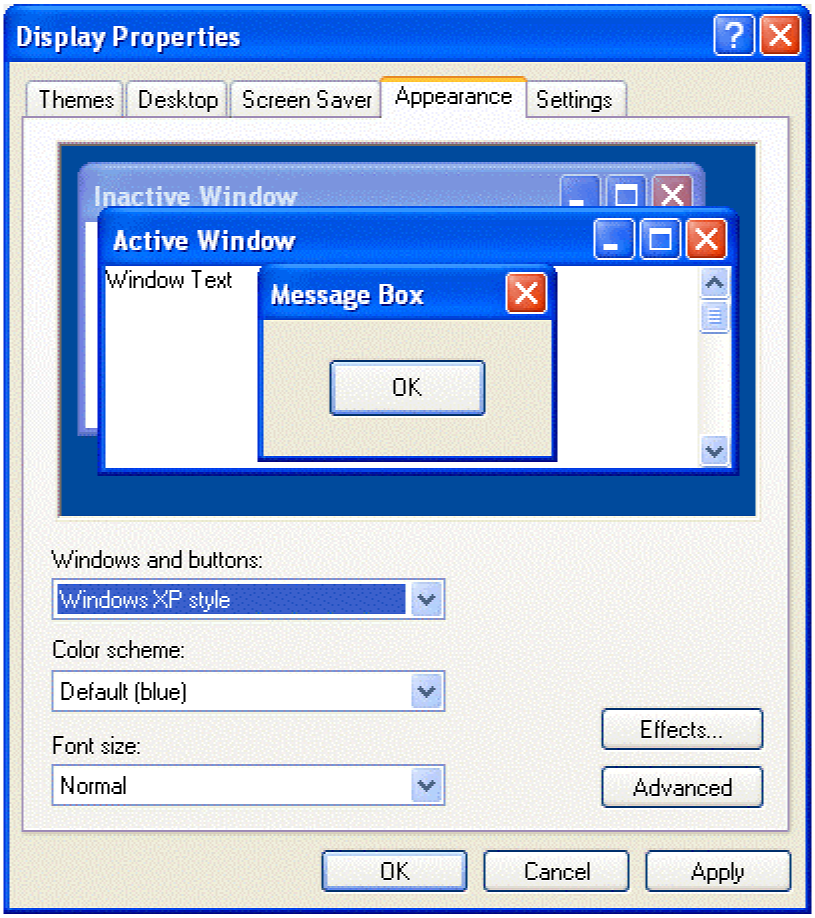
```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ();

protected:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
    DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
           CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
    BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
```

```
    ON_WM_CREATE()
    ON_WM_SHOWWINDOW()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    // Call the base class to create the window
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    return 0;
}

void CMainFrame::OnShowWindow(BOOL bShow, UINT nStatus)
{
    CFrameWnd::OnShowWindow(bShow, nStatus);

    // TODO: Add your message handler code here
    ShowWindow(SW_MAXIMIZE);
}

BOOL CExerciseApp::InitInstance()
{
    m_pMainWnd = new CMainFrame ;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CExerciseApp theApp;
```

2. Test the program and return to MSVC

4.2.3 Window Activation

WM_ACTIVATE: When two or more windows are running on the computer, only one can receive input from the user, that is, only one can actually be directly used at one particular time. Such a window has a title bar with the color identified in Control Panel as Active Window. The other window(s), if any, display(s) its/their title bar with a color called **Inactive Window**:



Windows and buttons: Windows XP style

Color scheme: Default (blue)

Font size: Normal

To manage this setting, the windows are organized in a 3-dimensional coordinate system and they are incrementally positioned on the Z coordinate, which defines the (0, 0, 0) origin on the screen (actually on the top-left corner of your monitor) with Z coordinate coming from the screen towards you.

In order to use a window other than the one that is active, you must activate it. To do this, you can send a message called **ON_WM_ACTIVATE**. The syntax of this message is:

```
afx_msg void OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized);
```

This message indeed does two things: it activates a window of your choice, or brings it to the front, and deactivates the other window(s) or sends it/them to the back of the window that is being activates. The nState argument specifies what action to apply. It is a constant that can assume of the following values:

Value	Description
WA_ACTIVE	Used to activate a window without using the mouse, may be by pressing Alt + Tab
WA_INACTIVE	Used to deactivate a window
WA_CLICKACTIVE	Used to activate a window using the mouse

If this message was sent by the window that is being activated, *pWndOther* designates the other window, the one being deactivated. If this message was sent by the window that is being deactivated, pWndOther designates the other window, the one being activated.

If the window that sent this message is being restored from its deactivation, pass the *bMinimized* value as TRUE.

When calling this message, before implementing the custom behavior you want, first call its implementation from the parent class:

```
void CMainFrame::OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized)
{
        CFrameWnd::OnActivate(nState, pWndOther, bMinimized);

        // TODO: Add your message handler code here
}
```

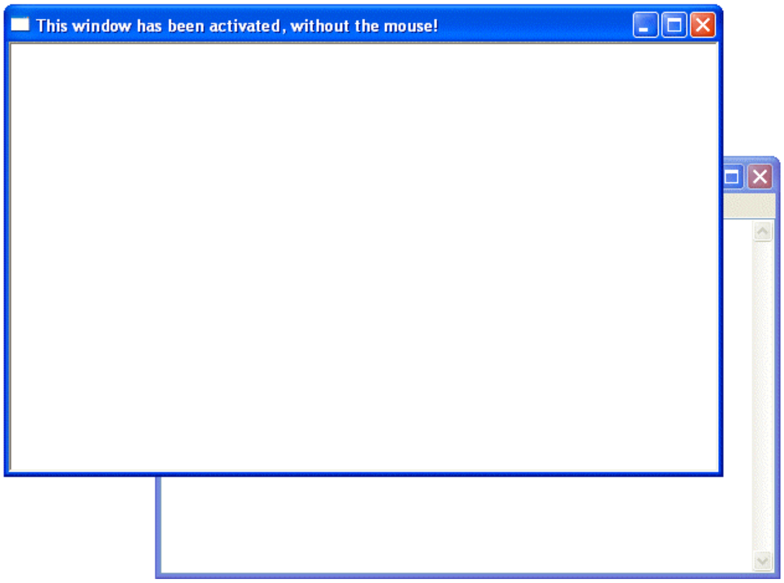
Practical Learning: Activating a Window

1. To activate a window that has been created, change the file as follows:

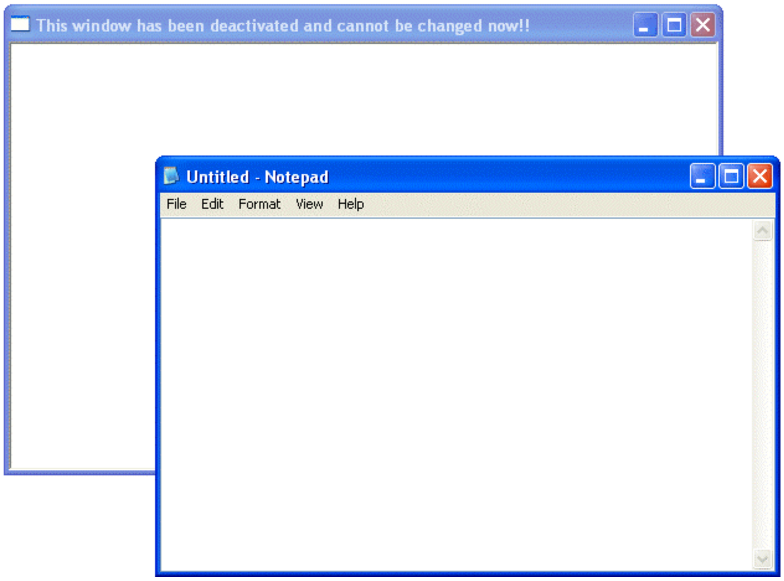
```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
        CMainFrame ();

protected:
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
        afx_msg void OnActivate(UINT nState, CWnd* pWndOther,
                                BOOL bMinimized);
        DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
        // Create the window's frame
        Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
            CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
        BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        ON_WM_CREATE()
        ON_WM_SHOWWINDOW()
        ON_WM_ACTIVATE()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        // Call the base class to create the window
        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;
        return 0;
}
```

```
void CMainFrame::OnShowWindow(BOOL bShow, UINT nStatus)
{
        CFrameWnd::OnShowWindow(bShow, nStatus);

        // TODO: Add your message handler code here
        //ShowWindow(SW_SHOW);
}

void CMainFrame::OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized)
{
        CFrameWnd::OnActivate(nState, pWndOther, bMinimized);

        // TODO: Add your message handler code here
        switch( nState )
        {
        case WA_ACTIVE:
                MessageBox("This window has been activated, without the mouse!");
                break;
        case WA_INACTIVE:
                MessageBox("This window has been deactivated and cannot be changed now!!");
                break;
        case WA_CLICKACTIVE:
                MessageBox("This window has been activated using the mouse!!!");
                break;
        }
}

BOOL CExerciseApp::InitInstance()
{
        m_pMainWnd = new CMainFrame ;
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();

        return TRUE;
}

CExerciseApp theApp;
```

2. Test the application. While it is displaying, open Notepad
3. Display each window using the mouse. Then activate your window using Alt+Tab



This window has been activated, without the mouse!



This window has been deactivated and cannot be changed now!!

4. Return to MSVC

4.2.4 Window Painting

WM_PAINT: Whether you have just created a window or you want to show it, you must ask the operating system to display it, showing its appearance. To display such a window, the operating system would need its location (left and top measures) and its dimension (width and height). This is because the window must be painted. Also, if the window was hidden somewhere such as behind another another window or was minimized, when it comes up, the operating system needs to paint it. To do this, the window that needs to be painted must send a message called **ON_WM_PAINT**. This message does not return anything but in its body, you can define what needs to be painted and how you want the job to be done. The syntax of this message is simply:

```
afx_msg void OnPaint()
```

Practical Learning: Using the Paint Message

1. To use the structure of the **WM_PAINT** message, change the program as follows:

```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
	CMainFrame ();

protected:
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
	afx_msg void OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized);
	afx_msg void OnPaint();
	DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
	// Create the window's frame
	Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
		CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
	BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
	ON_WM_CREATE()
	ON_WM_SHOWWINDOW()
	ON_WM_ACTIVATE()
	ON_WM_PAINT()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
			return -1;
	return 0;
}

void CMainFrame::OnShowWindow(BOOL bShow, UINT nStatus)
{
	CFrameWnd::OnShowWindow(bShow, nStatus);

	// TODO: Add your message handler code here
	//ShowWindow(SW_SHOW);
}

void CMainFrame::OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized)
{
    CFrameWnd::OnActivate(nState, pWndOther, bMinimized);

    // TODO: Add your message handler code here
    switch( nState )
    {
    case WA_ACTIVE:
            MessageBox("This window has been activated, without the mouse!");
            break;
    case WA_INACTIVE:
            MessageBox("This window has been deactivated and cannot be
changed now!!");
            break;
    case WA_CLICKACTIVE:
            MessageBox("This window has been activated using the mouse!!!");
            break;
    }
}

void CMainFrame::OnPaint()
{
    CFrameWnd::OnPaint();

    MessageBox("The window has been painted<==>");
}

BOOL CExerciseApp::InitInstance()
{
    m_pMainWnd = new CMainFrame ;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CExerciseApp theApp;
```

2. Test the application and return to MSVC

4.2.5 Window Sizing

WM_SIZE: When using an application, one of the actions a user can perform on a window is to change its size, provided the window allows this. Also, some time to time, if the window allows it, the user can minimize, maximize, or restore a window. Whenever any of these actions occur, the operating system must keep track of the size of a window. When the size of a window has changed, the window sends the ON_WM_SIZE message. Its syntax is:

```
afx_msg void OnSize(UINT nType, int cx, int cy);
```

The *nType* argument specifies what type of action to take. It can have one of the following values:

Value	Description
SIZE_MINIMIZED	The window has been minimized
SIZE_MAXIMIZED	The window has been maximized
SIZE_RESTORED	The window has been restored from being maximized or

	minimized
SIZE_MAXHIDE	Another window, other than this one, has been maximized
SIZE_MAXSHOW	Another window, other than this one, has been restored from being maximized or minimized

The cx argument specifies the new width of the client area of the window
The cy argument specifies the new height of the client area of the window.
To use this message, you should first call its implementation in the parent class before implementing the behavior you want. This can be done as follows:

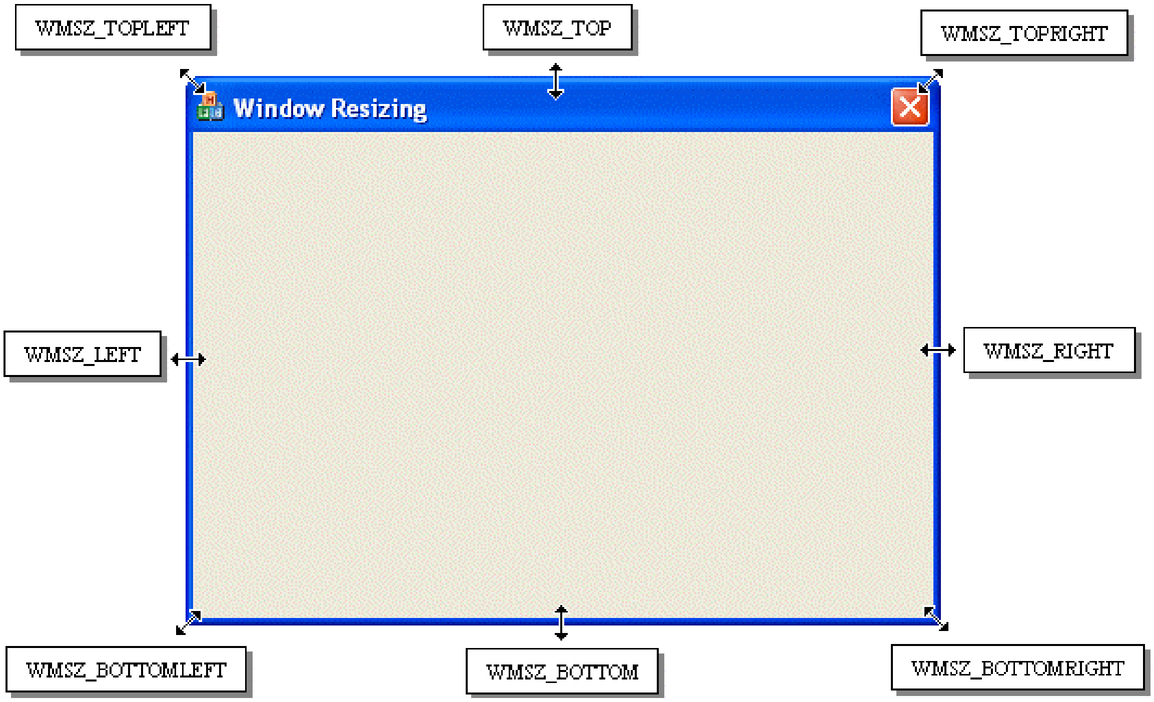
```
void CAnyWindow::OnSize(UINT nType, int cx, int cy)
{
	CParentClass::OnSize(nType, cx, cy);

	// TODO: Add your message handler code here
}
```

WM_SIZING: While the user is changing the size of a window, a message called ON_WM_SIZING is being sent. Its syntax is:

```
afx_msg void OnSizing(UINT nSide, LPRECT lpRect);
```

When a user is resizing a window, he or she typically drags one of the borders or corners on a direction of his or her choice. The first argument, nSize, indicates what edge is being moved when resizing the window. It can have one of the following values:

Value	Description
WMSZ_BOTTOM	Bottom edge
WMSZ_BOTTOMLEFT	Bottom-left corner
WMSZ_BOTTOMRIGHT	Bottom-right corner
WMSZ_LEFT	Left edge
WMSZ_RIGHT	Right edge
WMSZ_TOP	Top edge
WMSZ_TOPLEFT	Top-left corner
WMSZ_TOPRIGHT	Top-right corner



The second argument, lpRect, is the new rectangle that will enclose the window after the window has been moved, resized, or restored.

4.2.6 Window Moving

WM_MOVE: When a window has been moved, the operating system needs to update its location. Therefore, the window sends a message called ON_WM_MOVE. Its syntax is:

```
afx_msg void OnMove(int x, int y);
```

The first argument of this message specifies the left horizontal location of the left border of the window after the window has been moved. The second argument represents the vertical position of the top border of the window after the window has been moved.
If you want to send this message, you should first call its implementation in the parent class

WM_MOVING: While the user is moving a window, it (the window sends) an ON_WM_MOVING message. Its syntax is:

```
afx_msg void OnMoving( UINT nSide, LPRECT lpRect );
```

The first argument, nSide, is the edge of the window that is being moved. It is the same as for the ON_MOVE message.
The lpRect is the target dimension of the window that is being moved. That is, it contains the new dimensions of the window as it is being moved.

Practical Learning: Moving a Window

1. To see the effect of moving a window, change the file as follows:

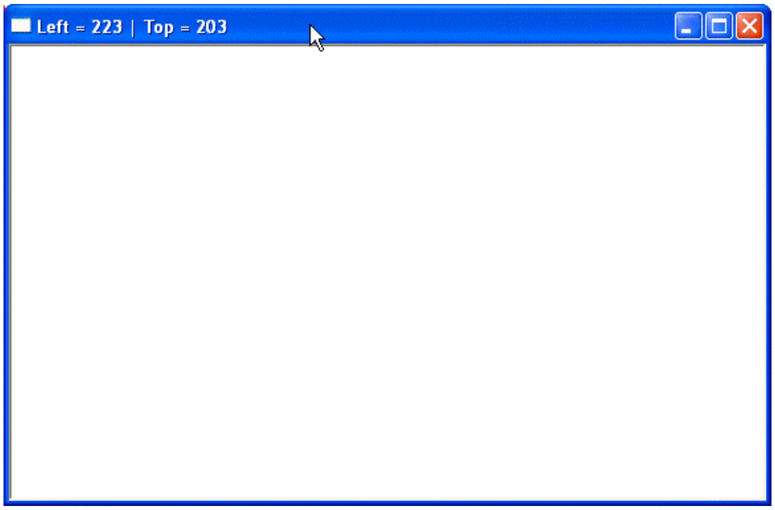
```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
```

```
{
public:
	CMainFrame ();

protected:
	afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
	afx_msg void OnShowWindow(BOOL bShow, UINT nStatus);
	afx_msg void OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized);
	afx_msg void OnPaint();
	afx_msg void OnSize(UINT nType, int cx, int cy);
	afx_msg void OnMove(int x, int y);
	DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
	// Create the window's frame
	Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
	       CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
	BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
	ON_WM_CREATE()
	ON_WM_SHOWWINDOW()
	ON_WM_ACTIVATE()
	ON_WM_PAINT()
	ON_WM_SIZE()
	ON_WM_MOVE()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct ) == -1)
		return -1;
	return 0;
}

void CMainFrame::OnShowWindow(BOOL bShow, UINT nStatus)
{
	CFrameWnd::OnShowWindow(bShow, nStatus);

	// TODO: Add your message handler code here
	//ShowWindow(SW_SHOW);
}

void CMainFrame::OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized)
{
	CFrameWnd::OnActivate(nState, pWndOther, bMinimized);

	// TODO: Add your message handler code here
	switch( nState )
	{
	case WA_ACTIVE:
```

```
                MessageBox("This window has been activated, without the mouse!");
                break;
        case WA_INACTIVE:
                MessageBox("This window has been deactivated and cannot be
changed now!!");
                break;
        case WA_CLICKACTIVE:
                MessageBox("This window has been activated using the mouse!!!");
                break;
        }
}

void CMainFrame::OnPaint()
{
        CFrameWnd::OnPaint();

        MessageBox("The window has been painted<==>");
}

void CMainFrame::OnSize(UINT nType, int cx, int cy)
{
/*      CMainFrame::OnSize(nType, cx, cy);

        // TODO: Add your message handler code here
        char *MsgToShow = new char[20];
        char *MsgCoord  = new char[20];

        switch(nType)
        {
        case SIZE_MINIMIZED:
                strcpy(MsgToShow, "Minimized ");
                break;
        case SIZE_MAXIMIZED:
                strcpy(MsgToShow, "Maximized ");
                break;
        case SIZE_RESTORED:
                strcpy(MsgToShow, "Restored ");
                break;
        case SIZE_MAXHIDE:
                strcpy(MsgToShow, "Maximized Not Me ");
                break;
        case SIZE_MAXSHOW:
                strcpy(MsgToShow, "Restored Not Me ");
                break;
        }

        sprintf(MsgCoord, "Left = %d | Top = %d", cx, cy);
        strcat(MsgToShow, MsgCoord);
        MessageBox(MsgToShow);
*/
}

void CMainFrame::OnMove(int x, int y)
{
        CFrameWnd::OnMove(x, y);

        // TODO: Add your message handler code here
        char *MsgCoord  = new char[20];
```

```
	sprintf(MsgCoord, "Left = %d | Top = %d", x, y);

	MessageBox(MsgCoord);
}

BOOL CExerciseApp::InitInstance()
{
	m_pMainWnd = new CMainFrame ;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}

CExerciseApp theApp;
```

2. Test the application:


Left = 223 | Top = 203

3. Return to MSVC

4.2.7 Window Destruction

WM_DESTROY: Once the window has been used and the user has closed it, the window must send a message to the operating system to destroy it. The message sent is called ON_WN_DESTROY and its syntax is:

```
afx_msg void OnDestroy( );
```

This message takes no argument but you can use its body to do any last minute assignment as needed. For example, you can use it either to prevent the window from being closed or you can enquire whether the user really wants to close the window.

4.3 Command Messages

4.3.1 Definition

One of the main features of a graphical application is to present Windows controls and resources that allow the user to interact with the machine. Examples of controls that we will learn are buttons, list boxes, combo boxes, etc. One type of resource we introduced in the previous lesson is the menu. Such controls and resources can initiate their own messages when the user clicks them. A message that emanates from a Windows control or a resource is called a command message.

4.3.2 Creating a Command Message

You can start by declaring a framework method. Here is an example:

```
afx_msg void OnLetItGo();
```

Then you can define the method as you see fit:

```
void CMainFrame::OnLetItGo()
{
	// Something to do here
}
```

The framework allows you to associate a member function to a command message. This is done using the ON_COMMAND macro. Its syntax:

```
ON_COMMAND(IDentifier, MethodName)
```

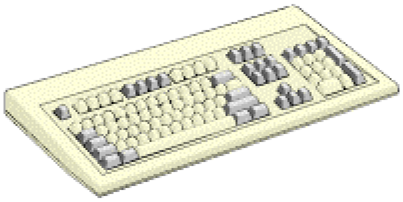
The first argument, IDentifier, of this macro must be the identifier of the menu item or Windows control you want to associate with a method.
The second argument, MethodName, must be the name of the member function that will implement the action of the menu item or Windows control such a button.
Imagine you have a menu item IDentified as ID_ACTION_LETITGO that you want to associate with the above OnLetItGo() method. You can use the ON_COMMAND macro as follows:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
	ON_COMMAND(ID_ACTION_LETITGO, OnLetItGo)
END_MESSAGE_MAP()
```

In the same way, you can create as many command messages as necessary.

4.4 Keyboard Messages

4.4.1 Introduction



A keyboard is a hardware object attached to the computer. By default, it is used to enter recognizable symbols, letters, and other characters on a control. Each key on the keyboard displays a symbol, a letter, or a combination of those, to give an indication of what the key could be used for.

The user typically presses a key, which sends a signal to a program. The signal is analyzed to find its meaning. If the program or control that has focus is equipped to deal with the signal, it may produce the expected result. If the program or control cannot figure out what to do, it ignores the action.

Each key has a code that the operating system can recognize. This code is known as the virtual key code and they are as follows:

Virtual Key	Used for	Virtual Key	Used for
VK_F1	F1	VK_F2	F2
VK_F3	F3	VK_F4	F4
VK_F5	F5	VK_F6	F6
VK_F7	F7	VK_F8	F8
VK_F9	F9	VK_F10	F10
VK_F11	F11	VK_F12	F12
VK_SCROLL	Scroll Lock	VK_SNAPSHOT	Prt Scrn (Depends on keyboard)
VK_PAUSE	Pause/Break	VK_TAB	Tab
VK_BACK	Backspace	VK_CAPITAL	Caps Lock
VK_SHIFT	Shift	VK_CONTROL	Ctrl
VK_MENU	Alt	VK_ESCAPE	Escape
VK_RETURN	Enter	VK_SPACE	Space Bar
VK_INSERT	Insert	VK_HOME	Home
VK_PRIOR	Page Up	VK_DELETE	Delete
VK_END	End	VK_NEXT	Page Down
VK_UP	Up Arrow Key	VK_RIGHT	Right Arrow Key
VK_DOWN	Down Arrow Key	VK_LEFT	Left Arrow Key
VK_LWIN	Left Windows Key	VK_RWIN	Right Windows Key
VK_APPS	Applications Key		
The following keys apply to the Numeric Keypad			
VK_NUMLOCK	Num Lock		
VK_NUMPAD0	0	VK_NUMPAD1	1
VK_NUMPAD2	2	VK_NUMPAD3	3
VK_NUMPAD4	4	VK_NUMPAD5	5
VK_NUMPAD6	6	VK_NUMPAD7	7
VK_NUMPAD8	8	VK_NUMPAD9	9
VK_DIVIDE	/	VK_MULTIPLY	*
VK_SUBTRACT	-	VK_ADD	+
VK_SEPARATOR		VK_DECIMAL	.

There are actually more keys than that but the above are the most frequently used.

Practical Learning: Introducing Keyboard Messages

1. Create a new and empty Win32 Project located in C:\Programs\MSVC Exercises and set its the Name to Messages2
2. Specify that you want to Use MFC in a Shared DLL
3. Create a C++ file and name it Exercise
4. To create a frame for the window, in the Exercise.cpp file, type the following:

```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
	CMainFrame ();
protected:
	afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);

	DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
	// Create the window's frame
	Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
		CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
	BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
	ON_WM_CREATE()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
		return -1;
	return 0;
}

BOOL CExerciseApp::InitInstance()
{
	m_pMainWnd = new CMainFrame ;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}

CExerciseApp theApp;
```

5. Test the application and return to MSVC

4.4.2 The Key Down Effect

When we think of the keyboard, the first thing that comes in mind if to press a key. When a keyboard key is pressed, a message called WM_KEYDOWN. Its syntax is:

```
afx_msg void OnKeyDown( UINT nChar, UINT nRepCnt, UINT nFlags );
```

The first argument, nChar, specifies the virtual code of the key that was pressed.
The second argument, nRepCnt, specifies the number of times counted repeatedly as the key was held down.
The nFlags argument specifies the scan code, extended-key flag, context code, previous key-state flag, and transition-state flag.

Practical Learning: Sending Key Down Messages

1. To experiment with the ON_KEYDOWN message, change the file as follows:

```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ();

protected:
    afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);

    DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
        CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
    BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_KEYDOWN()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
            return -1;
    return 0;
}

void CMainFrame::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar)
    {
    case VK_RETURN:
            MessageBox("You pressed Enter");
            break;
    case VK_F1:
            MessageBox("Help is not available at the moment");
            break;
    case VK_DELETE:
            MessageBox("Can't Delete This");
            break;
    default:
            MessageBox("Whatever");
    }
}

BOOL CExerciseApp::InitInstance()
{
     m_pMainWnd = new CMainFrame ;
     m_pMainWnd->ShowWindow(SW_SHOW);
     m_pMainWnd->UpdateWindow();

     return TRUE;
}

CExerciseApp theApp;
```

2. Test the application and return to MSVC

4.4.3 The Key Up Effect

When we think of the keyboard, the first thing that comes in mind might refer to typing, which consists of pressing a key and releasing it immediately. As this is done, a key is pressed down and brought back up. When the user is releasing a key a WM_KEYUP message is sent. Its syntax is:

```
afx_msg void OnKeyUp(UINT nChar, UINT nRepCnt, UINT  nFlags);
```

The first argument, nChar, is the code of the key that was pressed.
The second argument, nRepCnt, specifies the number of times counted repeatedly as the key was held down.
The nFlags argument specifies the scan code, extended-key flag, context code, previous key-state flag, and transition-state flag.

4.5 Mouse Messages

4.5.1 Introduction

The mouse is another object that is attached to the computer allowing the user to interact with the machine. The mouse and the keyboard can each accomplish some tasks that are not normally available on the other and both can accomplish some tasks the same way.

The mouse is equipped with two, three, or more buttons. When a mouse has two buttons, one is usually located on the left and the other is located on the right. When a mouse has three buttons, one is in the middle of the other two. The mouse is used to select a point or position on the screen. Once the user has located an item, which could also be an empty space, a letter or a word, he or she would position the mouse pointer on it. To actually use the mouse, the user would press either the left, the middle (if any), or the right button. If the user presses the left button once, this action is called Click. If the user presses the right mouse button, the action is referred to as Right-Click. If the user presses the left button twice and very fast, the action is called Double-Click.

4.5.2 Mouse-Down Messages

Imagine the user has located a position or an item on a document and presses one of the mouse buttons. While the button is pressed and is down, a button-down message is sent, depending on the button that was pressed.

If the left mouse button was pressed, an **ON_WM_LBUTTONDOWN** message is sent. The syntax of this message is:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

If the right mouse button was pressed, an ON_WM_RBUTTONDOWN message is sent. Its syntax is:

```
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
```

The first argument, nFlags, specifies what button is down or what keyboard key and what mouse button are down. It is a constant integer that can have one of the following values:

Value	Description
MK_CONTROL	A Ctrl key is held down
MK_LBUTTON	The left mouse button is down
MK_MBUTTON	The middle mouse button is down
MK_RBUTTON	The right mouse button is down
MK_SHIFT	A Shift key is held down

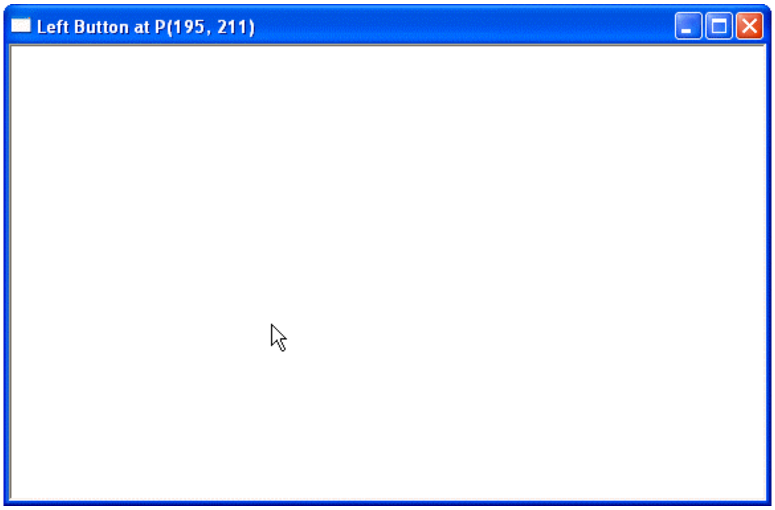
The point argument specifies the measure from the left and the top borders of the window to the mouse pointer.

Practical Learning: Sending a Mouse Down Message

1. To experiment with the mouse down effect, change the program as follows:

```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
	CMainFrame ();

protected:
	afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
	afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
	afx_msg void OnLButtonDown(UINT nFlags, CPoint point);

	DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
	// Create the window's frame
	Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
	       CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
	BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_KEYDOWN()
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
            return -1;
    return 0;
}

void CMainFrame::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar)
    {
    case VK_RETURN:
            MessageBox("You pressed Enter");
            break;
    case VK_F1:
            MessageBox("Help is not available at the moment");
            break;
    case VK_DELETE:
            MessageBox("Can't Delete This");
            break;
    default:
            MessageBox("Whatever");
    }
}

void CMainFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
    char *MsgCoord  = new char[20];

    sprintf(MsgCoord, "Left Button at P(%d, %d)", point.x, point.y);

    MessageBox(MsgCoord);
}

BOOL CExerciseApp::InitInstance()
{
    m_pMainWnd = new CMainFrame ;
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}

CExerciseApp theApp;
```

2. Test the program:



3. Return to MSVC

4.5.3 The Mouse-Up Messages

After pressing a mouse button, the user usually releases it. While the button is being released, a button-up message is sent and it depends on the button, left or right, that was down.

If the left mouse is being released, the ON_WM_LBUTTONUP message is sent. Its syntax is:

```
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
```

If the right mouse is being released, the ON_WM_TBUTTONUP message is sent. Its syntax is:

```
afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
```

The first argument, nFlags, specifies what button, right or middle, is down or what keyboard key and what mouse button were down. It is a constant integer that can have one of the following values:

Value	Description
MK_CONTROL	A Ctrl key was held down
MK_MBUTTON	The middle mouse button was down
MK_RBUTTON	The right mouse button was down
MK_SHIFT	A Shift key was held

The point argument specifies the measure from the (0, 0) origin of the window to the mouse pointer.

Practical Learning: Sending a Mouse Down Message

1. To experiment with a mouse up message, change the program as follows:

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

```
class CMainFrame : public CFrameWnd
{
public:
    CMainFrame ();

protected:
    afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);

    DECLARE_MESSAGE_MAP()
};

CMainFrame::CMainFrame()
{
    // Create the window's frame
    Create(NULL, "Windows Application", WS_OVERLAPPEDWINDOW,
          CRect(120, 100, 700, 480), NULL);
}

class CExerciseApp: public CWinApp
{
public:
    BOOL InitInstance();
};

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_KEYDOWN()
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONUP()
END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    return 0;
}

void CMainFrame::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar)
    {
    case VK_RETURN:
        MessageBox("You pressed Enter");
        break;
    case VK_F1:
        MessageBox("Help is not available at the moment");
        break;
    case VK_DELETE:
        MessageBox("Can't Delete This");
        break;
    default:
        MessageBox("Whatever");
    }
}
```

```
void CMainFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
	char *MsgCoord  = new char[20];

	sprintf(MsgCoord, "Left Button at P(%d, %d)", point.x, point.y);

	MessageBox(MsgCoord);
}

void CMainFrame::OnRButtonUp(UINT nFlags, CPoint point)
{
	MessageBox("Right Mouse Button Up");
}

BOOL CExerciseApp::InitInstance()
{
	m_pMainWnd = new CMainFrame ;
	m_pMainWnd->ShowWindow(SW_SHOW);
	m_pMainWnd->UpdateWindow();

	return TRUE;
}

CExerciseApp theApp;
```

2. Execute the program. To test it, click in the middle of the window and hold the mouse down
3. Then release the mouse. Notice that the title bar displays the new message only when the mouse is up.
4. Return to MSVC

4.5.4 The Double-Click Message

Instead of pressing and simply releasing a mouse button, a classic action the user can perform with the mouse is to double-click an object. When this is done, a double-click message is sent. The message to consider actually depends on the button that was double-pressed.

If the double-click was performed using the left mouse button, the WM_LBUTTONDBLCLK message is sent and its syntax is:

```
afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
```

If the action was performed using the right mouse button, the WM_RBUTTONDBLCLK message would be sent. Its syntax is:

```
afx_msg void OnRButtonDblClk(UINT nFlags, CPoint point);
```

In both cases the nFlags argument specifies the button that was double-clicked. This argument can have one of the following values:

Value	Description
MK_CONTROL	A Ctrl key was held down when the user double-clicked
MK_LBUTTON	The left mouse button is down

MK_MBUTTON	The middle mouse button is down
MK_RBUTTON	The right mouse button is down
MK_SHIFT	A Shift key was held down when the user double-clicked

The point argument specifies the location of the mouse pointer in x (horizontal) and y (vertical) coordinates.

4.5.5 Mouse Moving

After pressing one of the mouse buttons, depending on the requirement, the use may not need to immediately release the button. Another action performed with the mouse consists of clicking and holding the mouse button down, then dragging in a chosen direction. This action refers to the mouse moving. When this is done a WM_MOUSEMOVE message is sent. Its syntax is:

```
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
```

The nFlags argument specifies what button is held down or what key is pressed in combination with the button that is held down while the mouse pointer is moving. This argument can have one of the following values:

Value	While the mouse is moving
MK_CONTROL	A Ctrl key is held down
MK_LBUTTON	The left mouse button is down
MK_MBUTTON	The middle mouse button is down
MK_RBUTTON	The right mouse button is down
MK_SHIFT	A Shift key was held down

The point argument specifies the current location of the mouse pointer in x (horizontal) and y (vertical) coordinates at the time the message is captured.

4.6 Anytime Messages

4.6.1 Introduction

The messages we have used so far belong to specific events generated at a particular time by a window. Sometimes in the middle of doing something, you may want to send a message regardless of what is going on. This is made possible by a function called **SendMessage().** Actually, there are two **SendMessage()** versions available.

The Win32 API version of the **SendMessage()** function has the following syntax:

```
LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

The MFC version is carried by the CWnd class and declared as follows:

```
LRESULT SendMessage(UINT Msg, WPARAM wParam, LPARAM lParam);
```

Because the Win32 version is considered global, if you want to use it, you must precede it with the scope access operator "::" as in:

```
::SendMessage(WhatToDo);
```

The *hWnd* argument is the object or control that is sending the message.
The *Msg* argument is the message to be sent.
The *wParam* and the *lParam* values depend on the message that is being sent.

4.6.2 Sending Messages

The advantage of using the **SendMessage()** function is that, when sending this message, it would target the procedure that can perform the task and this function would return only after its message has been processed. Because this (member) function can sometimes universally be used, that is by any control or object, the application cannot predict the type of message that **SendMessage()** is carrying. Therefore, (the probable disadvantage is that) you must know the (name or identity of the) message you are sending and you must provide accurate accompanying items (like sending a letter with the right stamp; imagine you send a sexy letter to your grand-mother in Australia about her already dead grand grand-father who is celebrating his first job while he has just become 5 years old).

In order to send a message using the **SendMessage()** function, you must know what message you are sending and what that message needs in order to be complete. For example, to change the caption of a window at any time, you can use the **WM_SETTEXT** message. The syntax to use would be:

```
SendMessage(WM_SETTEXT, wParam, lParam);
```

Obviously you would need to provide the text for the caption you are trying to change. This string is carried by the *lParam* argument as a null-terminated string. For this message, the *wParam* is ignored.

Practical Learning: Sending Messages

1. To process a message using the SendMessage() function, change the OnRButtonUp() event as follows:

```
void CMainFrame::OnRButtonUp(UINT nFlags, CPoint point)
{
	const char *Msg = "This message was sent";
	SendMessage(WM_SETTEXT, 0, (LPARAM)(LPCTSTR)Msg);
}
```

2. Test the application and return to MSVC

Chapter 5: The Document/View Architecture

5.1 Overview of the Document/View Architecture

5.1.1 Introduction

The Document/View architecture is the foundation used to create applications based on the Microsoft Foundation Classes library. It allows you to make distinct the different parts that compose a computer program including what the user sees as part of your application and the document a user would work on. This is done through a combination of separate classes that work as an ensemble.

The parts that compose the Document/View architecture are a frame, one or more documents, and the view. Put together, these entities make up a usable application.

5.1.2 The View

A view is the platform the user is working on to do his or her job. For example, while performing word processing, the user works on a series of words that compose the text. If a user is performing calculations on a spreadsheet application, the interface the user is viewing is made of small boxes called cells. Another user may be in front of a graphic document while drawing lines and other geometric figures. The thing the user is starring at and performing changes is called a view. The view also allows the user to print a document.

To let the user do anything on an application, you must provide a view, which is an object based on the CView class. You can either directly use one of the classes derived from CView or you can derive your own custom class from CView or one of its child classes.

5.1.3 The Document

A document is similar to a bucket. It can be used to hold or carry water and that water can be retrieved when needed. For a computer application, a document holds the user's data. For example, after working on a text processor, the user may want to save the file. Such an action creates a document and this document must reside somewhere. In the same way, to use an existing file, the user must locate it, open it, and make it available to the application. These two jobs and many others are handled behind the scenes as a document.

To create the document part of this architecture, you must derive an object from the CDocument class.

5.1.4 The Frame

As its name suggests, a frame is a combination of the building blocks, the structure (in the English sense), and the borders of an item. A frame gives "physical" presence to a window. A frame defines the location of an object with regards to the Windows desktop. A frame provides borders to a window, borders that the user can grab to move, size, and resize the object. The frame is also a type of desk that holds the tools needed on an application.

An application cannot exist without a frame. As we saw in previous lessons, to provide a frame to an application, you can derive a class from **CFrameWnd**.

5.1.5 The Document/View Approach

To create an application, you obviously should start by providing a frame. This can be taken care of by deriving a class from CFrameWnd. Here is an example:

```
class CMainFrame : public CFrameWnd
{
	DECLARE_DYNCREATE(CMainFrame)

	DECLARE_MESSAGE_MAP()
};
```

To give "physical" presence to the frame of an application, you can declare an OnCreate() method. Here is an example:

```
class CMainFrame : public CFrameWnd
{
	DECLARE_DYNCREATE(CMainFrame)

	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	DECLARE_MESSAGE_MAP()
};
```

The easiest way you can implement this method is to call the parent class, CFrameWnd, to create the window. As we have seen in the past, if this method returns 0, the frame has been created. It returns -1, this indicates that the window has been destroyed. Therefore, you can create a frame as follows:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	// Call the base class to create the window
	if( CFrameWnd::OnCreate(lpCreateStruct) == 0)
		return 0;

	// else is implied
	return -1;
}
```

To allow users to interact with your application, you should provide a document. To do this, you can derive a class from CDocument so you can take advantage of this class. If you do not plan to do anything with the document, you can just make it an empty class. Here is an example:

```
class CExerciseDoc : public CDocument
{
	DECLARE_DYNCREATE(CExerciseDoc)

	DECLARE_MESSAGE_MAP()
};
```

Besides the few things we have learned so far, your next big decision may consist on the type of application you want to create. This is provided as a view. The most fundamental class of the view implementations in the MFC is CView. Because CView is an abstract class, you cannot directly use it in your application. You have two main alternatives. You can derive your own class based on CView (the CView class itself is based on CWnd) or you can use one of the many view classes that the MFC provides. The classes that are readily available to you are:

Class	Description
CEditView	Used for a basic text editing application
CRichEditView	Allows creating rich documents that perform text and paragraph formatting
CScrollView	Provides the ability to scroll a view horizontally and vertically
CListVie w	Provides a view that can display a list of items
CTreeView	Allows displaying a list of items arranged as a tree
CFormView	Used to create a view that resembles a dialog box but provides the document/view features
CDaoRecordView	Provides a view that res embles a dialog box but used for database controls
CCtrlView	Provides parental guidance to the CEditView, CListView, CTreeView, and CRichEditView

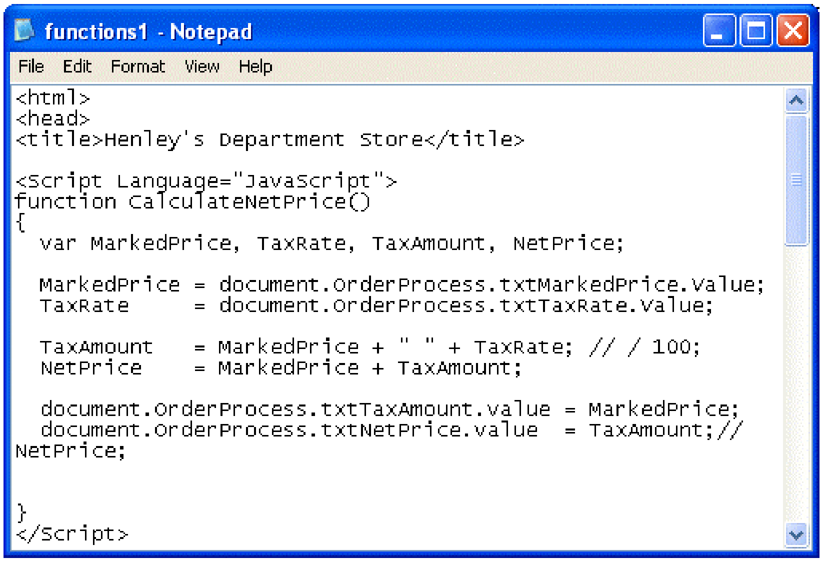
As we move on, we will study these classes as needed.

Once you have a frame, a document, and a view, you can create an application, which, as we have learned so far, is done by deriving a class from CWinApp and overriding the InitInstance() method. In the InitInstance() implementation, you must let the compiler know how a document is created in your application. To do this, you must provide a sample document, called a template, that defines the parts that constitute a document for your particular type of application. This is done using a pointer variable declared from CDocTemplate or one of its derived classes.

5.2 The Single Document Interface (SDI)

5.2.1 Overview

The expression Single Document Interface or SDI refers to a document that can present only one view to the user. This means that the application cannot display more than one document at a time. If the user wants to view another type of document of the current application, he or she must another instance of the application. Notepad and WordPad are examples of SDI applications:



```
<html>
<head>
<title>Henley's Department Store</title>

<Script Language="JavaScript">
function CalculateNetPrice()
{
  var MarkedPrice, TaxRate, TaxAmount, NetPrice;

  MarkedPrice = document.OrderProcess.txtMarkedPrice.Value;
  TaxRate     = document.OrderProcess.txtTaxRate.Value;

  TaxAmount   = MarkedPrice + " " + TaxRate; // / 100;
  NetPrice    = MarkedPrice + TaxAmount;

  document.OrderProcess.txtTaxAmount.value = MarkedPrice;
  document.OrderProcess.txtNetPrice.value  = TaxAmount;//
NetPrice;


}
</Script>
```

Figure 50: Notepad as an SDI

Notepad can be used to open only one document such as an HTML file, to view another file, the user would either replace the current document or launch another copy of Notepad.

To create an SDI, Microsoft Visual C++ provides the MFC wizard which provides all the basic functionality that an application needs, including the resources and classes.

5.2.2 Creating a Single Document Interface

As mentioned earlier, after creating a frame, a document, and a view, you can create an application by deriving a class from CWinApp and overriding the virtual **InitInstance()** member function. In **InitInstance(),** you must provide a template for your type of application. This is done using a **CDocTemplate** type of object.

To create an SDI, the **CDocTemplate** class provides the **CSingleDocTemplate** class used to create an application that provides only one view. Therefore, you can declare a pointer variable to **CSingleDocTemplate**. Using this pointer and the new operator, use the **CSingleDocTemplate** constructor to provide the template. The syntax of the **CSingleDocTemplate** constructor is:

```
CSingleDocTemplate(UINT nIDResource,
                   CRuntimeClass* pDocClass,
                   CRuntimeClass* pFrameClass,
                   CRuntimeClass* pViewClass);
```

The **CSingleDocTemplate** constructor needs the common identifier for the resources of your application. We saw in Lesson 3 that this can be done by using IDR_MAINFRAME as the common name of most or all main resources of an application. This is provided as the *nIDResource* argument.

The second argument, pDocClass, is the name of the class you derived from **CDocument**, as mentioned earlier.

The third argument, *pFrameClass*, if the frame class you derived from either *CFrameWnd* or one of its children.

The pViewClass argument can be an MFC CView-derived class. It can also be a class you created based on CView.

Each of these arguments must be passed as a pointer to CRuntimeClass. This can be taken care of by using the **RUNTIME_CLASS** macro. Its syntax is:

```
RUNTIME_CLASS(ClassName);
```

Each one of the classes you want to use must be provided as the *ClassName* argument. The **RUNTIME_CLASS** macro in turn returns a pointer to **CRuntimeClass**. To effectively use the **RUNTIME_CLASS** macro, you should make sure that the (each) class is created and implemented using the **DECLARE_DYNAMIC**, the **DECLARE_DYNCREATE**, or the **DECLARE_SERIAL** macros.

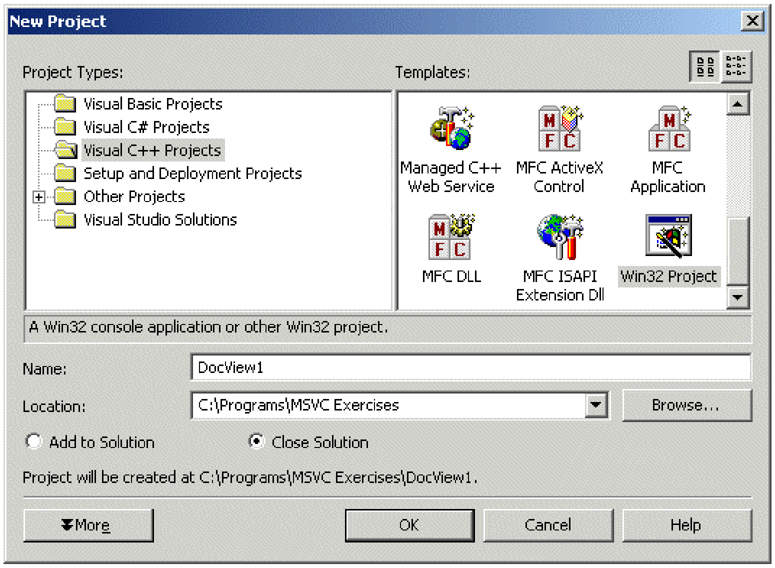
To actually create the application so it can be displayed to the user, the CWinApp class is equipped with the AddDocTemplate() method. Therefore, After creating a template, pass the CSingleDocTemplate pointer to the CWinApp::AddDocTemplate() method. Its syntax is:

```
void AddDocTemplate(CDocTemplate *pTemplate);
```

Everything else is subject to how you want your application to provide a useful experience to the user.

Practical Learning: Creating a Document/View Application

1. If necessary, start Microsoft Visual Studio or Visual C++
2. Create a new empty Win32 application named DocView1



A Win32 console application or other Win32 project.

Name: DocView1

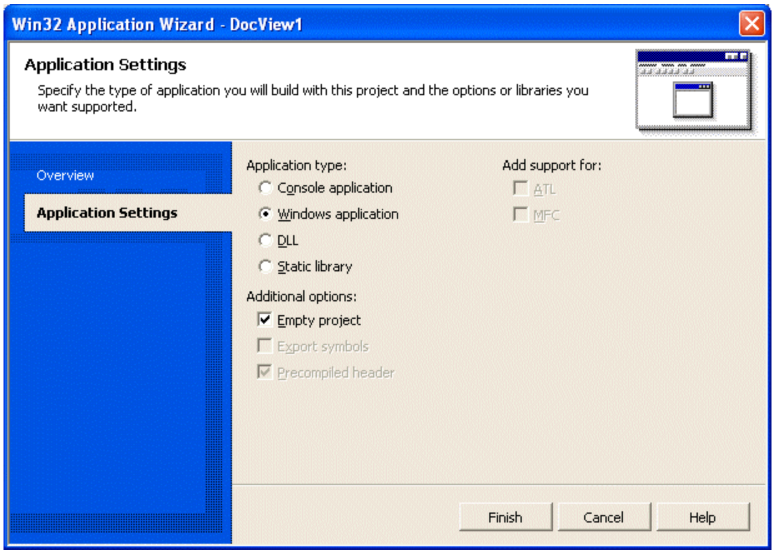
Location: C:\Programs\MSVC Exercises

○ Add to Solution ● Close Solution

Project will be created at C:\Programs\MSVC Exercises\DocView1.

Figure 51: New Project - DocView1

3. Click OK
4. Specify that you want to create an empty document:



Win32 Application Wizard - DocView1

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:

○ Console application
● Windows application
○ DLL
○ Static library

Additional options:

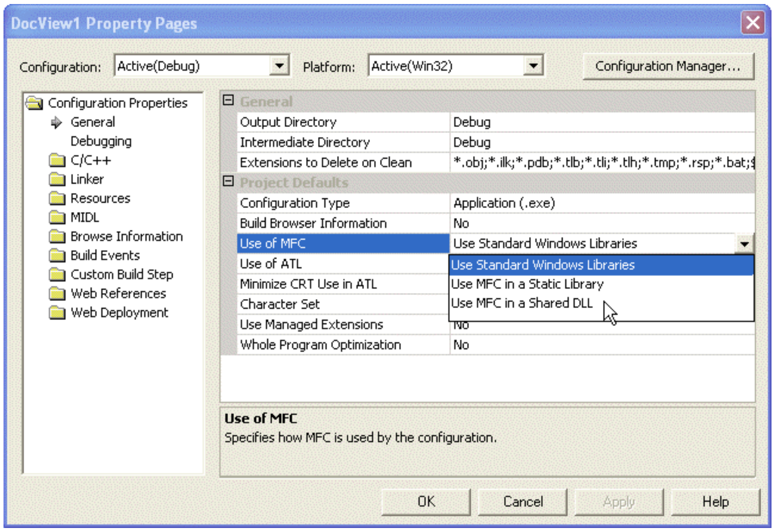
☑ Empty project
□ Export symbols
☑ Precompiled header

Add support for:

□ ATL
□ MFC

Figure 52: Win32 Application Wizard - DocView1

5. Click Finish



DocView1 Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

General	
Output Directory	Debug
Intermediate Directory	Debug
Extensions to Delete on Clean	*.obj;*.ilk;*.pdb;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.bat;$
Project Defaults	
Configuration Type	Application (.exe)
Build Browser Information	No
Use of MFC	Use Standard Windows Libraries
Use of ATL	
Minimize CRT Use in ATL	
Character Set	
Use Managed Extensions	No
Whole Program Optimization	No

Use of MFC
Specifies how MFC is used by the configuration.

Figure 53: DocView1 Property Pages

6. Specify that you will Use MFC In A Shared DLL

7. To create an icon, display the Add Resource dialog box and click Icon

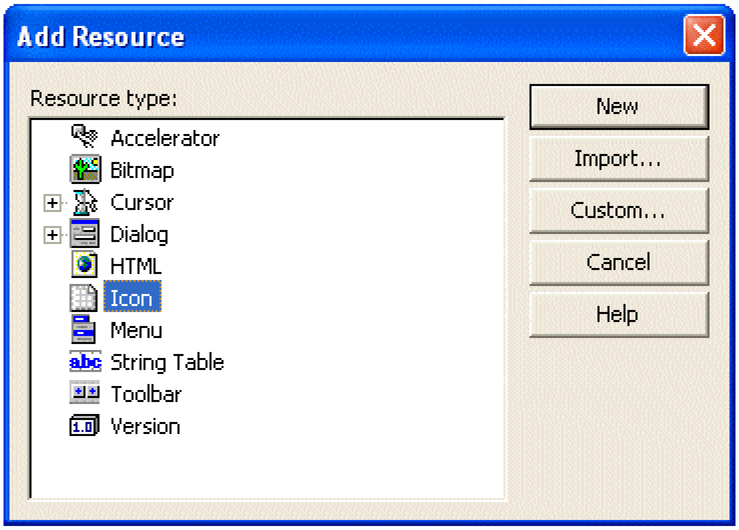


Figure 54: Adding an Icon Resource

8. Click New
9. Design the 32x32 icon as follows:

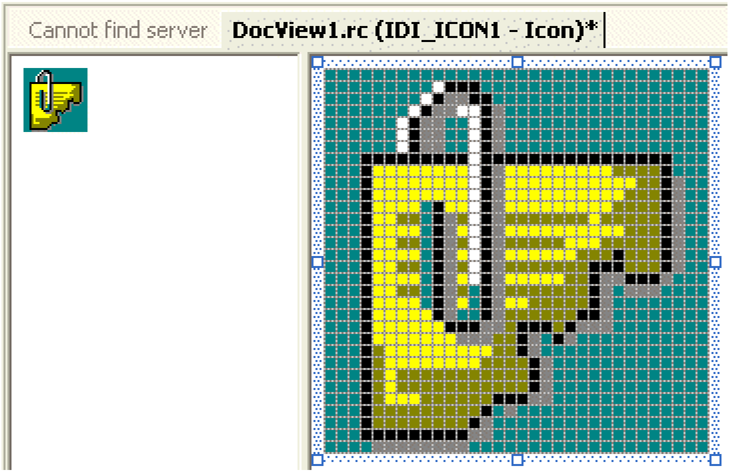


Figure 55: Icon Design - DocView1

10. Add a 16x16 icon and design it as follows:

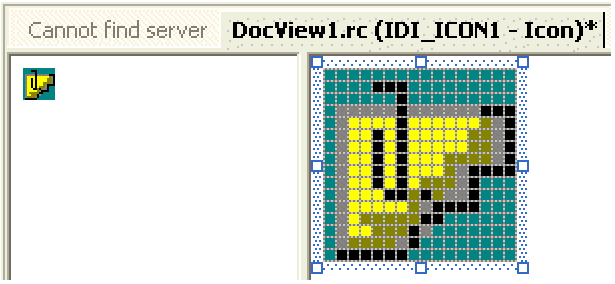
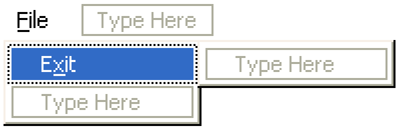


Figure 56: Icon Design - DocView2

11. Using the Properties window, change the ID of the icon to **IDR_MAINFRAME**
12. Display the Add Resource dialog box and double-click Menu
13. Change the ID of the menu from IDR_MENU1 to **IDR_MAINFRAME**

14. Create the menu items as follows:



15. (If you are using MSVC 5 or 6, first close both the menu and the icon windows. You will be asked to save the resource script and accept to save it. Save it as DocView. Then, on the main menu, click Project -> Add To Project -> Files... Select the DocView.rc file and click OK.

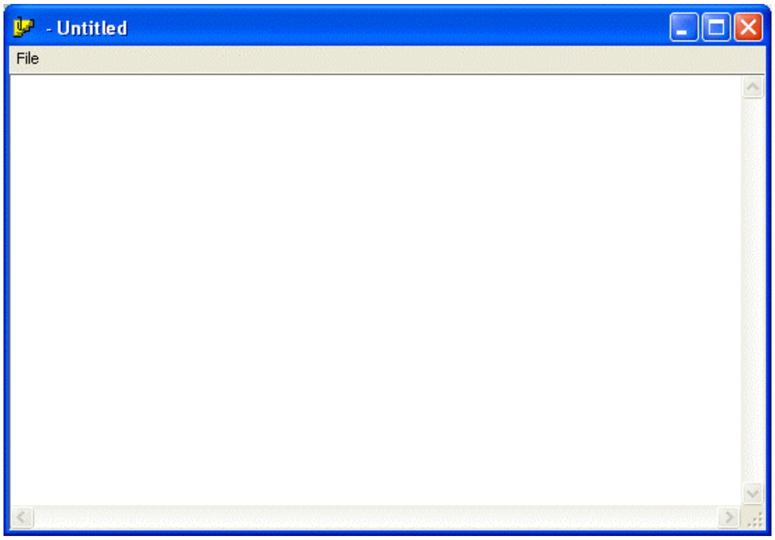
16. Add a new header file named Exercise and, in it, create the necessary classes as follows:

```
#include <afxext.h>     // For CEditView
#include "resource.h"

class CExerciseApp : public CWinApp
{
	BOOL InitInstance();

	DECLARE_MESSAGE_MAP()
};

class CMainFrame : public CFrameWnd
{
	DECLARE_DYNCREATE(CMainFrame)

	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	DECLARE_MESSAGE_MAP()
};

class CExerciseDoc : public CDocument
{
	DECLARE_DYNCREATE(CExerciseDoc)

	DECLARE_MESSAGE_MAP()
};

class CExerciseView : public CEditView
{
	DECLARE_DYNCREATE(CExerciseView)

	DECLARE_MESSAGE_MAP()
};
```

17. Add a new source file named Exercise and, in it, implement the classes as follows:

```
#include <afxwin.h>
#include "Exercise.h"

BEGIN_MESSAGE_MAP(CExerciseApp, CWinApp)

END_MESSAGE_MAP()

BOOL CExerciseApp::InitInstance()
{
	CSingleDocTemplate* pDocTemplate;
	pDocTemplate = new CSingleDocTemplate(
			IDR_MAINFRAME,
                RUNTIME_CLASS(CExerciseDoc),
                RUNTIME_CLASS(CMainFrame),
                RUNTIME_CLASS(CExerciseView));
        AddDocTemplate(pDocTemplate);

        CCommandLineInfo cmdInfo;

        // Dispatch commands specified on the command line
        if (!ProcessShellCommand(cmdInfo))
                return FALSE;

        // The one and only window has been initialized, so show and update it.
        m_pMainWnd->ShowWindow(SW_SHOW);
        m_pMainWnd->UpdateWindow();

        return TRUE;
}

// -- Frame Map -- //
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        ON_WM_CREATE()
END_MESSAGE_MAP()

// -- Document Map -- //
IMPLEMENT_DYNCREATE(CExerciseDoc, CDocument)
BEGIN_MESSAGE_MAP(CExerciseDoc, CDocument)

END_MESSAGE_MAP()

// -- View Map --
IMPLEMENT_DYNCREATE(CExerciseView, CEditView)
BEGIN_MESSAGE_MAP(CExerciseView, CEditView)

END_MESSAGE_MAP()

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        // Call the base class to create the window
        if( CFrameWnd::OnCreate(lpCreateStruct) == 0)
                return 0;

        return -1;
}

CExerciseApp theApp;
```

18. Test the application



19. Close it using its System Close button and return to MSVC

5.3 SDI Improvements

5.3.1 SDI Improvements: The Application

To make your programming experience a little faster and efficient, the framework provides many other features for each class used in an application.

The Application: The programs we will create in this book use classes of the Microsoft Foundation Classes (MFC) library. MFC classes are created is various libraries called DLLs. In order to use MFC objects in your application as opposed to non-MFC objects, you must let the compiler know. This is done by specifying that you want to Use MFC In A Shared DLL, as we have done so far. Additionally, if you want your windows to have a 3-D appearance, call the Enable3dControls() method. If you do not want the 3-D appearance, call the Enable3dControlsStatic() method. The best way to deal with this is to ask the compiler to check if you had allowed using MFC in a shared DLL or not, and then tell the compiler which of these two functions to execute. This is done using a #ifdef preprocessor in your InitInstance() method. Here is an example:

```
#include <afxwin.h>

class CSimpleFrame : public CFrameWnd
{
public:
        CSimpleFrame()
        {
                // Create the window's frame
                Create(NULL, "Windows Application");
        }
};

class CSimpleApp : public CWinApp
{
public:
        BOOL InitInstance();
};

BOOL CSimpleApp::InitInstance()
{
#ifdef _AFXDLL
   Enable3dControls( );
#else
   Enable3dControlsStatic();
#endif

        CSimpleFrame *Tester = new CSimpleFrame ();
        m_pMainWnd = Tester;

        Tester->ShowWindow(SW_SHOW);
        Tester->UpdateWindow();

        return TRUE;
}

CSimpleApp theApp;
```

To provide your application the ability to create a new document, the CWinApp class provides the OnFileNew() method. Its syntax is:

```
afx_msg void OnFileNew();
```

To use this method, create a menu item identified as ID_FILE_NEW. You should also create a prompt for it so the menu item can be added to the string table. This menu item is traditionally and obviously added to the File menu. After creating this menu item, in the message table of the application's source, invoke the CWinApp::OnFileNew() method using the ON_COMMAND() macro. This can be done as follows:

```
BEGIN_MESSAGE_MAP(CExoApp, CWinApp)
   ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
END_MESSAGE_MAP()
```

CWinApp also provides an application the ability to easily open a document. This is done using the OnFileOpen() method. In the same way, it can help with printing a document. Here is a summary:

Menu ID	CWinApp Message Map
ID_FILE_NEW	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ID_FILE_OPEN	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
ID_FILE_PRINT_SETUP	ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup

One of the last minute assignment you may need to perform when the user is closing an application is to check if the displayed document is "dirty", that is, if the document has been changed since it was last accessed. To help with this, simply create a menu item identified as ID_APP_EXIT and set a caption accordingly, such as the Exit menu we created in the previous Practical Learning section. It is always helpful to add a prompt to a menu item.

These command messages are implemented in the CWinApp class and can be helpful for your application. If their behavior does not fulfill your goal, you can write your own intended implementation of these menu items.

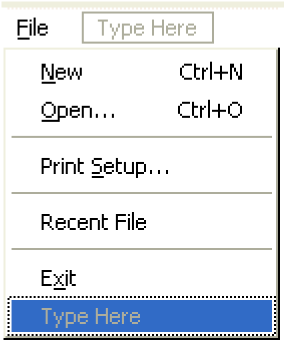
When using an application over and over, sometimes a user may want to open the last accessed document or at least see a list of the last documents opened on an application. To provide this functionality, create a menu item called ID_FILE_MRU_FILE1 and set its prompt to a string such as Recent File. This menu item is usually added to the File menu above the Exit or quit. The actual list of recent files is stored in an INI file that accompanies your application. To make this list available, you must call the LoadStdProfileSettings() method of the CWinApp class in your InitInstance() method. The syntax of this method is:

```
void LoadStdProfileSettings(UINT nMaxMRU = _AFX_MRU_COUNT);
```

By default, this allows the list to display up to four names of documents. This method takes one argument as the number of document names to be displayed in the list. If you do not want the default of 4, specify the nMaxMRU value to your liking.

Practical Learning: Improving the Application

1. To provide new functionality to the application, in the Resource View, change the IDentifier of the Exit menu item to ID_APP_EXIT and set its Prompt to Quit the application
2. Add the following menu item under File:



Caption	ID	Prompt
&New\tCtrl+N	ID_FILE_NEW	Create a new document
&Open...\tCtrl+O	ID_FILE_OPEN	Open an existing document
-		
P&rint Setup...	ID_FILE_PRINT_SETUP	Change the printer and printing options
-		
Recent file	ID_FILE_MRU_FILE1	Open this file
-		
E&xit	ID_APP_EXIT	Quit the application; prompt the save document

3. To allow the application to treat documents, change the InitInstance() implementation as follows:

```
BEGIN_MESSAGE_MAP(CExerciseApp, CWinApp)
	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
	ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

```
BOOL CExerciseApp::InitInstance()
{
#ifdef _AFXDLL
   Enable3dControls( );
#else
   Enable3dControlsStatic();
#endif
     LoadStdProfileSettings(6);

     CSingleDocTemplate* pDocTemplate;
     pDocTemplate = new CSingleDocTemplate(
               IDR_MAINFRAME,
               RUNTIME_CLASS(CExerciseDoc),
               RUNTIME_CLASS(CMainFrame),
               RUNTIME_CLASS(CExerciseView));
     AddDocTemplate(pDocTemplate);

     CCommandLineInfo cmdInfo;

     // Dispatch commands specified on the command line
     if (!ProcessShellCommand(cmdInfo))
               return FALSE;

     m_pMainWnd->ShowWindow(m_nCmdShow);
     m_pMainWnd->UpdateWindow();

     return TRUE;
}
```

4. Test the application and click the various menu items

5.3.2 SDI Improvements: The Document

The document is actually the object that holds the contents of a file. Based on this role, it is its responsibility to validate the creation of a new file or to store a file that is being saved. To perform these tasks and others, the CDocument class provides various methods you can conveniently add to your application or add and customize their behavior.

Earlier, we saw that, to give the user a convenient means of creating a new document, you can add an ID_FILE_NEW menu identifier and connect it to your application class in the InitInstance() method. Clicking this menu item only allows to initiate the action, the document that is the base of file contents must be aware and validate this action. When a user decides to create a new document or when the application opens and is about to create a new document, you may want to make sure that there is no existing document or you may want to delete the existing one. To take care of this, the CDocument class provides the virtual OnNewDocument() method. Its syntax is:

```
virtual BOOL OnNewDocument();
```

When a new file is about to be created, this method is called to initiate it. If everything goes fine and the file is created, the OnNewDocument() method returns TRUE. If the file cannot be initialized, this method returns FALSE or 0. This method, which really behaves like an event, does not create a new file. It is launched when a new file is going to be created and allows you to make it happen or to prevent the creation of a new file.

When a new file has been created, it displays as empty. Such a document is referred to as "clean".

We also saw earlier that, to help the user open an existing document, you can create a menu item identified as ID_FILE_OPEN and associate it with the CWinApp::OnFileOpen() method in your InitInstance() method. This time also, the menu item only provides a convenient way to perform the action. It makes the document available to the application and not to the document. Once a user has initiated the action of opening an existing file, you may want to check that the document not only exists but also can be opened and make its contents available to the user. This job can be handled by the OnOpenDocument() virtual method of the CDocument class. Its syntax is:

```
virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
```

This method usually results from the user clicking File -> Open... on the main menu, communicating a desire to open a file. When this action is initiated, the OnOpenDocument() method retrieves the path of the file as the lpszPathName argument. If the path is valid, that is, if the file exists, you can then check it or perform a last minute task before the file is opened. For example you can use this method to decide how the contents of the file will be displayed or dealt with by the document. You can also use to prevent the user from opening any file or to prevent the user from opening any file at all. You can also use this method to allow the user to open a type of file that your CDocument-derived class would not expect.

If the user has opened an existing file but has not (yet) changed anything in the document, the file is also called "clean". As we will learn eventually, some files can be changed and some do not allow modification. If a document allows the user to change it, he or she can manipulate it as necessary, including adding, deleting, or moving items. Once a user has changed anything on the document, the file is referred to as "dirty". You may want to keep track of such change(s) so you would know eventually if the document needs to be saved. To help you with this, the CDocument class provides the SetModifiedFlag() method. Its syntax is:

```
void SetModifiedFlag(BOOL bModified = TRUE);
```

To mark a document as clean or dirty, call the SetModifiedFlag() method. If you pass the bModified argument as TRUE, the document has been changed. Since the TRUE constant is its default value, you can also call the method simply as SetModifiedFlag(). To specify that the document is clean, pass the argument as FALSE. You can call this method whenever you judge necessary. For example, if the user saves the document while working on it but makes another change, you can mark it clean when it has just been saved and mark it dirty if the user changes anything again. At any time, you can check whether the document is dirty or clean using the CDocument::IsModified() method. Its syntax is:

```
BOOL IsModified();
```

This method simply checks the document to find out if it has been modified since the last time it was accessed. If the document has been modified, this method would return TRUE. If the document is clean, it returns FALSE.

Another action the user can perform on a document is to send it electronically to an email recipient. To allow the user to send the current document as an email attachment, first an email client (such as MS Outlook) must be installed on the user's computer. Therefore, add a menu item IDentified as ID_FILE_SEND_MAIL. Then, in the message table of your document implementation, add the following two macros:

```
BEGIN_MESSAGE_MAP(CTestDoc, CDocument)
    ON_COMMAND(ID_FILE_SEND_MAIL, OnFileSendMail)
    ON_UPDATE_COMMAND_UI(ID_FILE_SEND_MAIL, OnUpdateFileSendMail)
END_MESSAGE_MAP
```

Once a user has finished using a document, he or she would need to close it, which is done by either clicking the System Close button or using the main menu (File -> Exit). If the user clicks the System Close button , the application would need to close the frame. At this time, the document would call the OnCloseDocument() method. Its syntax is:

```
virtual void OnCloseDocument();
```

You can use this method (which really behave as an event) to decide what to do before a frame is closed.

When the user decides to close an application, the document class checks the file to know whether the file is "dirty". If the file is dirty, you may want to ask the user to save or not save the document. As we saw earlier with the ID_APP_EXIT pre-configured menu, the framework can check this setting for you. Also, while using a file, the user may want to save it. If the user is working on a document that was opened from a drive, the document may be saved immediately behind the scenes. If the user is working on a brand new document and decides to save it, you may want to check first if this is possible and what needs to be done in order to save the document.

To help the user save a document, you can create a menu item. Using the Document/View architecture, add a menu item with the identifier ID_FILE_SAVE in the IDR_MAINFRAME common resource name. It is that simple. This menu is usually positioned under File with a caption of &Save.

If the user wants to save a document with a different name and/or a different location, which is usually done by clicking File -> Save As... from the main menu, create a menu item with the ID_FILE_SAVE_AS identifier. This menu item is usually placed under Save in the File main menu. If the user is working on a new document (that has not been saved previously), or if the user working on a document that is marked as Read-Only, or if the user decides to close the application after working on a new document, and if the user decides to save the document, the action would initiate the File -> Save As action.

When the user has decided to save a document, if the document is dirty, a message box would display to the user for a decision. If the user decides to save the document, the CDocument class provides the OnSaveDocument() method to validate the action. The syntax of this method is:

```
virtual BOOL OnSaveDocument(LPCTSTR lpszPathName);
```

To save a document, the user must specify where the document would reside. This is communicated as the lpszPathName argument). In reality, as its name suggests, this method is not used to save a document. It allows you to validate the action or desire to save a document. For example, if the user clicks File -> Save on the main menu or if the user is attempting to close a dirty document, you can use this method to check what is going or to deny saving the file.

Serialization is the ability to store data on a drive. Therefore, to actually save a file, the CObject class provides the Serialize() method to its children. To store data of the file, its information is held by a class called CArchive. When saving a file, a CArchive object is passed to the Serialize method as reference, which modifies its value.

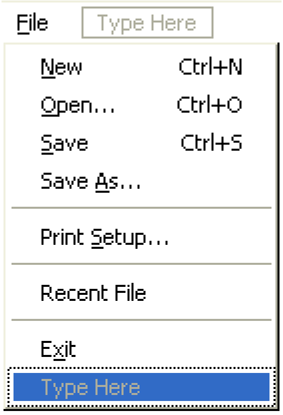
CArchive is used to either save the current document or open an existing one. To take care of each, it uses two methods (ReadObject() and WriteObject()). These methods are actually implemented using the extraction operators (>> and <<). Whenever you need to perform serialization in an application, add a method called Serialize() to your document class and pass it a CArchive object reference. The syntax of this method is:

```
virtual void Serialize(CArchive& ar);
```

The implementation of this method may depend on the document.

Practical Learning: Improving the Document

1. In the Resource View, add the follo wing menu items under the File menu:



Caption	ID	Prompt
&New\tCtrl+N	ID_FILE_NEW	Create a new document
&Open...\tCtrl+O	ID_FILE_OPEN	Open an existing document
&Save\tCtrl+S	**ID_FILE_SAVE**	**Save the current document**
Save &As...	**ID_FILE_SAVE_AS**	**Save the current document with a new name or location**
-		
P&rint Setup...	ID_FILE_PRINT_SETUP	Change the printer and printing options
-		
Recent file	ID_FILE_MRU_FILE1	Open this file
-		
E&xit	ID_APP_EXIT	Quit the application; prompt the save document

2. To use a CDocument method, in the header file, add the OnNewDocument() function to the document as follows:

```
class CExerciseDoc : public CDocument
{
	DECLARE_DYNCREATE(CExerciseDoc)

	virtual BOOL OnNewDocument();
	DECLARE_MESSAGE_MAP()
};
```

3. In the source file, implement the method as follows:

```
…

BOOL CExerciseDoc::OnNewDocument()
{
    return CDocument::OnNewDocument();
}

CExerciseApp theApp;
```

4. Test the application. Change the content of the empty file and save it
5. Close the application and return to MSVC.

5.3.3 SDI Improvements: The Frame

The **CWnd::OnCreate()** method is used to create a window and it is usually meant to do this using its default configured features. Therefore, anything you want to display on the frame when the application displays, you can do so when creating the application. Therefore, the frame is typically used to create and display the toolbar(s), dialog bar(s), and status bar.

After the frame has been created, if you want to modified something on it, you can do so after it has been created but before it is displayed to the user. To do this, you can use the PreCreateWindow() method of the CWnd class. Its syntax is:

```
virtual void PreCreateWindow(CREATESTRUCT& cs);
```

This method takes a reference to CREATESTRUCT class, modifies and returns it with the new characteristics. For example, you can use this method to remove the Minimize and the Maximize system buttons on the title bar as follows:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        cs.style &= ~(WS_MAXIMIZEBOX | WS_MINIMIZEBOX);

        return CFrameWnd::PreCreateWindow(cs);
}
```

Practical Learning: Improving the Frame

1. To use the CWnd::PreCreateWindow() virtual method, in the header file, declare it as follows:

```
class CMainFrame : public CFrameWnd
{
    DECLARE_DYNCREATE(CMainFrame)

    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    DECLARE_MESSAGE_MAP()
};
```

2. To modify the dimensions of the window at start up, implement the method as follows:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	// cs.style &= ~(WS_MAXIMIZEBOX | WS_MINIMIZEBOX);
	cs.cx = 450; // Change the width
	cs.cy = 350; // Change the height

	return CFrameWnd::PreCreateWindow(cs);
}
```

3. Test the application and return to MSVC

5.3.4 SDI Improvements: The View

As mentioned already, the object that holds a file's data is the document which is an object of CDocument type. To make this document available to the user to view it, in your view derived class, you should declare a pointer to the document class. The syntax used is:

```
CDocument* GetDocument() const;
```

When implementing this method, simply ask the application to return a pointer to the document class used in your application. This can be done by casting the global m_pDocument variable to your document class:

```
CExerciseDoc* CExe rciseView::GetDocument()
{
	return (CExerciseDoc*)m_pDocument;
}
```

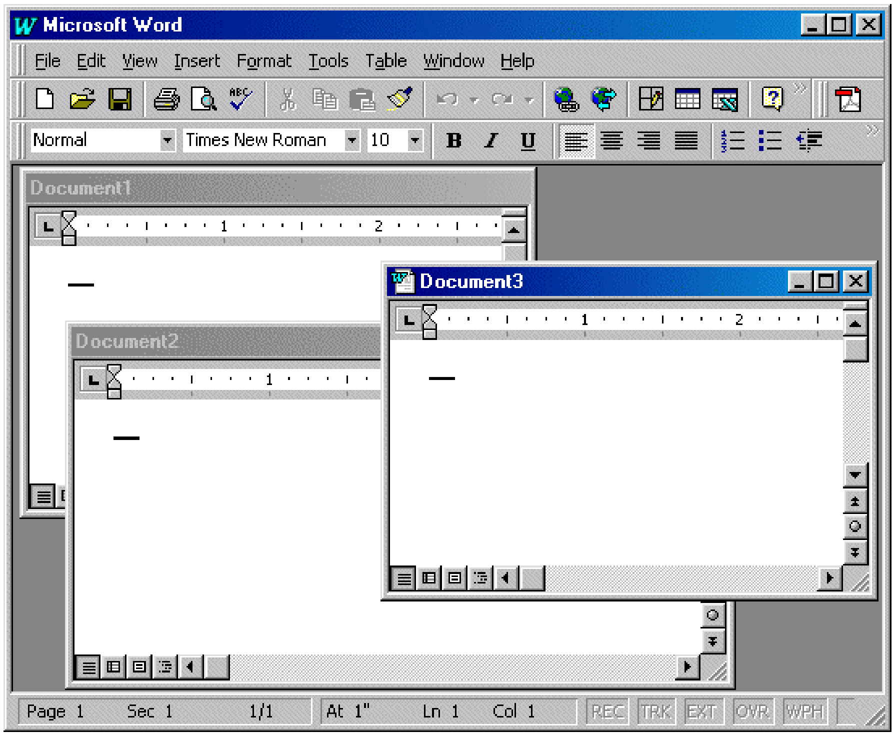
You can use this GetDocument() method in your view class to access the document. For example, the view class can access the contents of the class

If you want to allow the user to print a document, you can add a menu item identified as ID_FILE_PRINT. Then, in the message table of the view class, use the ON_COMMAND macro to associate it to the view parent class of your derived class. You can use this same approach to allow the user to preview document. The identifier for this action is called ID_FILE_PRINT_PREVIEW.

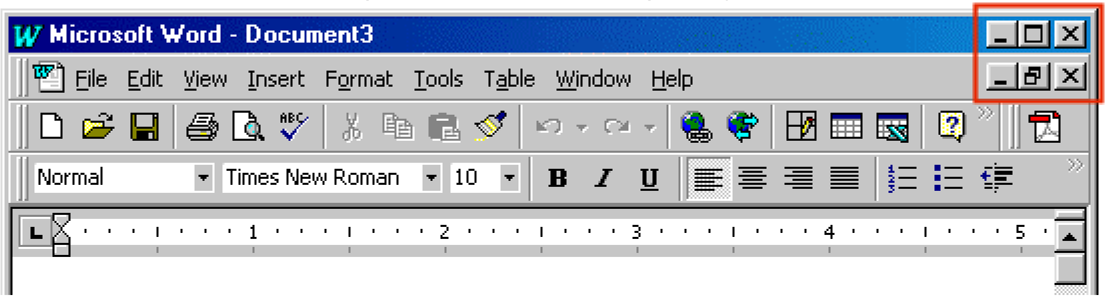
5.4 The Multiple Document Interface (MDI)

5.4.1 Overview

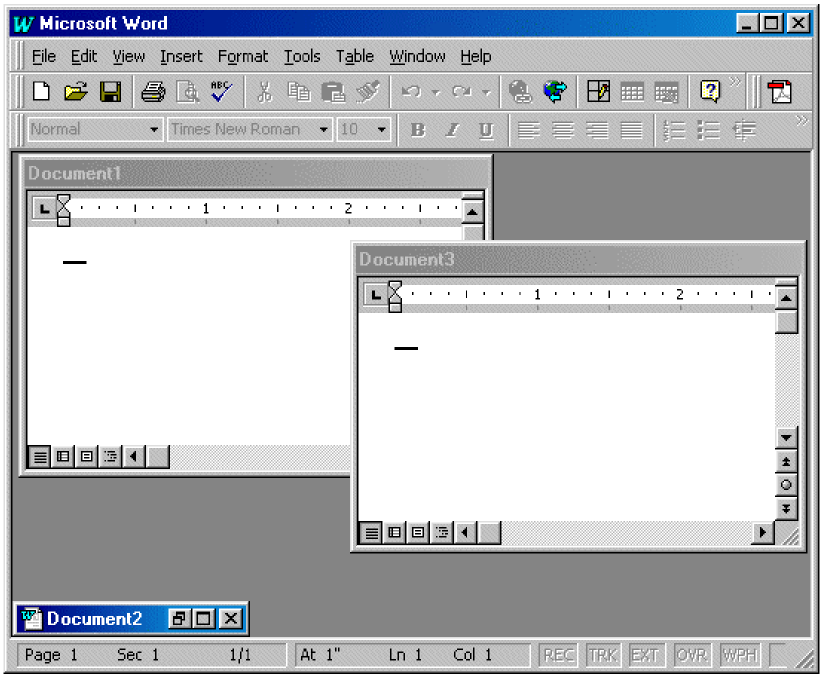
An application is referred to as Multiple Document Interface, or MDI, if the user can open more than one document in the application without closing it. to provide this functionality, the application provides a parent frame that acts as the main frame of the computer program. Inside of this frame, the application allows creating views that each uses its own frame, making it distinct from the other. Here is Microsoft Word 97 displaying as a classic MDI application:


Microsoft Word
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
10
Document1
Document2
Document3
Page 1 Sec 1 1/1 At 1" Ln 1 Col 1 REC TRK EXT OVR WPH

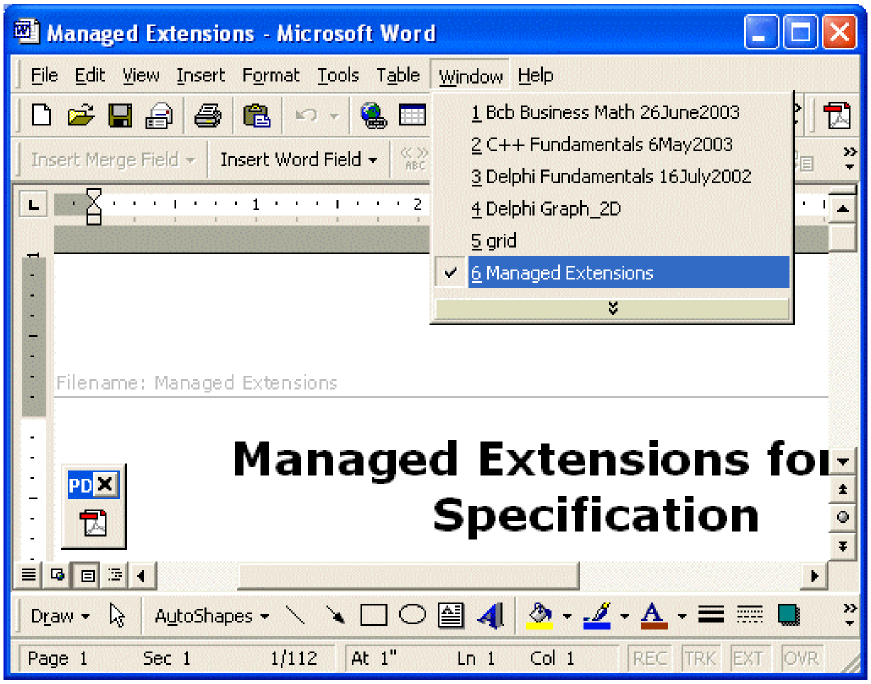
When using an MDI application, a user can create a document, open another while the first is still displaying, and even create new documents or open additional ones. The documents are stacked in a Z-order axis of a 3-D coordinate system. There are two ways to display the documents of an MDI. If one document is maximized, all documents are maximized. In this case, the main menu of the application would display the system buttons on its right, which creates two ranges of system buttons:


Microsoft Word - Document3
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
10

If no document is maximized, each document can assume one of two states: minimized or restored. While the child documents are confined to the borders of the main frame, if a document is or gets minimized, it would display its button inside the main frame. If a document is not minimized, it would display inside the main frame either with its original size or the size the user had given it:


Microsoft Word
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
10
Document1
Document3
Document2
Page 1 Sec 1 1/1 At 1" Ln 1 Col 1 REC TRK EXT OVR WPH

There are two main ways the user can access the different documents of an MDI. If they are maximized (remember that if the user maximizes one document, all the others get maximized also), the menu of the main frame, which we always call the main menu in this book, has an item called Window. When the Window menu item is accessed, it would display a list of the currently opened documents. The user can then select from that list:

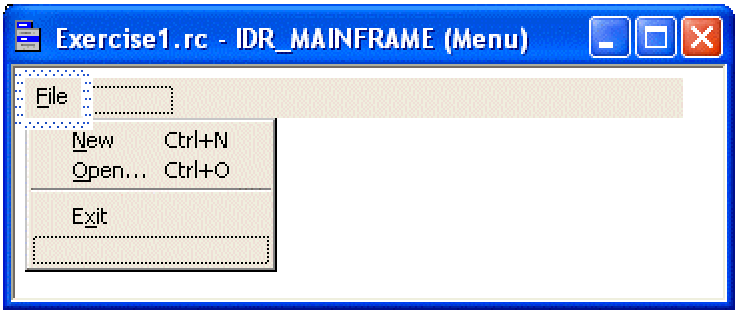

Managed Extensions - Microsoft Word
File Edit View Insert Format Tools Table Window Help
1 Bcb Business Math 26June2003
2 C++ Fundamentals 6May2003
3 Delphi Fundamentals 16July2002
4 Delphi Graph_2D
5 grid
6 Managed Extensions
Insert Merge Field
Insert Word Field
Filename: Managed Extensions
Managed Extensions for
Specification
Draw
AutoShapes
Page 1 Sec 1 1/112 At 1" Ln 1 Col 1 REC TRK EXT OVR

The separation of the parent and the child frames allows the user to close one child document or all documents. This would still leave the main frame of the application but the user cannot use it as a document. Still, the main frame allows the user to either create a new document or open an existing one.

To manage the differentiation between the parent and its children, the application uses two different menus: one for the main frame and one for a (or each) child. Both menus use the same menu bar, meaning that the application cannot display two frame menus at one time. It displays either the parent menu or a (the) child menu. When at least one document is displaying, the menu applies to the document. If no document is displaying, the main frame is equipped with a simplified menu with limited but appropriate operations. The user can only create a new document, only open an existing document, or simply close the application.

5.4.2 Creating a Multiple Document Interface

We have mentioned that an MDI is an application made of a parent frame and at least one child. Therefore, to create an MDI, because it requires a frame of its own, you should first create a series of resource objects that share a common name as IDR_MAINFRAME. A basic application should have an icon and a menu. The icon can be designed any way you like and you are recommended to create one made of a 32x32 and a 16x16 versions. The menu, since it will be used only when no document is available, should provide functionality that does not process a document. It should allow the user to create a new document, to open an existing document, and to quit the application. Such a menu can have the following items:



As done since Lesson 3 (when we studied resources), these two resources are sufficient to create an application since you can call the CFrameWnd::LoadFrame() method. To create a frame for a Multiple Document Interface, you must derive your frame class from CMDIFrameWnd:

```
BOOL CExercise1App::InitInstance()
{
        // Create a frame for the window
        CMainFrame* pMainFrame = new CMainFrame;
        if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
                return FALSE;
        m_pMainWnd = pMainFrame;

        CCommandLineInfo cmdInfo;

        // Dispatch commands specified on the command line
        if (!ProcessShellCommand(cmdInfo))
                return FALSE;
        pMainFrame->ShowWindow(m_nCmdShow);
        pMainFrame->UpdateWindow();
```

```
	return TRUE;
}
```

The above code allows only creating a window fra me for the parent window. To allow the user to interact with the computer through your MDI, you must provide a template for a document. The child document must have its own frame, distinct from that of the parent. To provide this frame, you have two main alternatives. You can directly use the CMDIChildWnd class or you can derive your own class from CMDIChildWnd:

```
class CChildFrame : public CMDIChildWnd
{
	DECLARE_DYNCREATE(CChildFrame)

	DECLARE_MESSAGE_MAP()
};
```

As opposed to a Single Document Interface application, to create a Multiple Document Interface (MDI) application, you use the CMultiDocTemplate class which, like the CSingleDocTemplate class, is derived from CDocTemplate. Therefore, you must declare a pointer variable to CMultiDocTemplate using the new operator, then use its constructor to initialize the template. The syntax of the CMultiDocTemplate constructor is:

```
CMultiDocTemplate(UINT nIDResource,
		CRuntimeClass* pDocClass,
		CRuntimeClass* pFrameClass,
		CRuntimeClass* pViewClass);
```

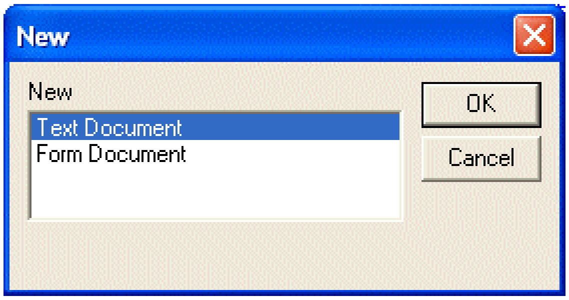
To make a document different from the parent, you must create an additional series of resources that share a common name other than IDR_MAINFRAME. The most basic resources you should create are a menu and an icon. There are two main types of MDI applications.

One kind of application may use only one particular category of documents such as only text-based. Because text-based documents can include ASCII text files, rich text documents (RTF), HTML files, script-based documents (JavaScript, VCScript, Perl, PHP, etc), etc, such an application may be configured to open only that type of document. To create such an MDI application, in the constructor of the CMultiDocTemplate object that you are using, specify a CView derived class (CEditView, CListView, etc) or your own class you derived from CView or one of its children. Here is an example:

```
BOOL CMultiEdit1App::InitInstance()
{
	CMultiDocTemplate* pDocEdit;
	pDocEdit = new CMultiDocTemplate(
		IDR_EDITTYPE,
		RUNTIME_CLASS(CMultiEdit1Doc),
		RUNTIME_CLASS(CChildFrame),
		RUNTIME_CLASS(CEditView));
	AddDocTemplate(pDocEdit);

	// create main MDI Frame window
	CMainFrame* pMainFrame = new CMainFrame;
	if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
		return FALSE;
	m_pMainWnd = pMainFrame;

	CCommandLineInfo cmdInfo;
	// Dispatch commands specified on the command line
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;
	pMainFrame->ShowWindow(m_nCmdShow);
	pMainFrame->UpdateWindow();

	return TRUE;
}
```

In this case, the user can create and/or open as many text files as the computer memory would allow.

Another type of MDI application you can create would allow the user to create or open more than one type of document. To provide this functionality, you must specify a template for each type. Again, there are various types of techniques you can use. You can ask the user to select the type of document he or she wants to create when the application starts:

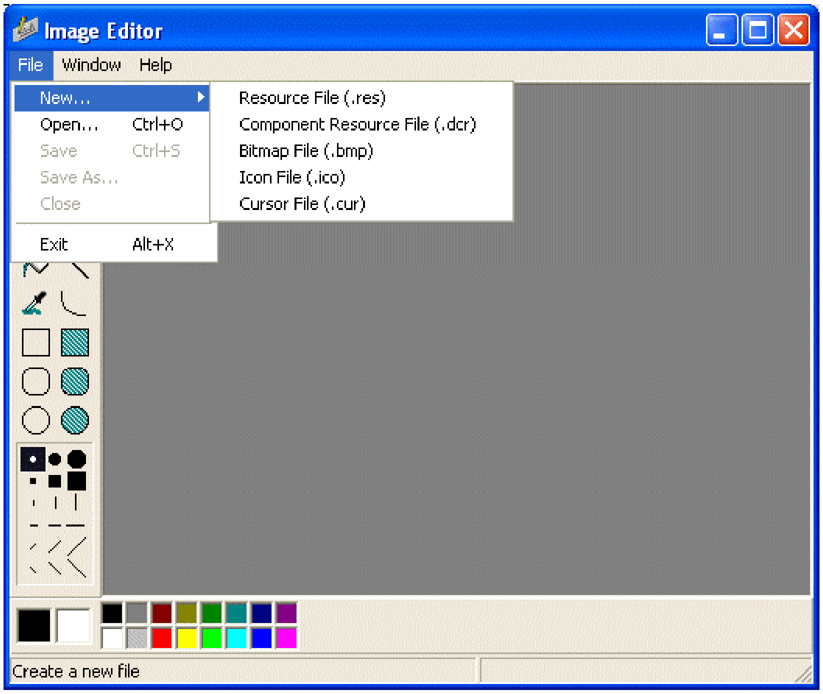


To do this, you can create a template for each type of document using a CDocMultiDocTemplate constructor for each:

```
BOOL CMultiEdit1App::InitInstance()
{
	CMultiDocTemplate* pDocEdit;
	pDocEdit = new CMultiDocTemplate(
		IDR_EDITTYPE,
		RUNTIME_CLASS(CMultiEdit1Doc),
		RUNTIME_CLASS(CChildFrame),
		RUNTIME_CLASS(CEditView));
	AddDocTemplate(pDocEdit);

	CMultiDocTemplate* pDocForm;
	pDocForm = new CMultiDocTemplate(
		IDR_FORMTYPE,
		RUNTIME_CLASS(CMultiEdit1Doc),
		RUNTIME_CLASS(CChildFrame),
		RUNTIME_CLASS(CEmplRecords));
	AddDocTemplate(pDocForm);

	// create main MDI Frame window
	CMainFrame* pMainFrame = new CMainFrame;
	if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
		return FALSE;
	m_pMainWnd = pMainFrame;

	CCommandLineInfo cmdInfo;
```

```
            // Dispatch commands specified on the command line
            if (!ProcessShellCommand(cmdInfo))
                        return FALSE;
            pMainFrame->ShowWindow(m_nCmdShow);
            pMainFrame->UpdateWindow();

            return TRUE;
}
```

You can also display an empty main frame when the application starts and let the user create a document based on the available types from the main menu. Here is such a menu from Borland Image Editor:



As you can see, creating an MDI is not necessarily too difficult but it can involve a few more steps than an SDI. Therefore, whenever we need an MDI, unless specified otherwise, we ill use the AppWizard.

5.4.3 The Visual C++ AppWizard

Microsoft Visual C++ provides various wizards to assist you with performing some tasks. Particularly, to create an application, it offers the AppWizard. AppWizard offers a convenient step-by-step series of choices to create various types of applications such as an SDI or an MDI (it offers many other choices of applications).

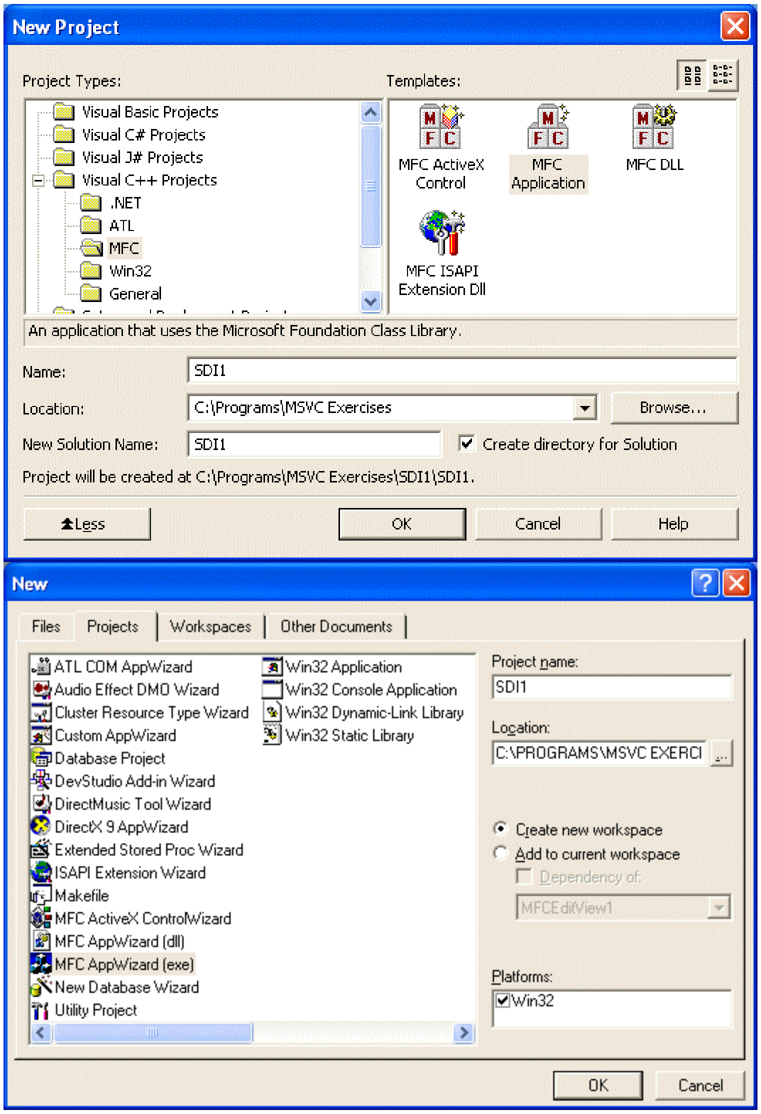
5.5 The AppWizard

5.5.1 An SDI With AppWizard

To get help in creating a single document interface application, you can use the Visual C++' AppWizard. to do this, from the New Project dialog box (MSVC .Net) or the Projects property page of the New dialog box (MSVC), select MFC Application or MFC AppWizard (exe), specify the name and location, then click OK. On the next dialog box, you must specify the Application Type as Single Document. One of the decisions you must make is to specify the view of application you are creating, that is, you must select CView or one of its children as the base class and click Finish. MFC's AppWizard offers various options to create a starting application as complete as possible. We will view these when needed.

Practical Learning: Creating an SDI

1. To create a new application, display the New or the New Project dialog box
2. Select MFC Application or MFC AppWizard (exe)
3. Specify the application Name as SDI1



An application that uses the Microsoft Foundation Class Library.

Name: SDI1

Location: C:\Programs\MSVC Exercises

New Solution Name: SDI1 ☑ Create directory for Solution

Project will be created at C:\Programs\MSVC Exercises\SDI1\SDI1.

Project name: SDI1

Location: C:\PROGRAMS\MSVC EXERCI

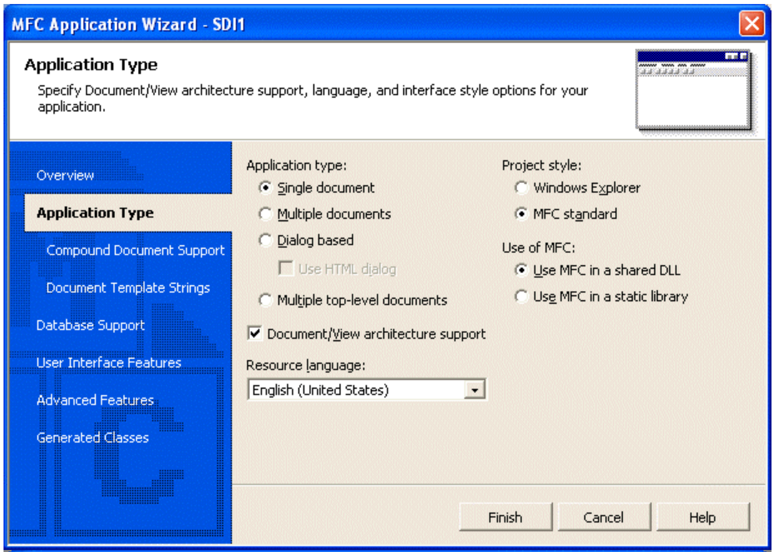
● Create new workspace

○ Add to current workspace

□ Dependency of: MFCEditView1

Platforms: ☑Win32

4. Click OK
5. Specify the application type as Single Document:



MFC Application Wizard - SDI1

Application Type

Specify Document/View architecture support, language, and interface style options for your application.

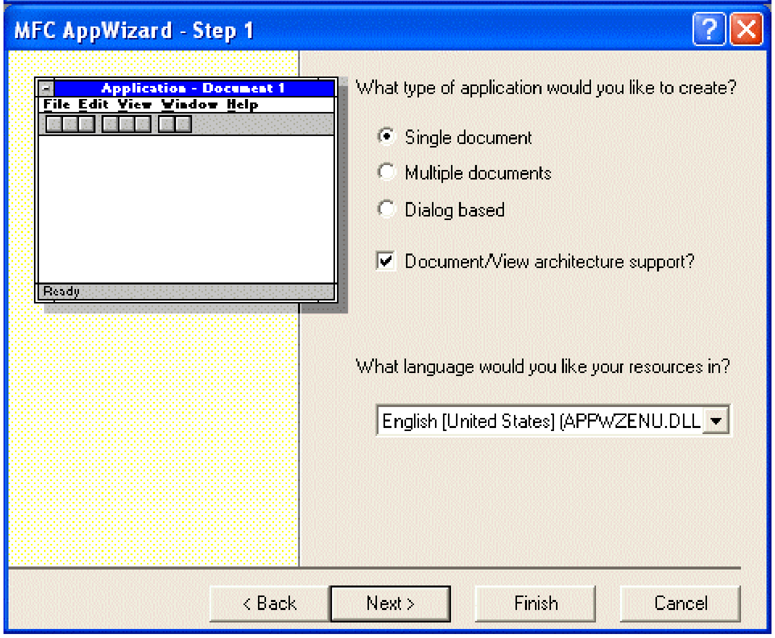
Application type:
- ● Single document
- ○ Multiple documents
- ○ Dialog based
  - □ Use HTML dialog
- ○ Multiple top-level documents

☑ Document/View architecture support

Resource language: English (United States)

Project style:
- ○ Windows Explorer
- ● MFC standard

Use of MFC:
- ● Use MFC in a shared DLL
- ○ Use MFC in a static library



MFC AppWizard - Step 1

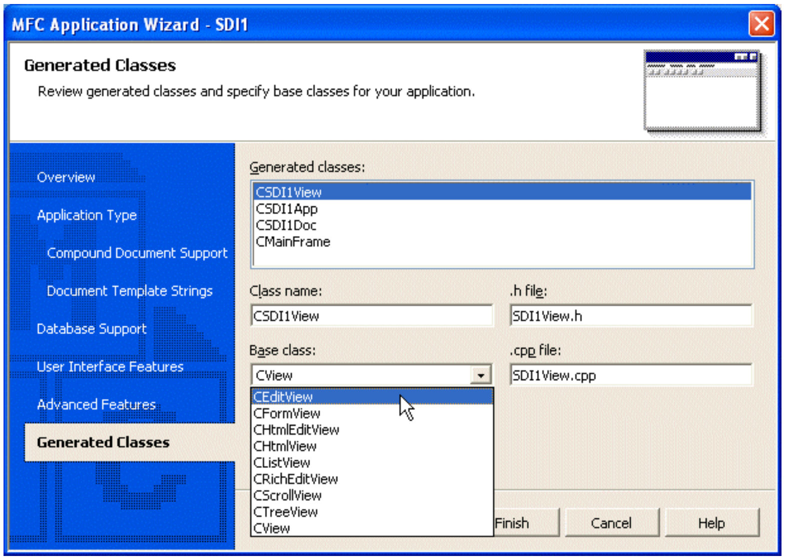
What type of application would you like to create?
- ● Single document
- ○ Multiple documents
- ○ Dialog based

☑ Document/View architecture support?

What language would you like your resources in?

English [United States] (APPWZENU.DLL

6. Specify the Base Class as CeditView



MFC Application Wizard - SDI1

Generated Classes

Review generated classes and specify base classes for your application.

Generated classes:

CSDI1View
CSDI1App
CSDI1Doc
CMainFrame

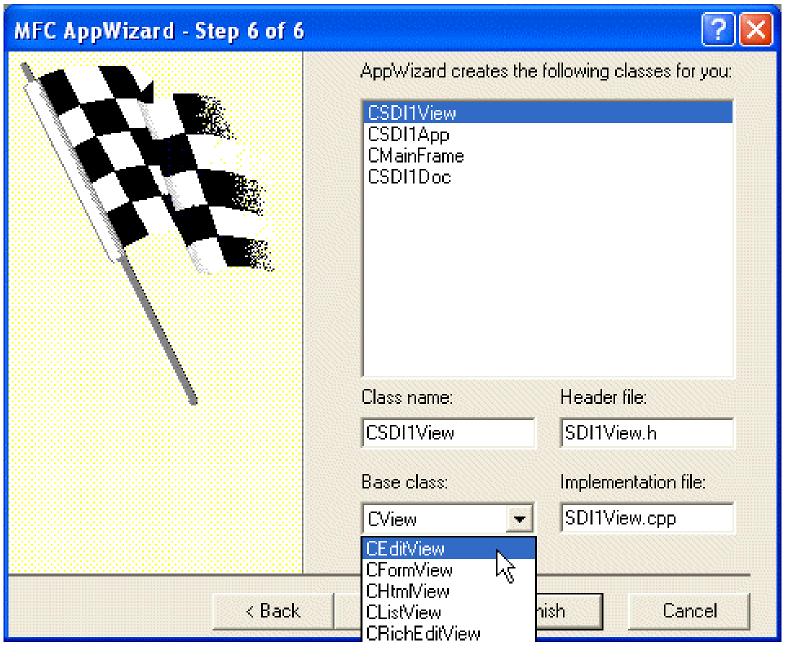
Class name: CSDI1View

.h file: SDI1View.h

Base class: CView

.cpp file: SDI1View.cpp

CEditView
CFormView
CHtmlEditView
CHtmlView
CListView
CRichEditView
CScrollView
CTreeView
CView



MFC AppWizard - Step 6 of 6

AppWizard creates the following classes for you:

CSDI1View
CSDI1App
CMainFrame
CSDI1Doc

Class name: CSDI1View

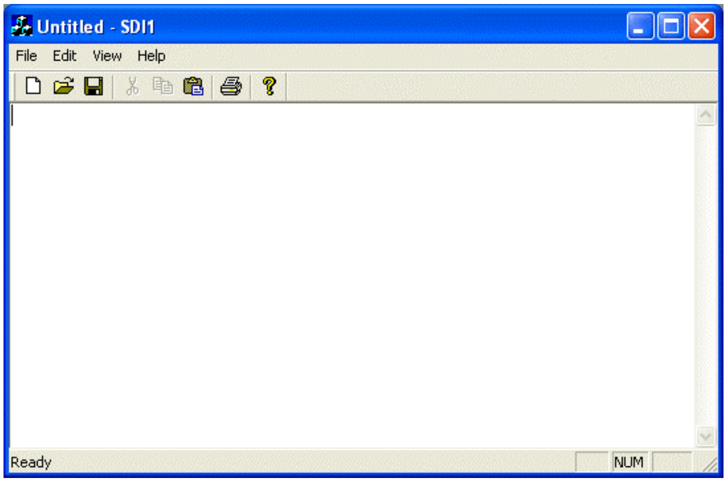
Header file: SDI1View.h

Base class: CView

Implementation file: SDI1View.cpp

CEditView
CFormView
CHtmlView
CListView
CRichEditView

7. Click Finish
8. Test the application



9. Close the window and return to MSVC

From now on, when you are asked to create an SDI, refer to this section.

5.5.2 An MDI With AppWizard

To reduce the amount of work involved with creating an MDI, you can use the MFC's AppWizard. To do this, in the MFC AppWizard, select the Multiple Document radio button.

If you want to create an application that would use a single type of document, select the desired base class.

If you want to create a Windows Explorer type of application, you should select a Single Document application type and select the project style as Windows Explorer. Then, in the Generated Classes section or the MFC AppWizard Step 6, specify a class for the left frame and another base class for the right frame. A classic example would have TreeView class for the left pane and a CListView class for the right side.

If you have created an SDI, you can change it into a multi-view application by adding a CView-derived class to the application. For example, in MSVC 6, if you had created an SDI based on a tree view and you want to create a second view as a form for the right frame, right-click the first node in Class View and click New Form.

Chapter 6:
The Graphical Device Interface

- ✍ Introduction to the GDI
- ✍ The Process of Drawing
- ✍ GDI Lines and Shapes

6.1 Introduction to the GDI

6.1.1 The Device Context

Imagine you want to draw an orange. You can pick up a piece of stone and start drawing somewhere. If you draw on the floor, the next rain is likely to wipe your master piece away. If you draw on somebody's wall, you could face a law suit. Nevertheless, you realize that, to draw, you need at least two things besides your hands and your imagination: a platform to draw on and a tool to draw with.

As it happens, drawing in a studio and drawing on the computer have differences. To draw in real life, the most common platform is probably a piece of paper. Then, you need a pen that would show the evolution of your work. Since a pen can have or use only one color, depending on your goal, one pen may not be sufficient, in which case you would end up with quite a few of them. Since the human hand sometimes is not very stable, if you want to draw straight line, you may need a ruler. Some other tools can also help you draw geometric figures faster.

A device context is everything under one name. It is an orchestra, an ensemble of what need in order to draw. It includes the platform you draw on, the dimensioning of the platform, the orientation and other variations of your drawing, the tools you need to draw on the platform, the colors, and various other accessories that can complete your imagination.

When using a computer, you certainly cannot position tools on the table or desktop for use as needed. To help with drawing on the Windows operating system, Microsoft created the Graphical Device Interface, abbreviated as GDI. It is a set of classes, functions, variables, and constants that group all or most of everything you need to draw on an application. The GDI is provided as a library called Gdi.dll and is already installed on your computer.

6.1.2 Grabbing the Device Context

As mentioned already, in order to draw, you need at least two things: a platform and a tool. The platform allows you to know what type of object you are drawing on and how you can draw on it. On a Windows application, you get this platform by creating a device context.

A device context is actually a whole class that provides the necessary drawing tools to perform the job. For example, it provides functions for selecting the tool to use when drawing. It also provides functions to draw text, lines, shapes etc. To select the platform on which to draw, that is, to create a device context, the MFC provides various classes:

CDC: This is the most fundamental class to draw in MFC. It provides all of the primary functions used to perform the basic drawing steps. In order to use this class, first declare a variable from it. Then call the BeginPaint() function to initialize the variable using the PAINSTRUCT class. Once the variable has been initialized, you can use it to draw. After using the device context call the EndPaint() function to terminate the drawing.

CPaintDC: Unlike the CDC object, the CPaintDC inherently initializes its drawing by calling the BeginPaint() function when you declare a CPaintDC variable. When the drawing with this class is over, it calls the EndPaint() to terminate the drawing.

CClientDC: This class is used when you want to draw in the client area of a window.

CMetaFileDC: This class is used to create Windows metafiles.

6.2 The Process of Drawing

6.2.1 Getting a Device Context

In order to draw using a device context, you must first declare a variable of the CDC class. This can be done as follows:

```
CDC dc;
```

To help with this, the CView class provides a virtual member function that can carry the drawing assignments on a program. The method provided is OnDraw() and its syntax is:

```
void OnDraw(CDC* pDC) = 0;
```

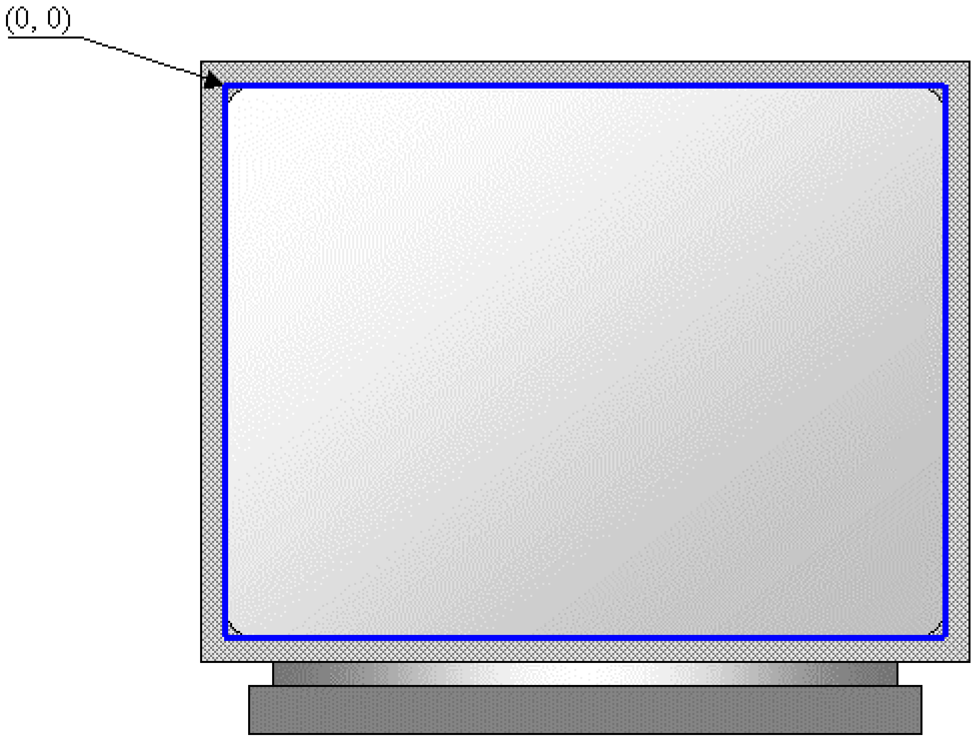
This means that each class that derives from CView must provides its own implementation of this method. If you use AppWizard to create an application and select a CView-derived base class, the AppWizard would define a basic implementation of OnDraw() for you. For a CView-based application, this can be a good place to perform a lot of your drawing.

As you can see, the OnDraw() method is passed a pointer to CDC. This allows you to use the pDC pointer as a variable and draw on the view object with it.

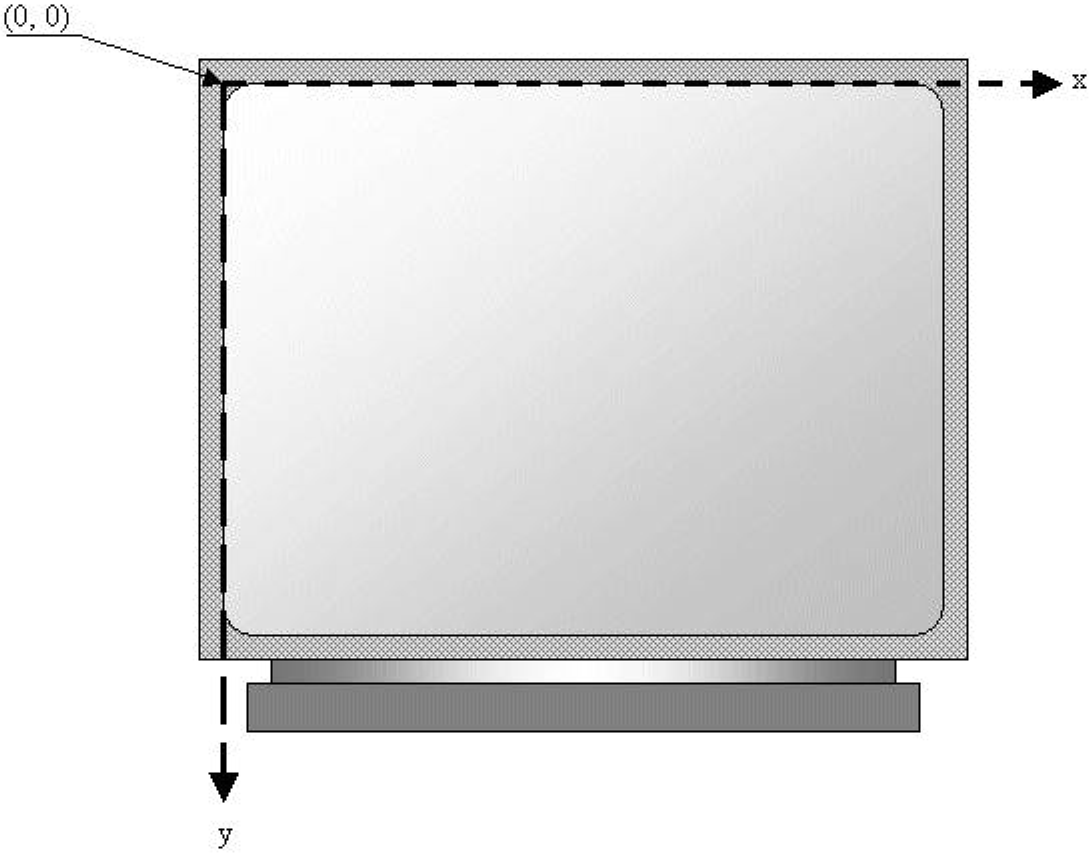
Declaring a CDC variable or receiving it as an argument to a function gives you a device context (DC) you can use. This DC initializes the drawing with some default objects such as a pen to draw the most basic points or lines.

6.2.2 Starting a Device Context's Shape

To keep track of the various drawings, the device context uses a coordinate system that has its origin (0, 0) on the top-left corner of the desktop:



Anything that is positioned on the screen is based on this origin. This coordinate system can get the location of an object using a horizontal and a vertical measurements. The horizontal measures are based on an x axis that moves from the origin to the right right direction. The vertical measures use a y axis that moves from the origin to the bottom direction:



This means that, if you start drawing something such as a line, it would start on the origin and continue where you want it to stop.

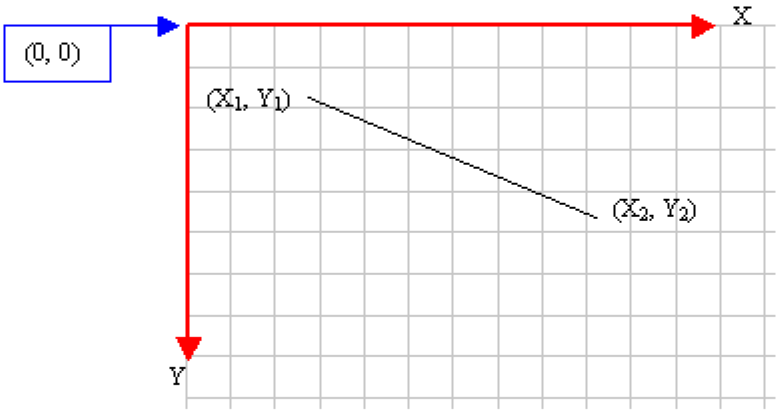
Practical Learning: Starting a Drawing

Start Microsoft Visual C++ and create a Single Document application called GDI1 and based on the CView class

6.3 GDI Lines and Shapes

6.3.1 Lines

A line is a junction of two points. This means that a line has a beginning and an end:



The beginning and the end are two distinct points that can be either POINT, CPoint, or a mix of a POINT and a CPoint values. In real life, before drawing, you should define where you would start. To help with this, the CDC class provides the MoveTo() method. It comes in two versions that are:

```
CPoint MoveTo(int x, int y);
CPoint MoveTo(POINT point);
```

The origin of a drawing is specified as a CPoint value. You can provide its horizontal and vertical measures as x and y or you can pass it as a POINT or a CPoint value.

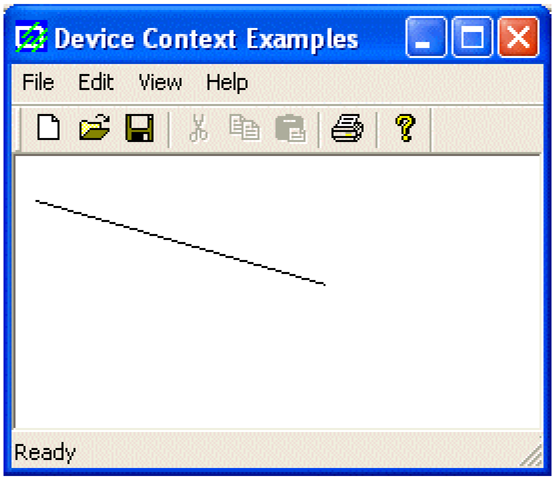
To end the line, you use the CDC::LineTo() method. It comes it two versions declared as follows:

```
BOOL LineTo(int x, int y);
BOOL LineTo(POINT point);
```

The end of a line can be defined by its horizontal (x) and its vertical measures (y). It can also be provided as a POINT or a CPoint value. The last point of a line is not part of the line. The line ends just before reaching that point.

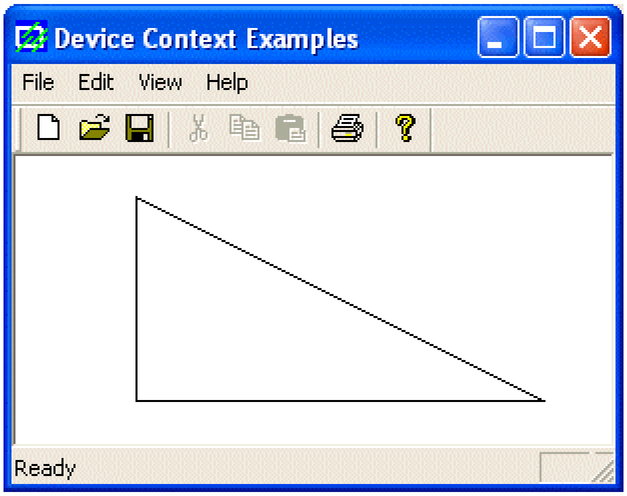
Here is an example that draws a line starting at a point defined as (10, 22) coordinates and ending at (155, 64):

```
void CExoView::OnDraw(CDC* pDC)
{
    pDC->MoveTo(10, 22);
    pDC->LineTo(155, 64);
}
```



We have mentioned that the CDC::MoveTo() method is used to set the starting position of a line. When using LineTo(), the line would start from the MoveTo() point to the LineTo() end. As long as you do not call MoveTo(), any subsequent call to LineTo() would draw a line from the previous LineTo() to the new LineTo() point. You can use this property of the LineTo() method to draw various lines. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	pDC->MoveTo(60, 20);
	pDC->LineTo(60, 122);
	pDC->LineTo(264, 122);
	pDC->LineTo(60, 20);
}
```



Practical Learning: Drawing Lines

Access the OnPaint event of your view class and implement it as follows:

```
 void CView1View::OnPaint()
{
	CPaintDC dc(this); // device context for painting
```

```
        // TODO: Add your message handler code here
        CPoint PtLine[] = { CPoint( 50,  50), CPoint(670,  50),
                            CPoint(670, 310), CPoint(490, 310),
                            CPoint(490, 390), CPoint(220, 390),
                            CPoint(220, 310), CPoint( 50, 310),
                            CPoint( 50,  50) };

        dc.MoveTo(PtLine[0]);
        dc.LineTo(PtLine[1]);
        dc.LineTo(PtLine[2]);
        dc.LineTo(PtLine[3]);
        dc.LineTo(PtLine[4]);
        dc.LineTo(PtLine[5]);
        dc.LineTo(PtLine[6]);
        dc.LineTo(PtLine[7]);
        dc.LineTo(PtLine[8]);
        // Do not call CView::OnPaint() for painting messages
}
```

Test the application

6.3.2 Polylines

A polyline is a series of connected lines. The lines are stored in an array of POINT or CPoint values. To draw a polyline, you use the CDC::Polyline() method. Its syntax is:

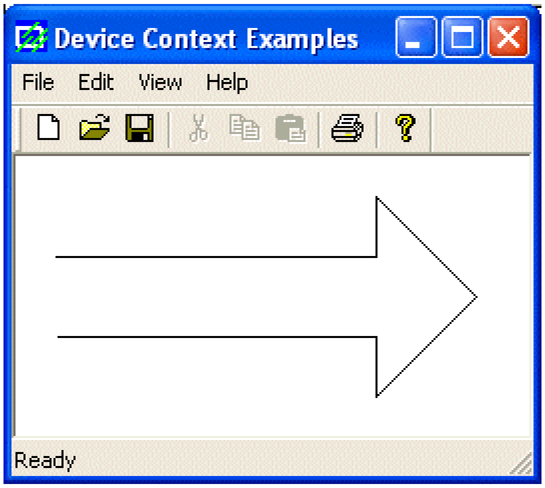
```
BOOL Polyline(LPPOINT lpPoints, int nCount);
```

The lpPoints argument is an array of points that can be of POINT or CPoint types. The nCount argument specifies the number of members of the array. When executing, the compiler moves the starting point to lpPoints[0]. The first line is drawn from lpPoints[0] to lpPoints[1] as in:

```
void CExoView::OnDraw(CDC* pDC)
{
        CPoint Pt[] = { CPoint(60, 20), CPoint(60, 122) };
        pDC->MoveTo(Pt[0]);
        pDC->LineTo(Pt[1]);
}
```

To draw a polyline, you must have at least two points. If you define more than two points, each line after the first would be drawn from the previous point to the next point until all points have been included. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CPoint Pt[7];
        Pt[0] = CPoint(20, 50);
        Pt[1] = CPoint(180, 50);
        Pt[2] = CPoint(180, 20);
        Pt[3] = CPoint(230, 70);
        Pt[4] = CPoint(180, 120);
        Pt[5] = CPoint(180, 90);
        Pt[6] = CPoint(20, 90);
        pDC->Polyline(Pt, 7);
}
```

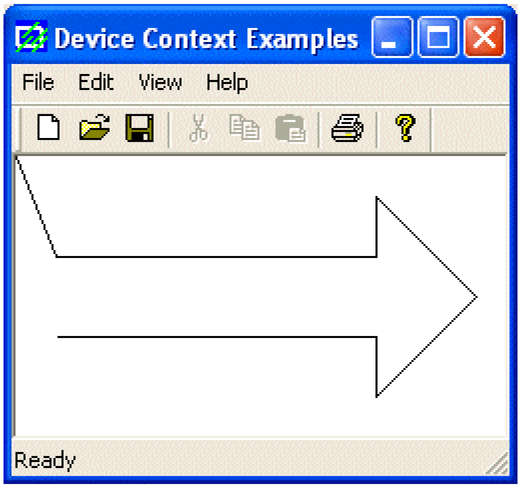


Besides the Polyline() method, the CDC class provides the PolylineTo() member function. Its syntax is:

```
BOOL PolylineTo(const POINT* lpPoints, int nCount);
```

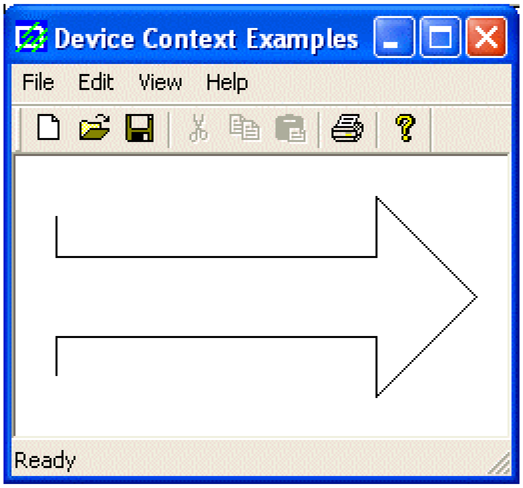
The lpPoints argument is the name of an array of POINT or CPoint objects. The nCount argument specifies the number of points that would be included in the figure. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CPoint Pt[7];

        Pt[0] = CPoint(20, 50);
        Pt[1] = CPoint(180, 50);
        Pt[2] = CPoint(180, 20);
        Pt[3] = CPoint(230, 70);
        Pt[4] = CPoint(180, 120);
        Pt[5] = CPoint(180, 90);
        Pt[6] = CPoint(20, 90);

        pDC->PolylineTo(Pt, 7);
}
```



While the Polyline() method starts the first line at lpPoints[0], the PolylineTo() member function does not control the beginning of the first line. Like the LineTo() method, it simply starts drawing, which would mean it starts at the origin (0, 0). For this reason, if you want to control the starting point of the PolylineTo() drawing, you can use the MoveTo() method:

```
void CExoView::OnDraw(CDC* pDC)
{
	CPoint Pt[7];

	Pt[0] = CPoint(20, 50);
	Pt[1] = CPoint(180, 50);
	Pt[2] = CPoint(180, 20);
	Pt[3] = CPoint(230, 70);
	Pt[4] = CPoint(180, 120);
	Pt[5] = CPoint(180, 90);
	Pt[6] = CPoint(20, 90);

	pDC->MoveTo(20, 30);
	pDC->PolylineTo(Pt, 7);
	pDC->LineTo(20, 110);
}
```



Practical Learning: Drawing Polylines

To draw a polyline, change the event as follows:

```
 void CView1View::OnPaint()
{
        CPaintDC dc(this); // device context for painting

        // TODO: Add your message handler code here
        CPoint PtLine[] = { CPoint( 50,  50), CPoint(670,  50),
                            CPoint(670, 310), CPoint(490, 310),
                            CPoint(490, 390), CPoint(220, 390),
                            CPoint(220, 310), CPoint( 50, 310),
                            CPoint( 50,  50) };
        CPoint PlLine[] = { CPoint( 55,  55), CPoint(665,  55),
                            CPoint(665, 305), CPoint(485, 305),
                            CPoint(485, 385), CPoint(225, 385),
                            CPoint(225, 305), CPoint( 55, 305),
                            CPoint(55, 55) };

        dc.MoveTo(PtLine[0]);
        dc.LineTo(PtLine[1]);
        dc.LineTo(PtLine[2]);
        dc.LineTo(PtLine[3]);
        dc.LineTo(PtLine[4]);
        dc.LineTo(PtLine[5]);
        dc.LineTo(PtLine[6]);
        dc.LineTo(PtLine[7]);
        dc.LineTo(PtLine[8]);

        dc.Polyline(PlLine, 9);
        // Do not call CView::OnPaint() for painting messages
}
```

Test the application

6.3.3 Multiple Polylines

The above polylines were used each as a single entity. That is, a polyline is a combination of lines. If you want to draw various polylines in one step, you can use the CDC::PolyPolyline() method. By definition, the PolyPolyline() member function is used to draw a series of polylines. Its syntax is:

```
BOOL PolyPolyline(const POINT* lpPoints, const DWORD* lpPolyPoints, int nCount);
```

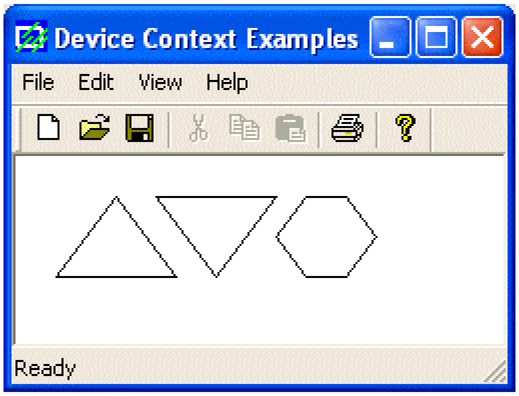
Like the above Polyline() method, the lpPoints argument is an array of POINT or CPoint values. The PolyPolyline() method needs to know how many polylines you would be drawing. Each polyline will use the points of the lpPoints value but when creating the array of points, the values must be incremental. This means that PolyPolyline() will not access their values at random. It will retrieve the first point, followed by the second, followed by the third, etc. Therefore, your first responsibility is to decide where one polyline starts and where it ends. The good news (of course depending on how you see it) is that a polyline does not start where the previous line ended. Each polyline has its own beginning and its own ending point.

Unlike Polyline(), here, the nCount argument is actually the number of shapes you want to draw and not the number of points (reme mber that each polyline "knows" or controls its beginning and end).

The lpPolyPoints argument is an array or positive integers (unsigned long). Each member of this array specifies the number of vertices (lines) that its corresponding polyline will have. For example, imagine you want to draw M, followed by L, followed by Z. The letter M has 4 lines but you need 5 points to draw it. The letter L has 2 lines and you need 3 points to draw it. The letter Z has 3 lines so 4 points are necessary to draw it. You can store this combination of lines in an array defined as { 5, 3, 4 }.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CPoint Pt[15];
        DWORD  lpPts[] = { 4, 4, 7 };

        // Left Triangle
        Pt[0] = Pt[3] = CPoint(50, 20);
        Pt[1] = CPoint(20, 60);
        Pt[2] = CPoint(80, 60);

        // Second Triangle
        Pt[4] = Pt[7] = CPoint(70, 20);
        Pt[5] = CPoint(100, 60);
        Pt[6] = CPoint(130, 20);

        // Hexagon
        Pt[8]  = Pt[14] = CPoint(145, 20);
        Pt[9]  = CPoint(130, 40);
        Pt[10] = CPoint(145, 60);
        Pt[11] = CPoint(165, 60);
        Pt[12] = CPoint(180, 40);
        Pt[13] = CPoint(165, 20);

        pDC->PolyPolyline(Pt, lpPts, 3);
}
```



6.3.4 Polygons

The polylines we have used so far were drawn by defining the starting point of the first line and the end point of the last line and there was no relationship or connection between these two extreme points. A polygon is a closed polyline. In other words, it is a polyline defined so that the end point of the last line is connected to the start point of the first line.

To draw a polygon, you can use the CDC::Polygon() method. Its syntax is:

```
BOOL Polygon(LPPOINT lpPoints, int nCount);
```

This member function uses the same types of arguments as the Polyline() method. The only difference is on the drawing of the line combination. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CPoint Pt[7];
	Pt[0] = CPoint(20, 50);
	Pt[1] = CPoint(180, 50);
	Pt[2] = CPoint(180, 20);
	Pt[3] = CPoint(230, 70);
	Pt[4] = CPoint(180, 120);
	Pt[5] = CPoint(180, 90);
	Pt[6] = CPoint(20, 90);

	pDC->Polygon(Pt, 7);
}
```

Practical Learning: Drawing Polygons

To draw some polygons, change the program as follows:

```
 void CView1View::OnPaint()
{
	CPaintDC dc(this); // device context for painting

	// TODO: Add your message handler code here
	CPoint PtLine[] = { CPoint( 50,  50), CPoint(670,  50),
	                    CPoint(670, 310), CPoint(490, 310),
	                    CPoint(490, 390), CPoint(220, 390),
	                    CPoint(220, 310), CPoint( 50, 310),
	                    CPoint( 50,  50) };

	CPoint Bedroom1[] = { CPoint( 55,  55), CPoint(175,  55),
	                      CPoint(175, 145), CPoint( 55, 145)};
	CPoint Closets[]  = { CPoint( 55, 150), CPoint(145, 150),
	                      CPoint(145, 205), CPoint( 55, 205) };
	CPoint Bedroom2[] = { CPoint(55, 210), CPoint(160, 210),
	                      CPoint(160, 305), CPoint(55, 305) };

	dc.MoveTo(PtLine[0]);
	dc.LineTo(PtLine[1]);
	dc.LineTo(PtLine[2]);
	dc.LineTo(PtLine[3]);
	dc.LineTo(PtLine[4]);
```

```
        dc.LineTo(PtLine[5]);
        dc.LineTo(PtLine[6]);
        dc.LineTo(PtLine[7]);
        dc.LineTo(PtLine[8]);

        dc.Polygon(Bedroom1, 4);
        dc.Polygon(Closets, 4);
        dc.Polygon(Bedroom2, 4);
        // Do not call CView::OnPaint() for painting messages
}
```

Test the application and return to MSVC

6.3.5 Multiple Polygons

If you want to draw multiple polygons, you can use the CDC::PolyPolygon() method whose syntax is:

```
BOOL PolyPolygon(LPPOINT lpPoints, LPINT lpPolyCounts, int nCount);
```

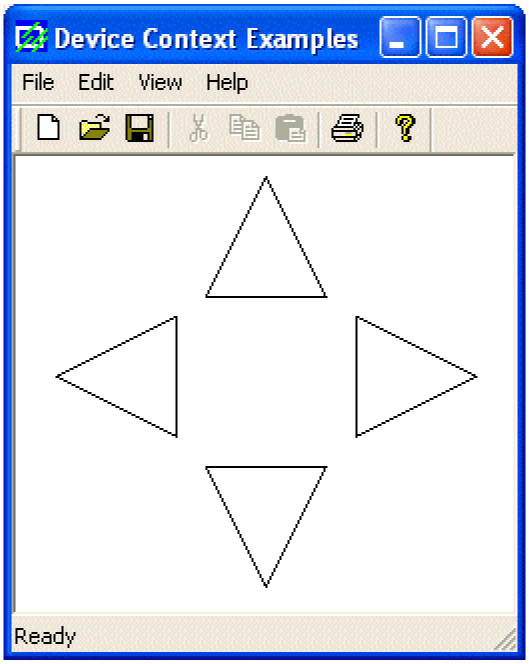
Like the Polygon() method, the lpPoints argument is an array of POINT or CPoint values. The PolyPolygon() method needs to know the number of polygons you would be drawing. Each polygon uses the points of the lpPoints values but when creating the array of points, the values must be incremental. This means that PolyPolygon() will not randomly access the values of lpPoints. Each polygon has its own set of points.

Unlike Polygon(), the nCount argument of PolyPolygon() is the number of polygons you want to draw and not the number of points.

The lpPolyCounts argument is an array or integers. Each member of this array specifies the number of vertices (lines) that its polygon will have..

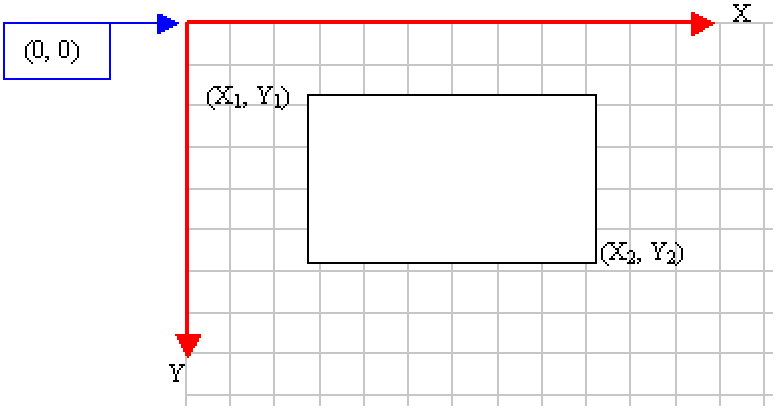
Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CPoint Pt[12];
        int lpPts[] = { 3, 3, 3, 3 };

        // Top Triangle
        Pt[0] = CPoint(125,  10);
        Pt[1] = CPoint( 95,  70);
        Pt[2] = CPoint(155,  70);

        // Left Triangle
        Pt[3] = CPoint( 80,  80);
        Pt[4] = CPoint( 20, 110);
        Pt[5] = CPoint( 80, 140);

        // Bottom Triangle
        Pt[6] = CPoint( 95, 155);
        Pt[7] = CPoint(125, 215);
        Pt[8] = CPoint(155, 155);

        // Right Triangle
        Pt[9] = CPoint(170,  80);
        Pt[10] = CPoint(170, 140);
        Pt[11] = CPoint(230, 110);

        pDC->PolyPolygon(Pt, lpPts, 4);
}
```



6.3.6 Rectangles and Squares

A rectangle is a geometric figure made of four sides that compose four right angles. Like the line, to draw a rectangle, you must define where it starts and where it ends. This can be illustrated as follows:



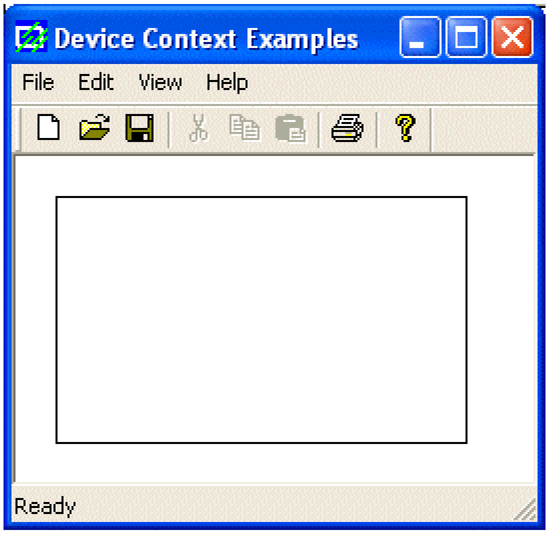
The drawing of a rectangle typically starts from a point defined as (X1, Y1) and ends at another point (X2, Y2).

To draw a rectangle, you can use the CDC::Rectangle() method. Its syntax is:

```
BOOL Rectangle(int x1, int y1, int x2, int y2);
```

As seen on the figure and the formula, a rectangle spans from coordinates (x1, y1) to (x2, y2). Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        pDC->Rectangle(20, 20, 226, 144);
}
```



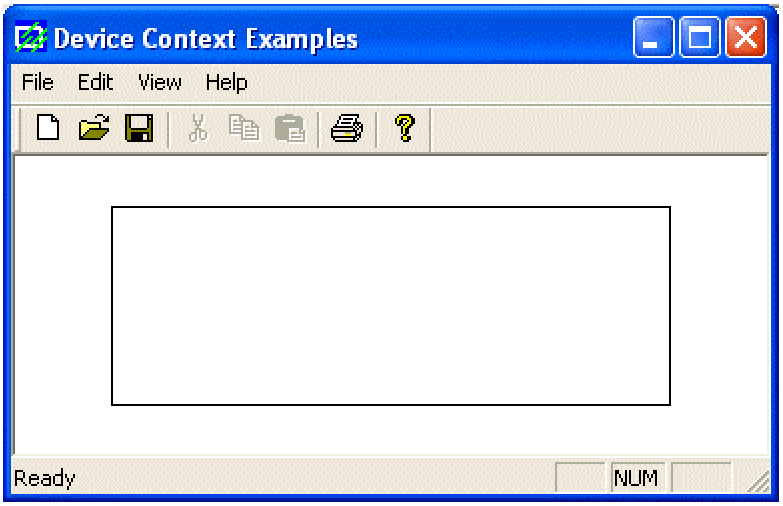
When drawing a rectangle, if the value of x2 is less than that of x1, then the x2 coordinate would mark the left beginning of the figure. This scenario would also apply if the y2 coordinate were lower than y1.

To draw a rectangle, you can also use a RECT or a CRect object. The syntax you would use is:

```
BOOL Rectangle(LPCRECT lpRect);
```

In this case, you must have defined a RECT or a CRect value and pass it as a pointer to the Rectangle() method. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CRect Recto(328, 125, 48, 25);
        pDC->Rectangle(&Recto);
}
```



A square is a rectangle whose sides are all equal. Therefore, to draw a square, when specifying the arguments of the Rectangle() method, make sure that |x1 - x2| = |y1 - y2|.

Practical Learning: Drawing Rectangles

6.3.7 A Rectangle With Edges

The CDC class provides another function member you can use to draw a rectangle. This time you can control how the edges of the rectangle would be drawn. The method used is called DrawEdge and its syntax is:

```
BOOL DrawEdge(LPRECT lpRect, UINT nEdge, UINT nFlags);
```

The lpRect argument is passed as a pointer to a RECT or CRect, which is the rectangle that would be drawn.

The nEdge value specifies how the interior and the exterior of the edges of the rectangle would be drawn. It can be a combination of the following constants:

Value	Description
BDR_RAISEDINNER	The interior edge will be raised
BDR_SUNKENINNER	The interior edge will be sunken
BDR_RAISEDOUTER	The exterior edge will be raised
BDR_SUNKENOUTER	The exterior edge will be sunken

These values can be combined using the bitwise OR operator. On the other hand, you can use the following constants instead:

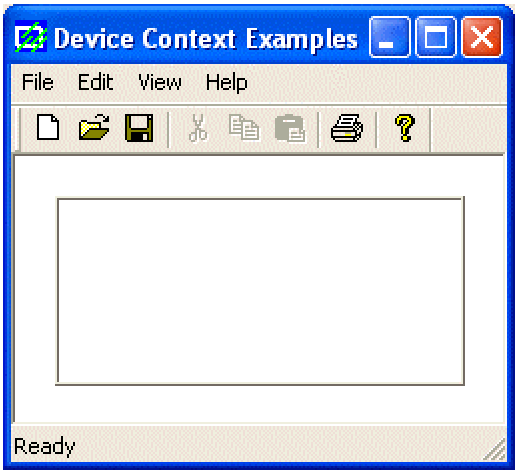
Value	Used For
EDGE_DUMP	BDR_RAISEDOUTER | BDR_SUNKENINNER
EDGE_ETCHED	BDR_SUNKENOUTER | BDR_RAISEDINNER
EDGE_RAISED	BDR_RAISEDOUTER | BDR_RAISEDINNER
EDGE_SUNKEN	BDR_SUNKENOUTER | BDR_SUNKENINNER

The nFlags value specifies what edge(s) would be drawn. It can have one of the following values:

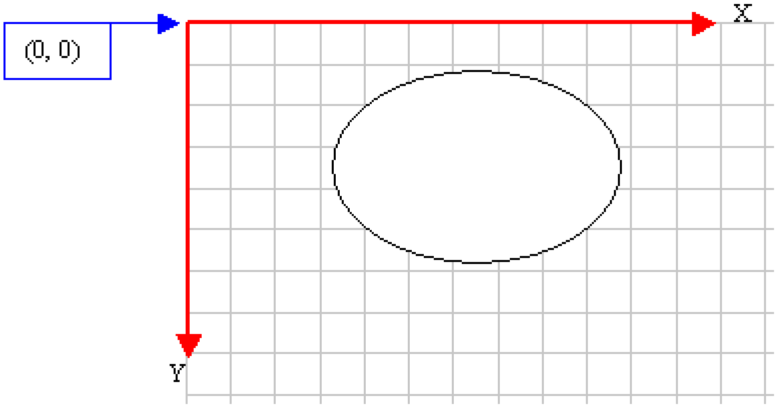
Value	Description
BF_RECT	The entire rectangle will be drawn
BF_TOP	Only the top side will be drawn
BF_LEFT	Only the left side will be drawn
BF_BOTTOM	Only the bottom side will be drawn
BF_RIGHT	Only the right side will be drawn
BF_TOPLEFT	Both the top and the left sides will be drawn
BF_BOTTOMLEFT	Both the bottom and the left sides will be drawn
BF_TOPRIGHT	Both the top and the right sides will be drawn
BF_BOTTOMRIGHT	Both the bottom and the right sides will be drawn
BF_DIAGONAL_ENDBOTTOMLEFT	A diagonal line will be drawn from the top-right to the bottom-left corners
BF_DIAGONAL_ENDBOTTOMRIGHT	A diagonal line will be drawn from the top-left to the bottom-right corners
BF_DIAGONAL_ENDTOPLEFT	A diagonal line will be drawn from the bottom-right to the top-left corners
BF_DIAGONAL_ENDTOPRIGHT	A diagonal line will be drawn from the bottom-left to the top-right corners

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CRect Recto(20, 20, 225, 115);
	pDC->DrawEdge(&Recto, BDR_RAISEDOUTER | BDR_SUNKENINNER, BF_RECT);
}
```



6.3.8 Ellipses and Circles

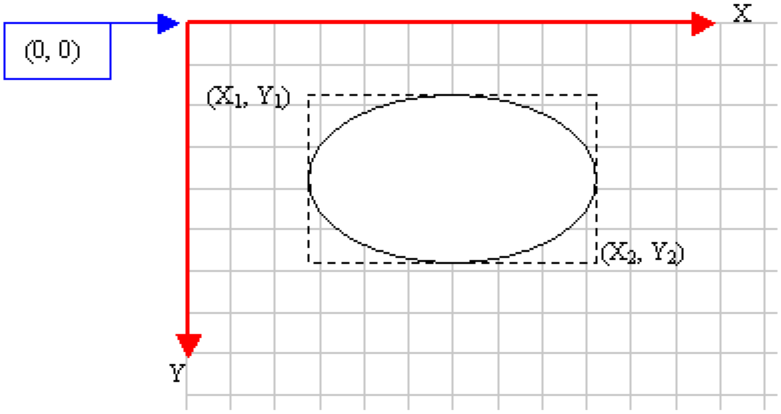


An ellipse is a closed continuous line whose points are positioned so that two points exactly opposite each other have the exact same distant from a central point. It can be illustrated as follows:

Because an ellipse can fit in a rectangle, in GDI programming, an ellipse is defined with regards to a rectangle it would fit in. Therefore, to draw an ellipse, you specify its rectangular corners. The syntax used to do this is:

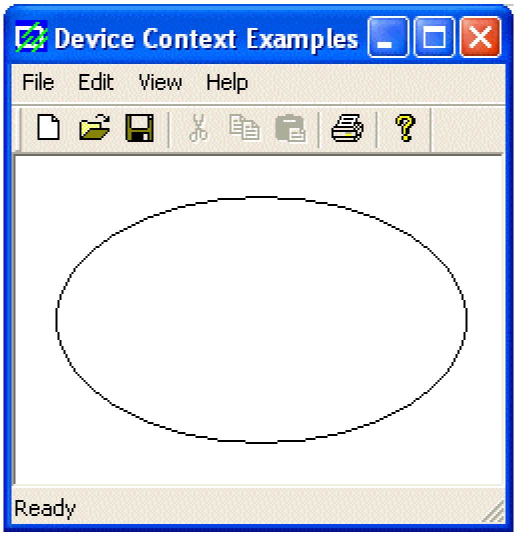
```
BOOL Ellipse(int x1, int y1, int x2, int y2);
```

The arguments of this method play the same roll as those of the Rectangle() method:



Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
            pDC->Ellipse(20, 20, 226, 144);
}
```

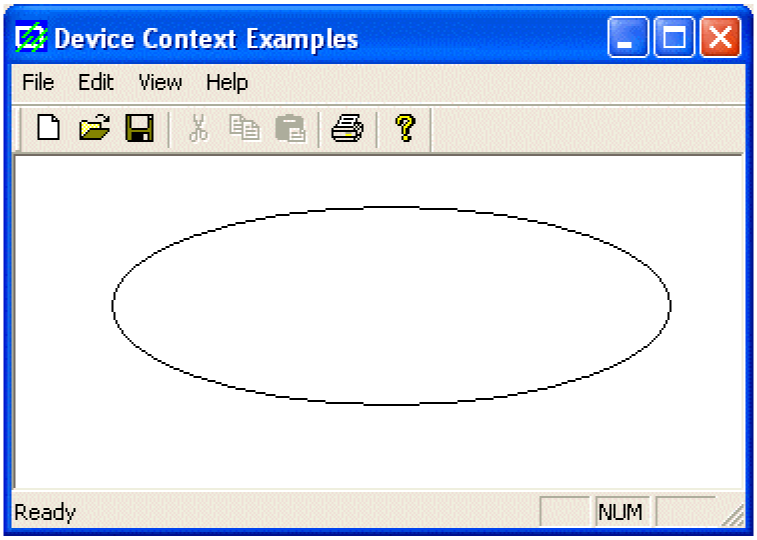


Like the rectangle, you can draw an ellipse using a RECT or CRect object it would fit in. The syntax you would use is:

```
BOOL Ellipse(LPCRECT lpRect);
```

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CRect Recto(328, 125, 48, 25);
	pDC->Ellipse(&Recto);
}
```

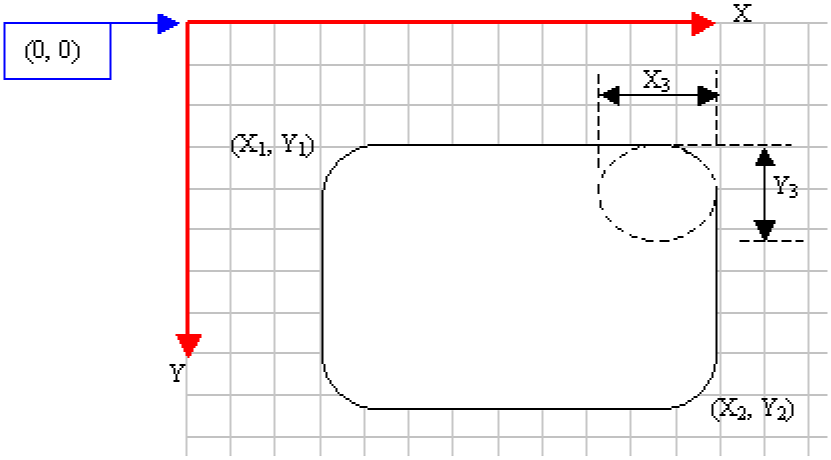


A circle is an ellipse whose all points have the same distance with regards to a central point.

Practical Learning: Drawing Ellipses

6.3.9 Round Rectangles and Round Squares

A rectangle qualifies as round if its corners do not form straight angles but rounded corners. It can be illustrated as follows:



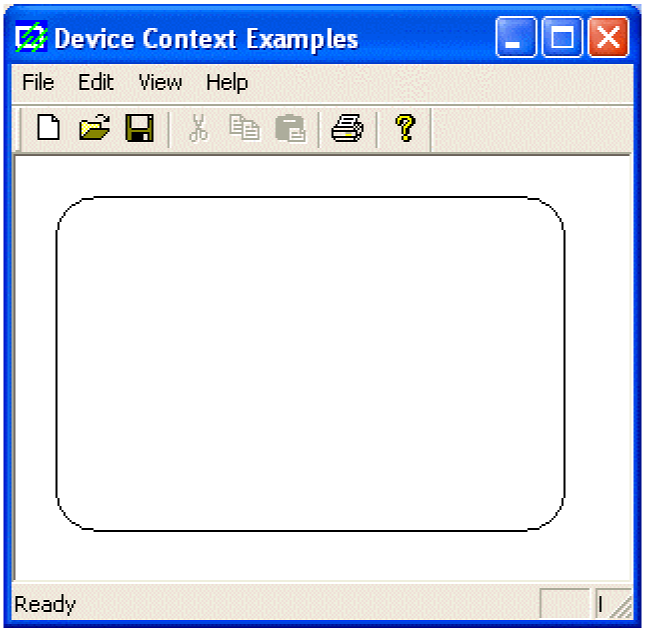
To draw such a rectangle, you can use the CDC::RoundRect() method. Its syntaxes are:

```
BOOL RoundRect(int x1, int y1, int x2, int y2, int x3, int y3);
BOOL RoundRect(PCRECT lpRect, POINT point);
```

When this member function executes, the rectangle is drawn from the (x1, y1) to the (x2, y2) points. The corners are rounded by an ellipse whose width would be x3 and the ellipse's height would be x3.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	pDC->RoundRect(20, 20, 275, 188, 42, 38);
}
```

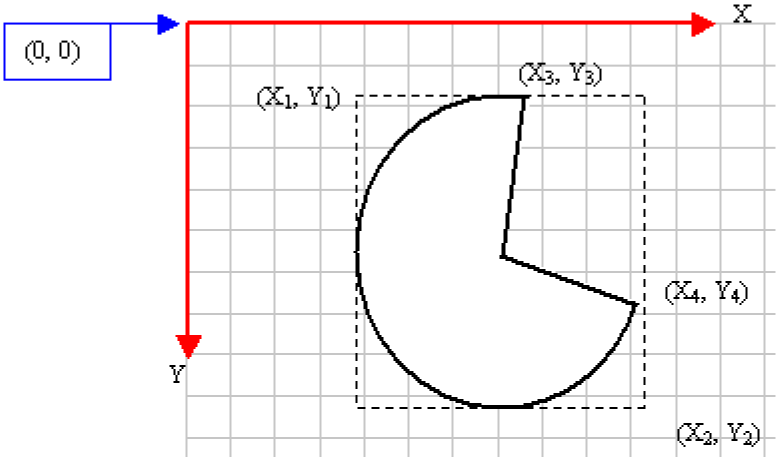


A round square is a square whose corners are rounded.

Practical Learning: Drawing Round Rectangles

6.3.10 Pies

A pie is a fraction of an ellipse delimited by two lines that span from the center of the ellipse to one side each. It can be illustrated as follows:



To draw a pie, you can use the CDC::Pie() method whose syntaxes are:

```
BOOL Pie(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
BOOL Pie(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
```

The (x1, y1) point determines the upper-left corner of the rectangle in which the ellipse that represents the pie fits. The (x2, y2) point is the bottom-right corner of the rectangle.

These two points can also be combined in a RECT or a CRect variable and passed as the lpRect value.

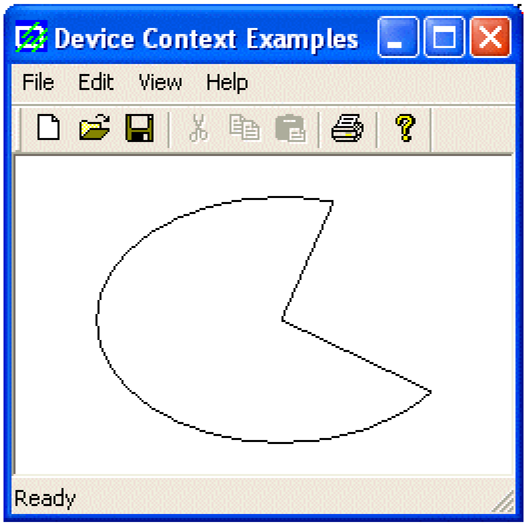
The (x3, y3) point, that can also supplied as a POINT or CPoint for lpStart argument, specifies the starting corner of the pie in a default counterclockwise direction.

The (x4, y4) point, or ptEnd argument, species the end point of the pie.

To complete the pie, a line is drawn from (x3, y3) to the center and from the center to the (x4, y4) points.

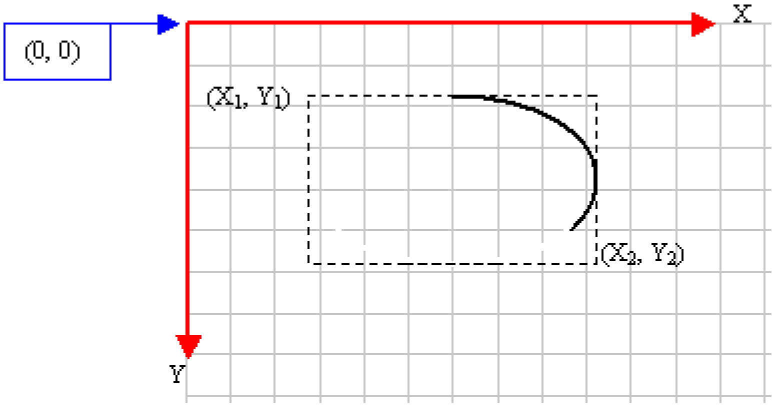
Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
   pDC->Pie(40, 20, 226, 144, 155, 32, 202, 115);
}
```



6.3.11 Arcs

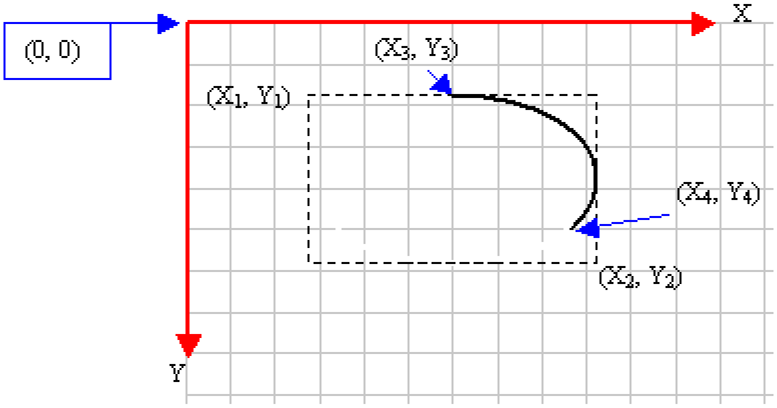
An arc is a portion or segment of an ellipse, meaning an arc is a non-complete ellipse. Because an arc must confirm to the shape of an ellipse, it is defined as it fits in a rectangle and can be illustrated as follows:



To draw an arc, you can use the CDC::Arc() method whose syntax is:

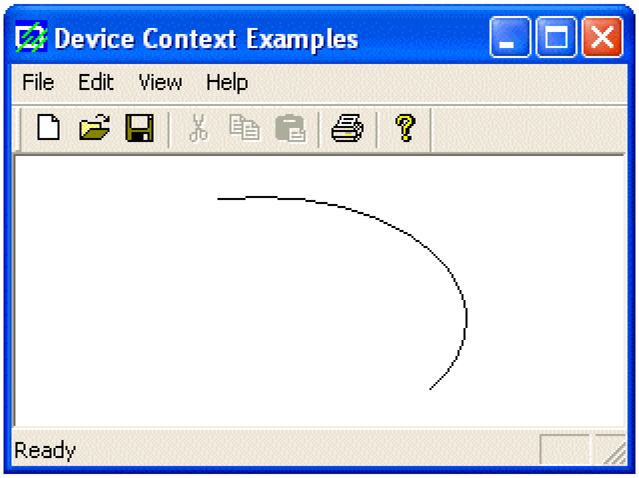
```
BOOL Arc(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
```

Besides the left (x1, y1) and the right (x2, y2) borders of the rectangle in which the arc would fit, an arc must specify where it starts and where it ends. These additional points are set as the (x3, y3) and (x4, y4) points of the figure. Based on this, the above arc can be illustrated as follows:



Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	pDC->Arc(20, 20, 226, 144, 202, 115, 105, 32);
}
```



An arc can also be drawn using the location and size of the rectangle it would fit in. Such a rectangle can be passed to the Arc() method as a RECT or a CRect object. In this case, you must define the beginning and end of the arc as two POINT or CPoint values. The syntax used is:

```
BOOL Arc(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
```

Here is an example that produces the same result as above:

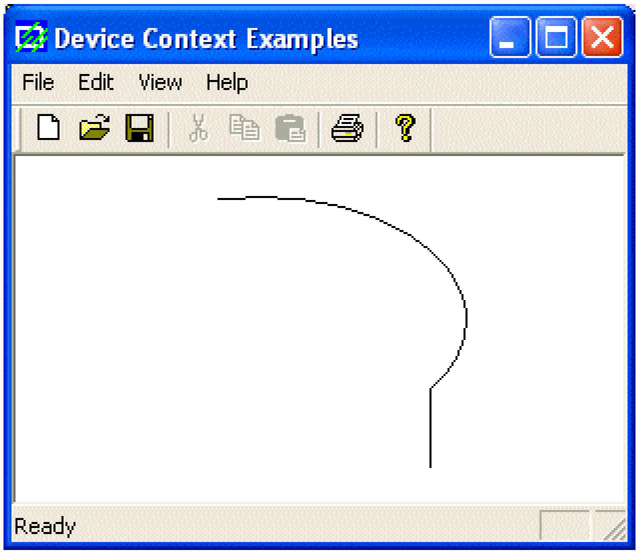
```
void CExoView::OnDraw(CDC* pDC)
{
	CRect Recto(20, 20, 226, 144);
	CPoint Pt1(202, 115);
	CPoint Pt2(105, 32);
	pDC->Arc(Recto, Pt1, Pt2);
}
```

Besides the Arc() method, the CDC class provides the ArcTo() member function used to draw an arc. It also comes in two syntaxes as follows:

```
BOOL ArcTo(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
BOOL ArcTo(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
```

This method uses the same arguments as Arc(). The difference is that while Arc() starts drawing at (x3, y3), ArcTo() does not inherently control the drawing starting point. It refers to the current point, exactly like the LineTo() (and the PolylineTo()) method. Therefore, if you want to specify where the drawing should start, can call CDC::MoveTo() before ArcTo(). Here is an example:

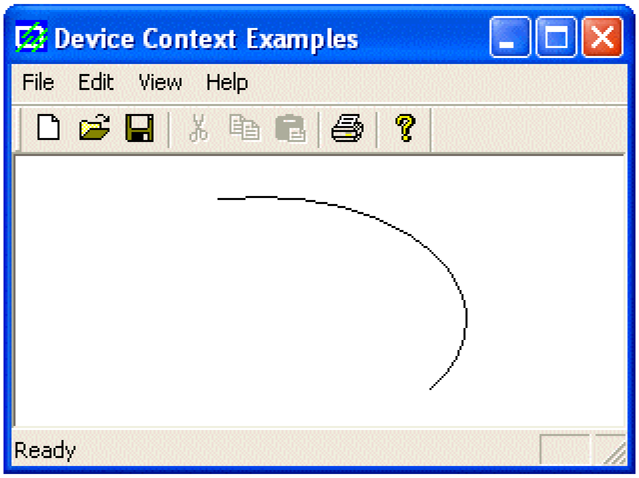
```
void CExoView::OnDraw(CDC* pDC)
{
	CRect Recto(20, 20, 226, 144);
	CPoint Pt1(202, 115);
	CPoint Pt2(105, 32);

	pDC->MoveTo(207, 155);
	pDC->ArcTo(Recto, Pt1, Pt2);
}
```



6.3.12 The Arc's Direction

Here is and arc we drew earlier with a call to Arc():

```
void CExoView::OnDraw(CDC* pDC)
{
	pDC->Arc(20, 20, 226, 144, 202, 115, 105, 32);
}
```



You may wonder why the arc is drawn to the right side of a vertical line that would cross the center of the ellipse instead of the left. This is because the drawing of an arc is performed from right to left or from bottom to up, in the opposite direction of the clock. This is known as the counterclockwise direction. To control this orientation, the CDC class is equipped with the SetArcDirection() method. Its syntax is:

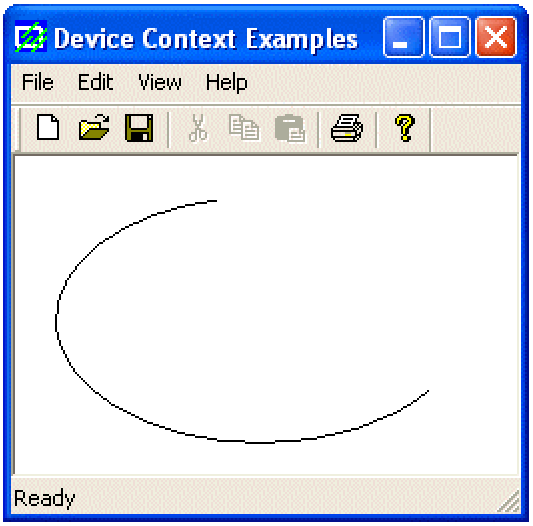
```
int SetArcDirection(int nArcDirection);
```

This method specifies the direction the Arc() method should follow from the starting to the end points. The argument passed as nArcDirection controls this orientation. It can have the following values:

Value	Orientation
AD_CLOCKWISE	The figure is drawn clockwise
AD_COUNTERCLOCKWISE	The figure is drawn counterclockwise

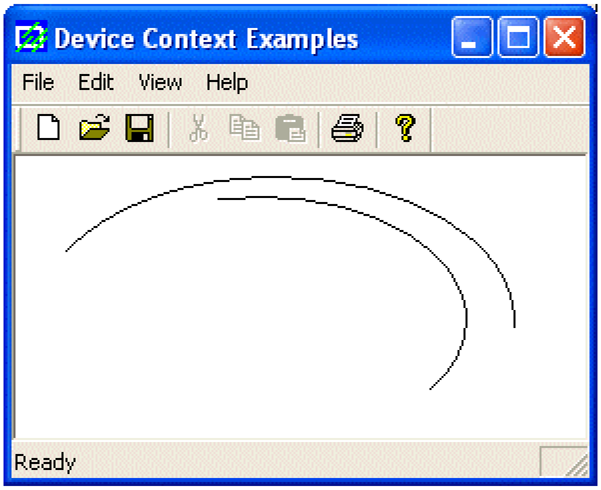
The default value of the direction is AD_COUNTERCLOCKWISE. Therefore, this would be used if you do not specify a direction. Here is an example that uses the same values as above with a different orientation:

```
void CExoView::OnDraw(CDC* pDC)
{
	pDC->SetArcDirection(AD_CLOCKWISE);

	pDC->Arc(20, 20, 226, 144, 202, 115, 105, 32);
}
```



After calling SetArcDirection() and changing the previous direction, all drawings would use the new direction to draw arcs using Arc() or ArcTo() and other figures (such as chords, ellipses, pies, and rectangles). Here is an example:

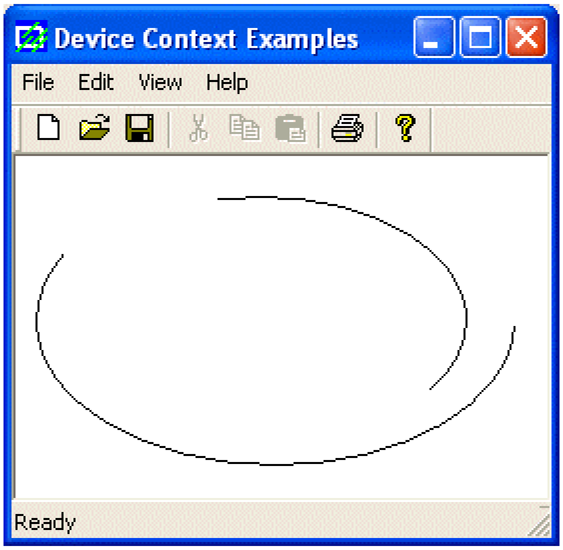
```
void CExoView::OnDraw(CDC* pDC)
{
        pDC->SetArcDirection(AD_COUNTERCLOCKWISE);

        pDC->Arc(20, 20, 226, 144, 202, 115, 105, 32);

        pDC->Arc(10, 10, 250, 155, 240, 85, 24, 48);
}
```



If you want to change the direction, you must call SetArcDirection() with the desired value. Here is an example;

```
void CExoView::OnDraw(CDC* pDC)
{
        pDC->SetArcDirection(AD_COUNTERCLOCKWISE);
        pDC->Arc(20, 20, 226, 144, 202, 115, 105, 32);

        pDC->SetArcDirection(AD_CLOCKWISE);
```

```
            pDC->Arc(10, 10, 250, 155, 240, 85, 24, 48);
}
```



At any time, you can find out the current direction used. This is done by calling the GetArcDirection() method. Its syntax is:

```
int GetArcDirection() const;
```

This method returns the current arc direction as AD_CLOCKWISE or AD_COUNTERCLOCKWISE

6.3.13 Angular Arcs

You can (also) draw an arc using the CDC::AngleArc() method. Its syntax is:

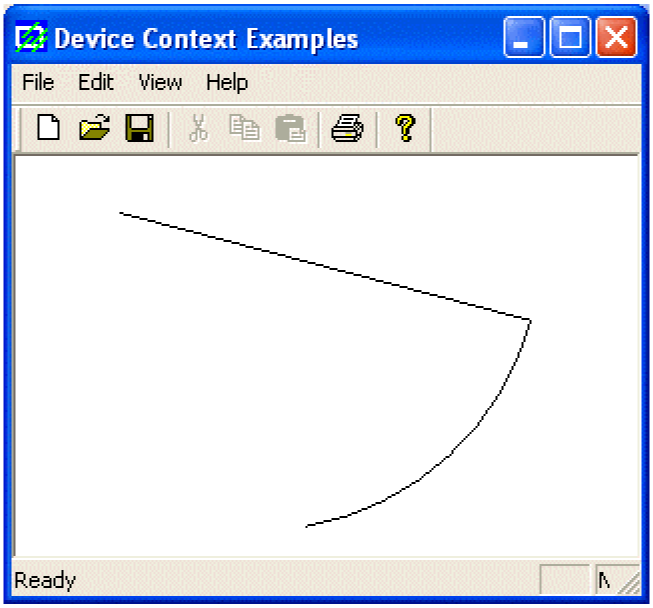
```
BOOL AngleArc(int x, int y, int nRadius, float fStartAngle, float fSweepAngle);
```

This member function draws a line and an arc connected. The arc is based on a circle and not an ellipse. This implies that the arc fits inside a square and not a rectangle. The circle that would be the base of the arc is defined by its center located at C(x, y) with a radius of nRadius. The arc starts at an angle of fStartAngle. The angle is based on the x axis and must be positive. That is, it must range from 0° to 360°. If you want to specify an angle that is below the x axis, such as -15°, use 360º-15°=345°. The last argument, fSweepAngle, is the angular area covered by the arc.

The AngleArc() method does not control where it starts drawing. This means that it starts at the origin, unless a previous call to MoveTo() specified the beginning of the drawing.

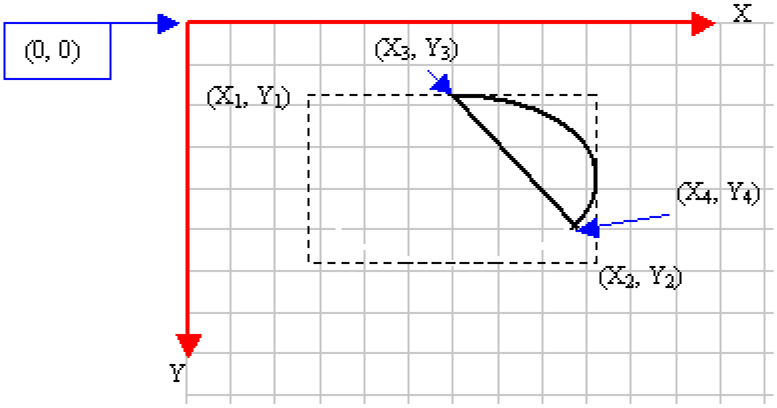
Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
            pDC->MoveTo(52, 28);
            pDC->AngleArc(120, 45, 142, 345, -65);
}
```



6.3.14 Chords

The arcs we have drawn so far are considered open figures because they are made of a line that has a beginning and an end (unlike a circle or a rectangle that do not). A chord is an arc whose two ends are connected by a straight line. In other words, a chord is an ellipse that is divided by a straight line from one side to another:



To draw a chord, you can use the CDC::Chord() method. It is defined as follows:

```
BOOL Chord(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4);
BOOL Chord(LPCRECT lpRect, POINT ptStart, POINT ptEnd);
```

The x1, y1, x2, and y2 are the coordinates of the rectangle in which the chord of the circle would fit. This rectangle can also be defined as a RECT or a CRect value.

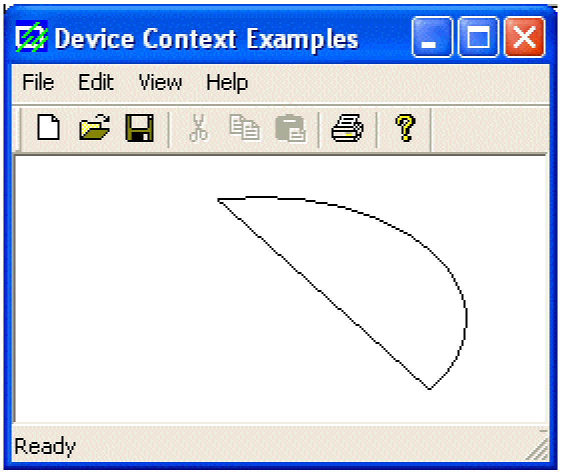
These x3 and y3 coordinates specify where the arc that holds the chord starts. These coordinates can also be defined as the ptStart argument.

The x4 and y4 that can also be defined as ptEnd specify the end of the arc.

To complete the chord, a line is drawn from (x3, y3) to (x4, y4).

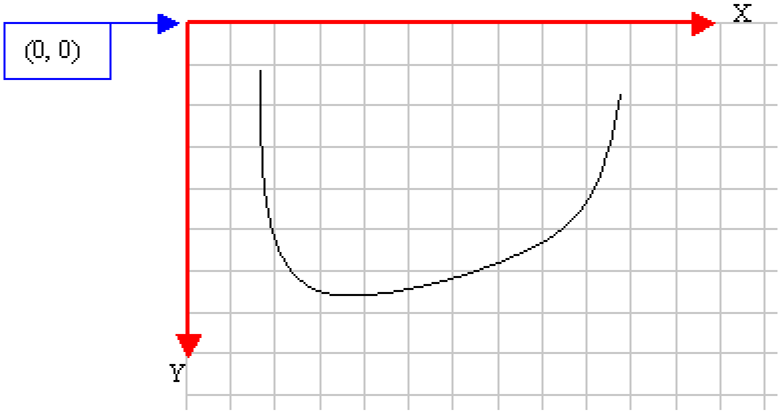
Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	pDC->Chord(20, 20, 226, 144, 202, 115, 105, 32);
}
```

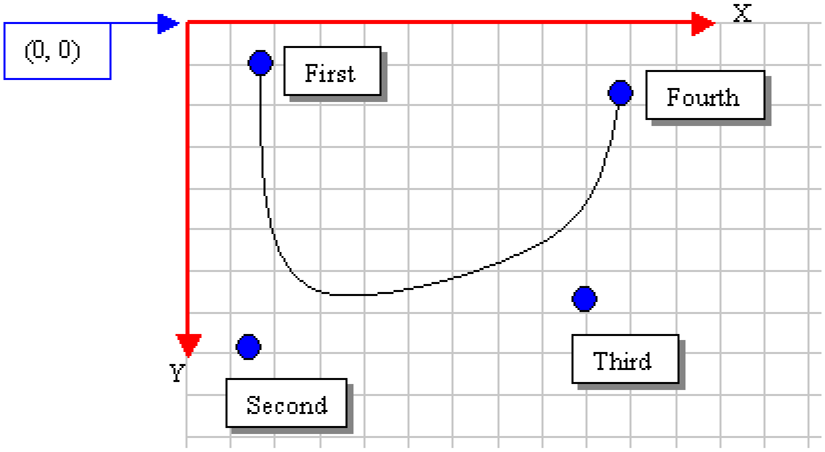


6.3.15 Bézier Curves

A bézier line is an arc that is strictly based on a set number of points instead of on an ellipse. A bézier curve uses at least four points to draw on. A bézier line with four points can be illustrated as follows:



To draw this line (with four points), the compiler would draw a curve from the first to the fourth points. Then it would bend the curve by bringing each middle (half-center) side close to the second and the third points respectively, of course without touching those second and third points. For example, the above bézier curve could have been drawn using the following four points:

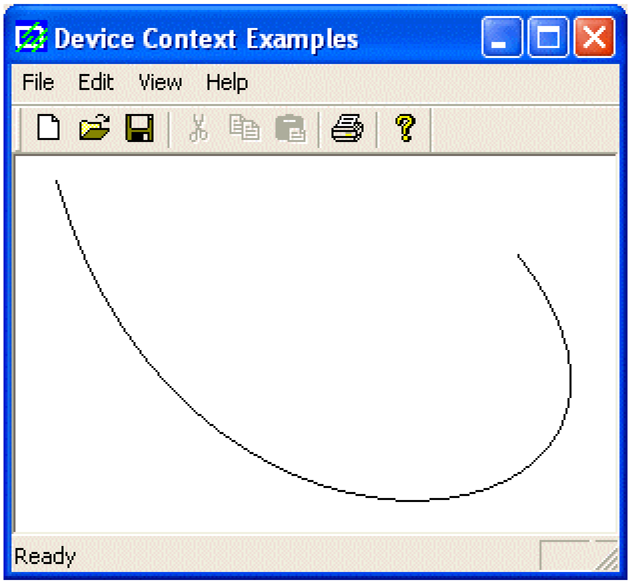


PolyBezier(): To draw a bézier curve, the CDC provides the PolyBezier() method. Its syntax is:

```
BOOL PolyBezier(const POINT* lpPoints, int nCount);
```

The lpPoints argument is an array of POINT or CPoint values. The nCount argument specifies the number of points that will be used to draw the line. Here is an example:

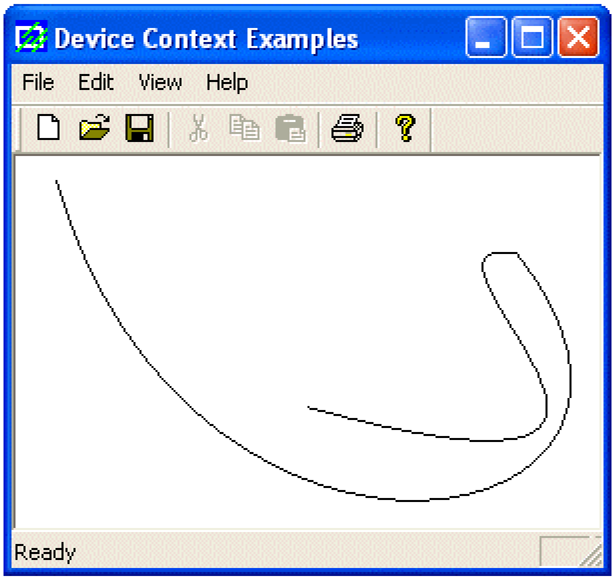
```
void CExoView::OnDraw(CDC* pDC)
{
   CPoint Pt[4] = { CPoint(20, 12), CPoint(88, 246),
                    CPoint(364, 192), CPoint(250, 48) };

   pDC->PolyBezier(Pt, 4);
}
```



In the same way, you can draw a series of complicated subsequent lines. This is done by adding reference points to the array. To do this, you must add points in increments of three. After drawing the first curve based on the first four points, to draw the next line, the function would use the fourth point as the starting point. Since the bézier line requires

4 points, you must add three more. You can continue adding points by three to draw the bézier curve. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
   CPoint Pt[] = {CPoint(20, 12), CPoint(88, 246),
                  CPoint(364, 192), CPoint(250, 48),
                  CPoint(175, 38), CPoint(388, 192), CPoint(145, 125) };

   pDC->PolyBezier(Pt, 7);
}
```

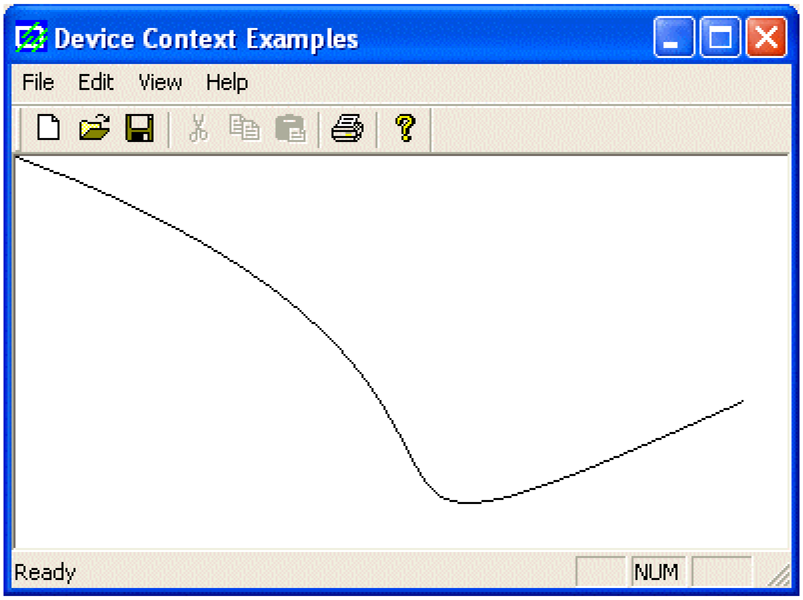


PolyBezierTo(): The CDC::PolyBezier() method requires at least four points to draw its curve. This is because it needs to know where to start drawing. Another way you can control where the curve would start is by using the CDC::PolyBezierTo() method. Its syntax is:

```
BOOL PolyBezierTo(const POINT* lpPoints, int nCount);
```

The PolyBezierTo() method draws a bézier curve. Its first argument is a pointer to an array of POINT or CPoint values. This member function requires at least three points. It starts drawing from the current line to the third point. If you do not specify the current line, it would consider the origin (0, 0). The first and the second lines are used to control the curve. The nCount argument is the number of points that would be considered. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
   CPoint Pt[] = { CPoint(320, 120), CPoint(88, 246), CPoint(364, 122) };

   pDC->PolyBezierTo(Pt, 3);
}
```


Device Context Examples
File Edit View Help
Ready
NUM

Chapter 7: GDI Accessories and Tools

✍ Colors

✍ Drawing with Colors

✍ Bitmaps

☞ Fonts

☞ Pens

☞ Brushes

7.1 Colors

7.1.1 Overview

The color is one the most fundamental objects that enhances the aesthetic appearance of an object. The color is a non-spatial object that is added to an object to modify some of its visual aspects. The MFC library, combined with the Win32 API, provides various actions you can use to take advantage of the various aspects of colors.

Three numeric values are used to create a color. Each one of these values is 8 bits. The first number is called red. The second is called green. The third is called blue:

	Bits							
Red	7	6	5	4	3	2	1	0
Green	7	6	5	4	3	2	1	0
Blue	7	6	5	4	3	2	1	0

Converted to decimal, each one of these numbers would produce:

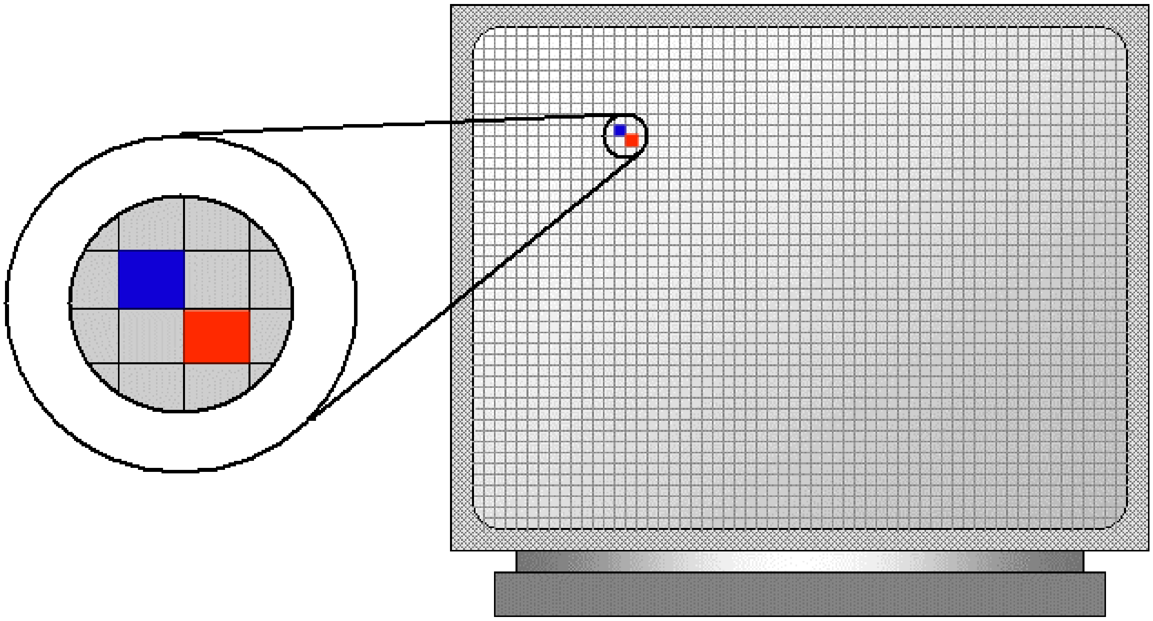
$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$$
$$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$$
$$= 255$$

Therefore, each number can have a value that ranges from 0 to 255 in the decimal system. These three numbers are combined to produce a single number as follows:

Color Value	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Blue								Green								Red							

Converted to decimal, this number has a value of 255 * 255 * 255 = 16581375. This means that we can have approximately 16 million colors available. The question that comes to mind is how we use these colors, to produce what effect.

You computer monitor has a surface that resembles a series of tinny horizontal and vertical lines. The intersection of a one horizontal line and a vertical line is called a pixel. This pixel holds, carries, or displays one color.



As the pixels close to each other have different colors, the effect is a wonderful distortion that creates an aesthetic picture. It is by changing the colors of pixels that you produce the effect of color variances seen on pictures and other graphics.

7.1.2 The Color as a Data Type

Microsoft Windows considers that a color is a 32-bit numeric value. Therefore, a color is actually a combination of 32 bits:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The bits of the most significant byte (the left byte) are reserved for the operating system's internal use and must be set to 0. Based on this, each color is characterized by its combination of a red, a green, and a blue values.

When all three numbers have their lowest value, which is 0, the color is called black. When the numbers are at their highest value, which is 255, the color is called white. Not all color combinations have a name. In fact, in MS Windows programming, the names are very limited. For this reason, color are rarely called by a name and usually, a name would depend on who is using it. Nevertheless, there are popular names that most people recognize. Examples are Black, White, Red, Green, Blue, Yellow. Except for black and white, each color can assume different variations. For example, when all three numbers have the same value but neither 0 nor 255, the color is called Gray by there are more than 250 possible combinations. Sometimes the combination is called Silver (each value is 192) or Gray (values=128).

The 32-bit numeric value used to characterize a color is defined by the COLORREF data type in Microsoft Windows programming. It can be used to declare a color value. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
    COLORREF NewColor;
}
```

When of after declaring such a variable, you can initialize it with a 32-bit decimal value. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
    COLORREF NewColor = 16711935;
}
```

Although the above number (16711935) is a legitimate color value, it does not mean much. To create a color value, the Win32 API provides the RGB macro. Its syntax is:

```
COLORREF RGB(BYTE byRed, BYTE byGreen, BYTE byBlue);
```

The RGB macro behaves like a function and allows you to pass three numeric values separated by a comma. Each value must be between 0 and 255. Therefore, the above initialization can be done as follows:

```
void CExoView::OnDraw(CDC* pDC)
{
    COLORREF NewColor = RGB(255, 0, 255);
}
```

Whether a color was initialized by a 32-bit integer or using the RGB macro, if you want to retrieve the red, green, and blue values of a color, you can use the GetRValue(), the GetGValue(), of the GetBValue() macros to extract the value of each. The syntaxes of these macros are:

```
BYTE GetRValue(DWORD rgb);
BYTE GetGValue(DWORD rgb);
BYTE GetBValue(DWORD rgb);
```

Each macro takes a 32-bit value as argument, arg. The GetRValue() macro returns the red value of the rgb number. The GetGValue() macro returns the green value of the rgb number. The GetBValue() macro returns the blue value of the rgb number.

7.1.3 Color Palettes

Device independence is the ability for an application to draw its intended figures, text, shapes, and display colors regardless of the device on which the drawing is performed. One way to take care of this is to manage colors at the operating system level so that Microsoft Windows can select the right color to render an object or portion of it. In some cases, a device, such as a monitor or a printer, may need to take care of the coloring details of the jobs it is asked to perform.

A color palette is a list of colors that a device can display. For example, one device may be able to handle only two colors; such is the case for a black and white printer. Another device could be able to use more colors than that. To control this situation, Microsoft Windows keeps track of the color palette of each device installed on the computer.

There are two types of color palettes. The default color palette is a list of colors that the operating system would use on a device unless notified otherwise. There are typically 20 reserved colors as default. A logical palette is a palette that an application creates for a specific device context.

7.2 Drawing With Colors

7.2.1 Coloring a Pixel

As mentioned above, a pixel is the real object that holds a color. Although it is so small, you can access a pixel and change its color. The pixels are stored in an array of [x][y] value. In fact, when you try accessing a pixel, you would be asked to provide a color for it. To change the color of a pixel, you can call the CDC::SetPixel() method. Its syntaxes are:

```
COLORREF SetPixel(int x, int y, COLORREF crColor);
COLORREF SetPixel(POINT point, COLORREF crColor);
```

The pixel you want to access is defined by its x and y coordinates, which can also be specified with a POINT or CPoint object as the point argument. The color you want to specify is the crColor argument.

7.2.2 Rectangles With 3-D Effect

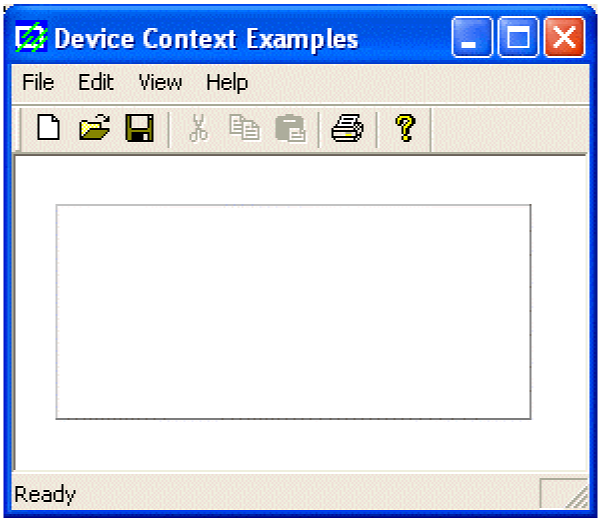
Using colors, you can draw a rectangle with a 3-D effect. To do this, the CDC class provides the Draw3dRect() method. Its syntaxes are:

```
void Draw3dRect(LPCRECT lpRect,
                   COLORREF clrTopLeft, COLORREF clrBottomRight);
void Draw3dRect(int x, int y, int cx, int cy,
                  COLORREF clrTopLeft, COLORREF clrBottomRight);
```

The rectangle to draw can be provided by its location and size through the x, y, cx, and cy arguments. You can also pass it as a pointer to RECT or CRect for the lpRect argument. Specify a color for the top and left sides of the rectangle as the clrTopLeft argument. The clrBottomRight argument holds the color of the right and bottom sides of the rectangle.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
   pDC->Draw3dRect(20, 24, 238, 108, RGB(192, 192, 192), RGB(128, 128, 128));
}
```



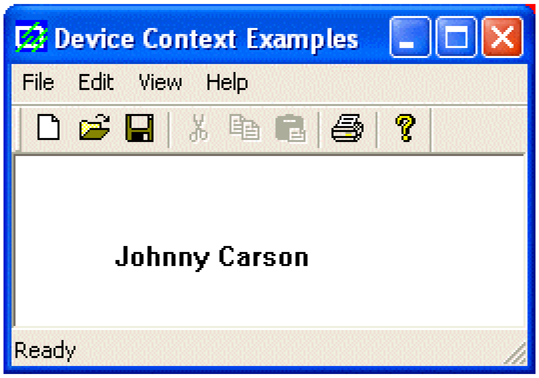
7.2.3 Drawing Text

By default, the CDC class is able to draw text using a font pre-selected, known as the System Font. To draw text, you can use the CDC::TextOut() method. Its syntax is:

```
virtual BOOL TextOut(int x, int y, LPCTSTR lpszString, int nCount);
```

To us this method, you must specify where the text would start. This location is determined from the (0, 0) origin to the right (x) and to the bottom (y). The text to display is the lpszString. The nCount value is the length of the text. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->TextOut(50, 42, "Johnny Carson", 13);
}
```



If you want to control the color used to draw the text, use the CDC::SetTextColor() method whose syntax is:

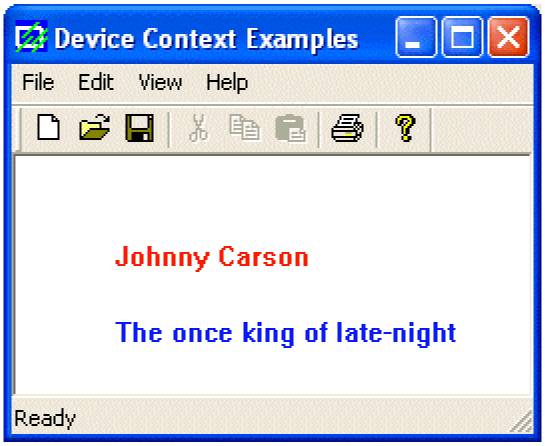
```
virtual COLORREF SetTextColor(COLORREF crColor);
```

The argument can be provided as a COLORREF variable or by calling the RGB macro. is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetTextColor(RGB(255, 25, 2));
	pDC->TextOut(50, 42, "Johnny Carson", 13);
}
```

As you will learn from now on concerning the device context, once you change one of its characteristics, it stays there until you change it again. It is similar to picking a spoon and start eating. As long as you are eating, you can use only the spoon and only that spoon. It you want to cut the meat, you must replace the spoon in your hand with a knife. In the same way, if you change the color of text and draw more than one line of text, all of them would use the same color. If you want to use a different color, you must change the color used by calling the SetTextColor() method again. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetTextColor(RGB(255, 25, 2));
	pDC->TextOut(50, 42, "Johnny Carson", 13);

	pDC->SetTextColor(RGB(12, 25, 255));
	pDC->TextOut(50, 80, "The once king of late-night", 27);
}
```



Johnny Carson

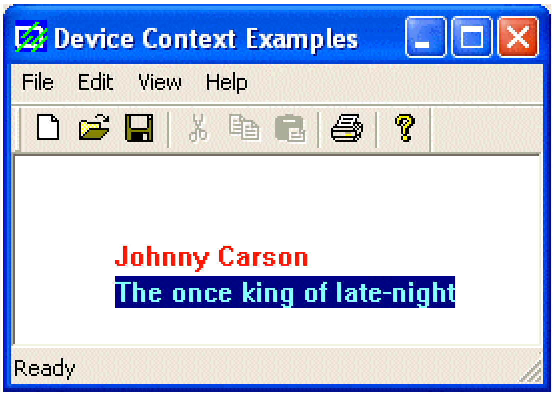
The once king of late-night

If you want to highlight the text, which is equivalent to changing its background, you can call the CDC::SetBkColor() method. Its syntax is:

```
virtual COLORREF SetBkColor(COLORREF crColor);
```

You must provide the color you want to use as the *crColor* argument. If this method succeed, it changes the background of the next text that would be drawn and it returns the previous background color, which you can restore at a later time. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CExoDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        pDC->SetTextColor(RGB(255, 25, 2));
        pDC->TextOut(50, 42, "Johnny Carson", 13);

        pDC->SetBkColor(RGB(0, 0, 128));
        pDC->SetTextColor(RGB(128, 255, 255));
        pDC->TextOut(50, 60, "The once king of late-night", 27);
}
```



If you want to know the background color applied on the object drawn, you can call the CDC::GetBkColor() method. Its syntax is:

```
COLORREF GetBkColor() const;
```

This member function returns the color used to highlight the text, if the text is highlighted. The highlighting of text is actually controlled by the CDC::SetBkMode() method whose syntax is:

```
int SetBkMode(int nBkMode);
```

This method specifies whether the background color should be applied or not. This is set by the nBkMode argument. It can have one of two values. If it is:

OPAQUE: the background would be drawn using the *crColor* value
TRANSPARENT: the background would not be drawn

If you want to find out what background mode is applied to the object(s) drawn, you can call the CDC::GetBkMode() method. It is declared as follows:

```
int GetBkMode() const;
```

You can also draw text and include it in a (colored) rectangle. This can be done using the CDC::ExtTextOut() method. Its syntax is:

```
virtual BOOL ExtTextOut(int x, int y, UINT nOptions, LPCRECT lpRect,
    LPCTSTR lpszString, UINT nCount, LPINT lpDxWidths);
```

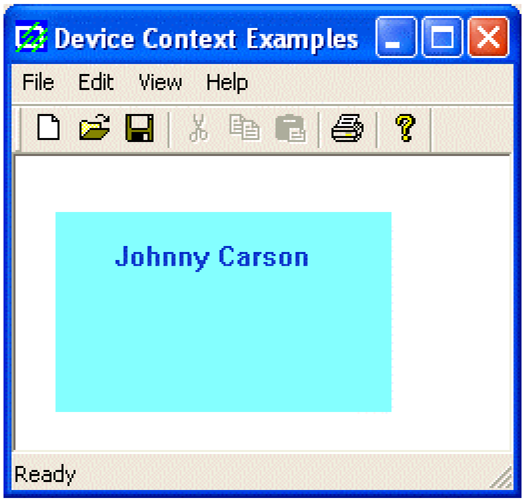
The x and y values specify the location of the first character of the text to be drawn.
The nOptions argument holds a constant that determines how the rectangle will be drawn. It can be:

ETO_OPAQUE: in this case the color set by SetBkColor() would be used to fill the rectangle
ETO_CLIPPED: the color previously specified by SetBkColor() will only highlight the text

The lpRect is a RECT or CRect rectangle that will be drawn behind the text.
The lpszString value is the text to be drawn.
The nCount is the number of characters of lpszString.
The lpDxWidths argument is an array of integers that specifies the amount of empty spaces that will be used between each combination of two characters. Unless you know what you are doing, set this argument as 0, in which case the regular space used to separate characters would be used.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
   CExoDoc* pDoc = GetDocument();
   ASSERT_VALID(pDoc);

   pDC->SetTextColor(RGB(25, 55, 200));
   pDC->SetBkColor(RGB(128, 255, 255));
   pDC->ExtTextOut(50, 42, ETO_OPAQUE, CRect(20, 28, 188, 128), "Johnny Carson",
13, NULL);
}
```



7.3 Bitmaps

7.3.1 Introduction

A bitmap is a series of points (bits) arranged like a map so that, when put together, they produce a picture that can be written to, copied from, re-arranged, changed, manipulated, or stored as a a computer file. Bitmaps are used to display pictures on graphical applications, word processors, database files, or audience presentations. To display its product on a device such as a monitor or a printer, a bitmap holds some properties and follows a set of rules.

There are various types of bitmap, based on the number of colors that the bitmap can display. First of all, a bitmap can be monochrome in which case each pixel corresponds to 1 bit. A bitmap can also be colored. The number of colors that a bitmap can display is equal to 2 raised to the number of pits/pixel. For example, a simple bitmap uses only 4 pits/pixel or 4 bpp can handle only 24 = 16 colors. A more enhanced bitmap that requires 8 bpp can handle 28 = 256 colors. Bitmaps are divided in two categories that control their availability to display on a device.

A device-independent bitmap (DIB) is a bitmap that is designed to be loaded on any application or display on any device and produce the same visual effect. To make this possible, such a bitmap contains a table of colors that describes how the colors of the bitmap should be used on pixels when displaying it. The characteristics of a DIB are defined by the BITMAPINFO structure.

A device-dependent bitmap (DDB) is a bitmap created from the BITMAP structure the dimensions of the bitmap.

7.3.2 Bitmap Creation

Unlike the other GDI tools, creating a bitmap usually involves more steps. For example, you may want to create a bitmap to display on a window. You may create another bitmap to paint a geometric area, in which case the bitmap would be used as a brush.

Before creating a bitmap as a GDI object, you should first have a bitmap. You can do this by defining an array of unsigned hexadecimal numbers. Such a bitmap can be used for a brush.

To create and manipulate bitmaps, the MFC library provides the CBitmap class. The use of this class depends on the type of bitmap you want to create and how you want to use that bitmap. One way you can use a bitmap is to display a picture on a window. To do this, you must first have a picture resource. Although the Image Editor built-in Microsoft Visual C++ is meant to help with regular application resources, it has a problem handling a bitmap that displays more than 16 colors. The remedy used is to import the bitmap you want to use. Once your bitmap is ready, call the CBitmap::LoadBitmap() method. Its syntaxes:

```
BOOL LoadBitmap(UINT nIDResource);
BOOL LoadBitmap(LPCTSTR lpszResourceName);
```

The first version takes, as argument, the identifier of the bitmap you want to use. If the bitmap is recognized by its name, you can use the second version of this method and provide the lpszResourceName argument.

Before selecting the newly created bitmap object, allocate a block of computer memory that would hold the bitmap and can then copy it to the actual device. This job can be taken care of by the CDC::CreateCompatibleDC() method. Its syntax is:

```
virtual BOOL CreateCompatibleDC(CDC* pDC);
```

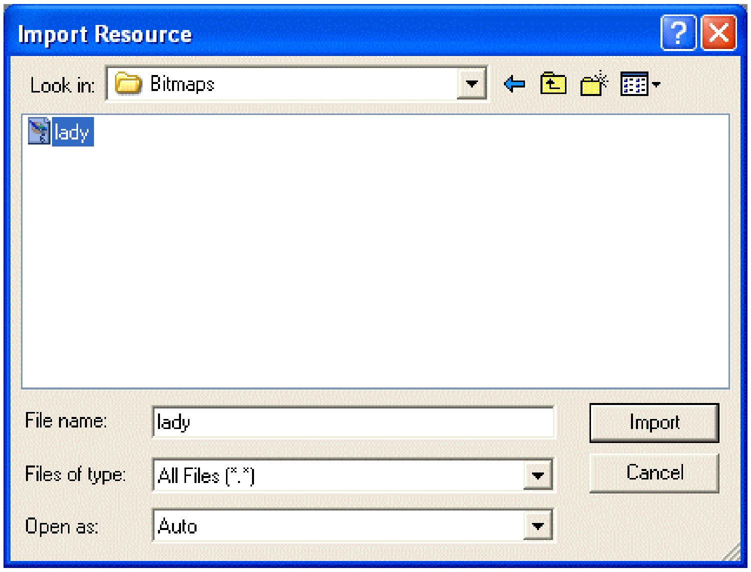
This method takes a pointer to a device context. If it is successful, it returns TRUE or a non-zero value. If it is not, it returns FALSE or 0.

Practical Learning: Loading a Bitmap

1. Start a new project and name it Bitmap1
2. Create it as a Single Document application based on CView
3. In the Class View, expand everything and access the CMainFrame::PreCreateWindow() method
4. Change its code as follows:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreateWindow(cs) )
		return FALSE;

	// The new width of the window's frame
	cs.cx = 480;
	// The new height of the window's frame
	cs.cy = 490;
	// Remove the Untitled thing
	cs.style &= ~FWS_ADDTOTITLE;

	return TRUE;
}
```

5. In the Resource View, display the string table and change the IDR_MAINFRAME Caption to
Lady on Phone\n\nBitmap\n\n\nBitmap1.Document\nBitmap Document
6. Right-click any folder and click Import...
7. In the Import Resource dialog box, change the Files of Type to All Files and, in the Look In combo box, change the folder to the one that holds the accompanying exercises for this book.
8. Select lady.bmp



Import Resource

Look in: Bitmaps

lady

File name: lady

Files of type: All Files (*.*)

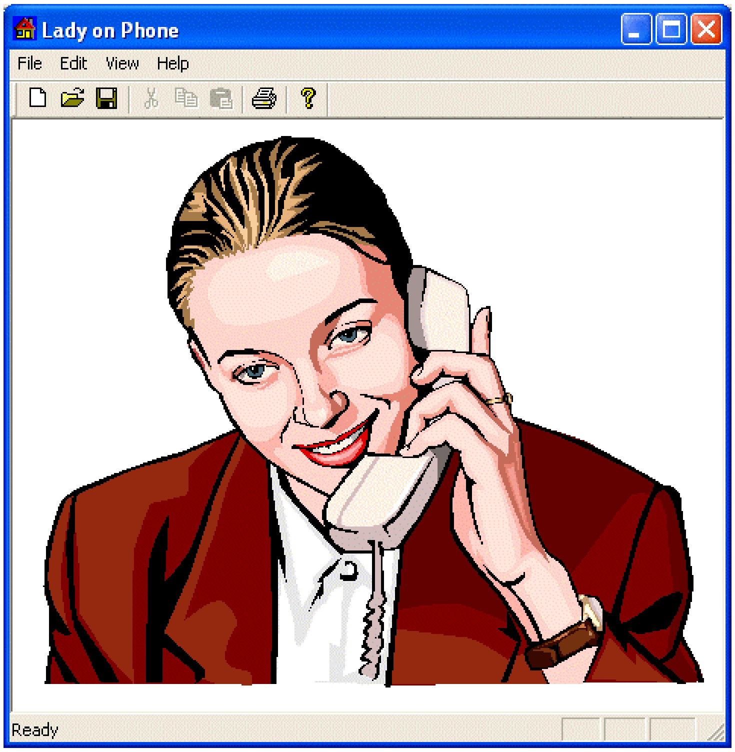
Open as: Auto

9. Click Import. After the bitmap has been imported, you may receive a message box, click OK
10. Right-click the new IDB_BITMAP1 resource and click Properties

11. Change its ID to IDB_LADY
12. Add a message handler of the WM_PAINT message for the CBitmap1View class and implement it as follows:

```
void CBitmap1View::OnPaint()
{
	CPaintDC dc(this); // device context for painting

	// TODO: Add your message handler code here
	CBitmap BmpLady;
	CDC MemDCLady;

	// Load the bitmap from the resource
	BmpLady.LoadBitmap(IDB_LADY);
	// Create a memory device compatible with the above CPaintDC variable
	MemDCLady.CreateCompatibleDC(&dc);
	// Select the new bitmap
	CBitmap *BmpPrevious = MemDCLady.SelectObject(&BmpLady);

	// Copy the bits from the memory DC into the current dc
	dc.BitBlt(20, 10, 436, 364, &MemDCLady, 0, 0, SRCCOPY);

	// Restore the old bitmap
	dc.SelectObject(BmpPrevious);
	// Do not call CView::OnPaint() for painting messages
}
```

13. Test the application



14. Return to MSVC

7.4 Fonts

7.4.1 Introduction

A font is a list of symbols that can be drawn on a device context to produce a message. A font is designed by an artist but usually follows a specific pattern. For example a font designed to produce symbols readable in the English language must be designed by a set of predetermined and agreed upon symbols. These English symbols are grouped in an entity called the English alphabet. When designing such a font, the symbols created must conform to that language. This also implies that one font can be significantly different from another and a font is not necessarily a series of readable symbols.

Just like everything else in the computer, a font must have a name. To accommodate the visual needs, a font is designed to assume different sizes.

7.4.2 Font Selection

Before using a font to draw text in a device, the font must have been installed. Microsoft Windows installs many fonts during setup. To handle its various assignments, the operating system uses a particular font known as the System Font. This is the font used to display the menu items and other labels for resources in applications. If you want to use a different font to draw text in your application, you must select it.

Selecting a font, as well as selecting any other GDI object we will use from now on, is equivalent to specifying the characteristics of a GDI object you want to use. To do this, you must first create the object, unless it exists already. To select an object, pass it as a pointer to the CDC::SelectObject() method. For example, to select a font, the syntax you would use is:

```
virtual CFont* SelectObject(CFont* pFont);
```

This method takes as argument the font you want to use, pFont. It returns a pointer to the font that was previously selected. If there was a problem when selecting the font, the method returns NULL. As you can see, you must first have a font you want to select.

7.4.3 Font Creation

A font is created as a variable of the CFont class (of course, you can also use the Win32 API's HFONT class). The CFont class is based on CGdiObject. To declare a CFont variable, you can use the default constructor of this class. This can be easily done as follows:

```
CFont NewFont;
```

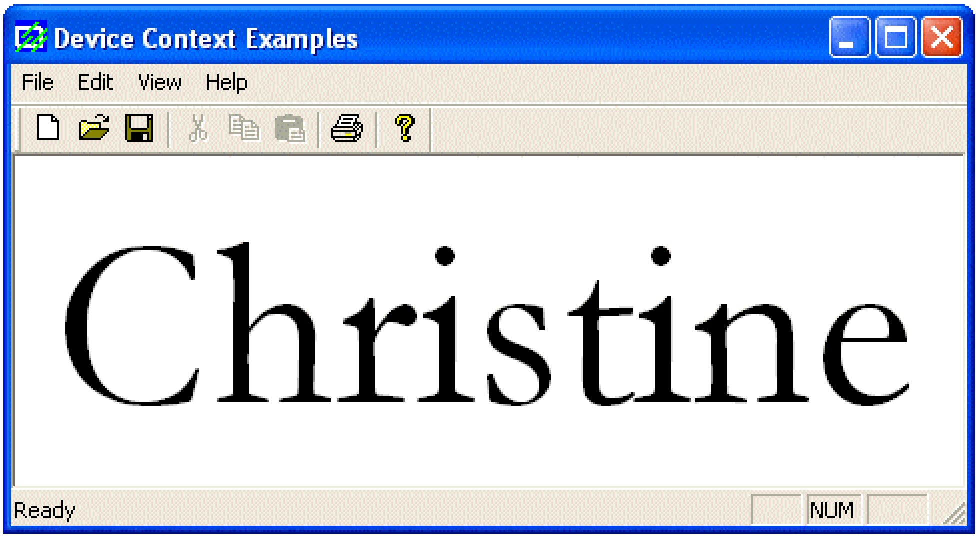
After declaring a CFont variable, you must initialize it. This can be done by calling one of the *Create* member functions. The easiest technique of creating a font is done with the **CreatePointFont()** method. Its syntax is:

```
BOOL CreatePointFont(int nPointSize, LPCTSTR lpszFaceName, CDC* pDC =
NULL);
```

The *nPointSize* is the height of the font. It is supplied as a multiple of 1/10.

This method requires two value. The name of the font is specified with the lpszFaceName value. If you do not want to specify a font, you can pass the argument as NULL. If you have a CDC variable that can be used convert the value of the height to logical units. If you do not have this value, set it to NULL.
Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CFont font;

	font.CreatePointFont(920, "Garamond");
	CFont *pFont = pDC->SelectObject(&font);
	pDC->TextOut(20, 18, "Christine", 9);

	pDC->SelectObject(pFont);
	font.DeleteObject();
}
```



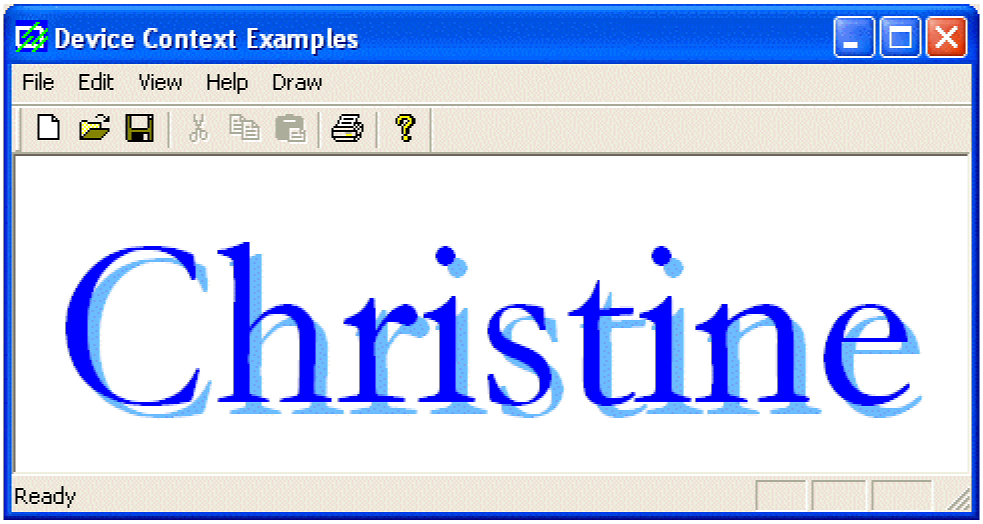
To control the color applied when drawing the text, you can call the CDC::SetTextColor() method. For example, the above name can be drawn in blue as follows:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CFont font;

	font.CreatePointFont(920, "Garamond");
	CFont *pFont = pDC->SelectObject(&font);

	pDC->SetTextColor(RGB(0, 125, 250));
	pDC->TextOut(20, 18, "Christine", 9);

	pDC->SelectObject(pFont);
```

```
            font.DeleteObject();
}
```

To produce a shadow effect, you can add another copy of the same text on a slightly different location and call the CDC::SetBkMode() method. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
            CExoDoc* pDoc = GetDocument();
            ASSERT_VALID(pDoc);

            CFont font;

            font.CreatePointFont(920, "Garamond");
            CFont *pFont = pDC->SelectObject(&font);

            pDC->SetBkMode(TRANSPARENT);

            pDC->SetTextColor(RGB(110, 185, 250));
            pDC->TextOut(26, 24, "Christine", 9);

            pDC->SetTextColor(RGB(0, 0, 255));
            pDC->TextOut(20, 18, "Christine", 9);

            pDC->SelectObject(pFont);
            font.DeleteObject();
}
```



One of the most complete means of creating a font is by using the CFont::CreateFont() method. Its syntax is:

```
BOOL CreateFont(int nHeight,
                 int nWidth,
                 int nEscapement,
                 int nOrientation,
                 int nWeight,
                BYTE bItalic,
                BYTE bUnderline,
                 BYTE cStrikeOut,
                BYTE nCharSet,
                 BYTE nOutPrecision,
                BYTE nClipPrecision,
                 BYTE nQuality,
                BYTE nPitchAndFamily,
                LPCTSTR lpszFacename);
```

The nHeight argument is the height applied to the text.

The nWidth value is the desired width that will be applied on the text.

The nEscapement is the angle used to orient the text. The angle is calculated as a multiple of 0.1 and oriented counterclockwise.

The nOrientation is the angular orientation of the text with regards to the horizontal axis.

The nWeight is used to attempt to control the font weight of the text because it is affected by the characteristics of the font as set by the designer. It holds values that displays text from thin heavy bold. The possible values are:

Constant	Value		Constant	Value
FW_DONTCARE	0		FW_THIN	100
FW_EXTRALIGHT	200		FW_ULTRALIGHT	200
FW_LIGHT	300			
FW_NORMAL	400		FW_REGULAR	400
FW_MEDIUM	500			
FW_SEMIBOLD	600		FW_DEMIBOLD	600
FW_BOLD	700			
FW_EXTRABOLD	800		FW_ULTRABOLD	800
FW_BLACK	900		FW_HEAVY	900

The bItalic specifies whether the font will be italicized (TRUE) or not (FALSE).

The bUnderline is used to underline (TRUE) or not underline (FALSE) the text.

The cStrikeOut is specifies whether the text should be stroke out (TRUE) or not (FALSE) with a line.

The nCharSet, specifies the character set used. The possible values are:

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

The nOutPrecision controls the amount precision used to evaluate the numeric values used on this function for the height, the width, and angles. It can have one of the following values: OUT_CHARACTER_PRECIS, OUT_STRING_PRECIS, OUT_DEFAULT_PRECIS, OUT_STROKE_PRECIS, OUT_DEVICE_PRECIS, OUT_TT_PRECIS, OUT_RASTER_PRECIS

If some characters may be drawn outside of the area in which they are intended, the nClipPrecision is used to specify how they may be clipped. The possible value used are CLIP_CHARACTER_PRECIS, CLIP_MASK, CLIP_DEFAULT_PRECIS, CLIP_STROKE_PRECIS, CLIP_ENCAPSULATE, CLIP_TT_ALWAYS, CLIP_LH_ANGLES.

The nQuality specifies how the function will attempt to match the font's characteristics. The possible values are DEFAULT_QUALITY, PROOF_QUALITY, and DRAFT_QUALITY.
The nPitchAndFamily specifies the category of the font used. It combines the pitch and the family the intended font belongs to. The pitch can be specified with DEFAULT_PITCH, VARIABLE_PITCH, or FIXED_PITCH. The pitch is combined using the bitwise OR operator with one of the following values:

Value	Description
FF_DECORATIVE	Used for a decorative or fancy font
FF_DONTCARE	Let the compiler specify
FF_MODERN	Modern fonts that have a constant width
FF_ROMAN	Serif fonts with variable width
FF_SCRIPT	Script-like fonts
FF_SWISS	Sans serif fonts with variable width

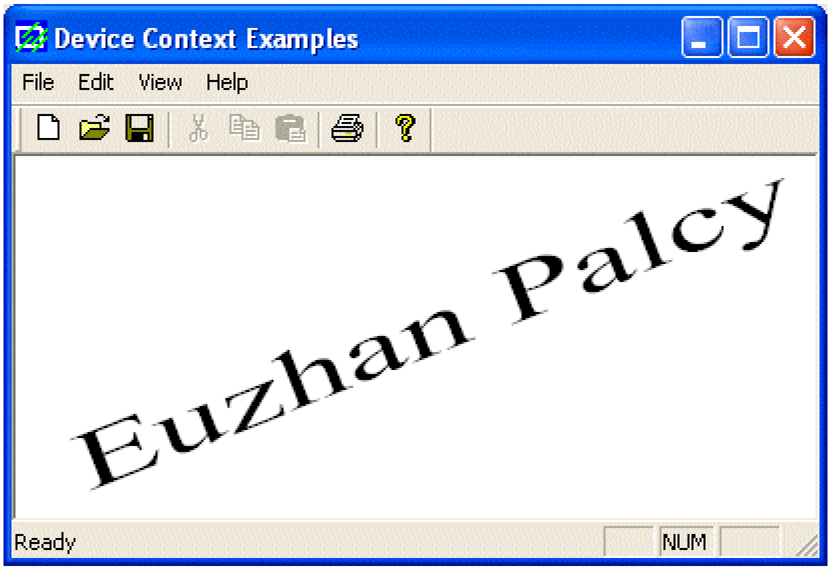
The lpszFacename is the name of the font used.

Once you have created a font, you can select it into the device context and use it it for example to draw text.

After using a font, you should delete it to reclaim the memory space its variable was using. This is done by calling the CGdiObject::DeleteObject() method.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
   CFont font;

   font.CreateFont(46, 28, 215, 0,
                 FW_NORMAL, FALSE, FALSE, FALSE, ANSI_CHARSET,
                    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_ROMAN, "Times New Roman");

   CFont *pFont = pDC->SelectObject(&font);
   pDC->TextOut(20, 128, "Euzhan Palcy", 12);

   pDC->SelectObject(pFont);
   font.DeleteObject();
}
```



Remember that once an object such as a font has been selected, it remains in the device context until further notice. For example, if you have created and selected a font, any text you draw would follow the characteristics of that font. If you want another font, you must change the previously selected font.

The computer uses the default black color to draw the text. Once again, if you want to draw text with a different color, you can first call the CDC::SetTextColor() method and specify the color of your choice.

The CFont::CreateFont() method is used to specify all characteristics of a font in one step. Alternatively, if you want to specify each font property, you can declare a LOGFONT variable and initialize it. It is defined as follows:

```
typedef struct tagLOGFONT {
        LONG lfHeight;
        LONG lfWidth;
        LONG lfEscapement;
        LONG lfOrientation;
        LONG lfWeight;
        BYTE lfItalic;
        BYTE lfUnderline;
        BYTE lfStrikeOut;
        BYTE lfCharSet;
        BYTE lfOutPrecision;
        BYTE lfClipPrecision;
        BYTE lfQuality;
        BYTE lfPitchAndFamily;
        TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT, *PLOGFONT;
```

This time, you do not have to provide a value for each member of the structure and even if you do, you can supply values in the order of your choice. For any member whose value is not specified, the compiler would use a default value but you may not like some of the default values. Therefore, you should specify as many values as possible.

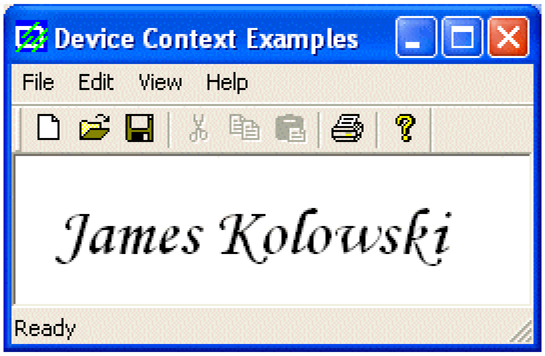
After initializing the LOGFONT variable, call the CFont::CreateFontIndirect() method. Its syntax is:

```
BOOL CreateFontIndirect(const LOGFONT* lpLogFont);
```

When calling this member function, pass the LOGFONT variable as a pointer, lpLogFont.

To select the selected font, call the CDC::SelectObject() method. Once done, you can use the new font as you see fit. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CFont font;
	LOGFONT LogFont;

	LogFont.lfStrikeOut = 0;
	LogFont.lfUnderline = 0;
	LogFont.lfHeight = 42;
	LogFont.lfEscapement = 0;
	LogFont.lfItalic = TRUE;

	font.CreateFontIndirect(&LogFont);
	CFont *pFont = pDC->SelectObject(&font);
	pDC->TextOut(20, 18, "James Kolowski", 14);

	pDC->SelectObject(pFont);
	font.DeleteObject();
}
```



7.4.4 Font Retrieval

If some text is displaying and you want to get the font properties of that text, you can call the CDC::GetLogFont() method. Its syntax is:

```
int GetLogFont(LOGFONT * pLogFont);
```

To get the current font characteristics on a device context, pass a LOGFONT variable as pointer to this method. After execution, it returns the LOGFONT argument with these characteristics. If this method succeeds, it returns TRUE or non-zero. It it fails, it returns FALSE or 0.

Here is an example:

```
void CAboutDlg::OnButton1()
{
	CFont *font;
```

```
        LOGFONT LogFont;

        font = this->GetFont();
        font->GetLogFont(&LogFont);
        char StrFont[40];

        strcpy(StrFont, LogFont.lfFaceName);

        CClientDC dc(this);
        dc.TextOut(58, 120, StrFont, strlen(StrFont));
}
```

7.5 Pens

7.5.1 Introduction

In the previous lesson, we mentioned that, in order to draw, two primary objects are needed: a platform and a tool. So far, we were using the platform, called a device context. We introduced the main device context class as the CDC class. To draw, we have been using a pointer to CDC. Now, we need to realize that, declaring a CDC variable does not just give us access to the device context, it also initializes it.

The device context is a combination of the platform on which the drawing is performed and the necessary tools to draw on it. As such, when declaring a CDC variable, it also creates and selects a black pen. This is why we have been able to draw lines and other shapes so far.

7.5.2 The Fundamentals of a Pen

A pen is a tool used to draw lines and curves on a device context. In the graphics programming, a pen is also used to draw the borders of a geometric closed shape such as a rectangle or a polygon.

To make it an efficient tool, a pen must produce some characteristics on the lines it is asked to draw. These characteristics can range from the width of the line drawn to their colors, from the pattern applied to the level of visibility of the lines. To manage these properties, Microsoft Windows considers two types of pens: cosmetic and geometric.

A pen is referred to as cosmetic when it can be used to draw only simple lines of a fixed width, less than or equal to 1 pixel.

A pen is geometric when it can assume different widths and various ends.

7.5.3 Creating and Selecting a Pen

When you declare a CDC variable, it creates and selects a pen that can draw a 1-pixel width black line. If you want a more refined pen, the MFC provides the CPen class. Therefore, the first step in creating a pen is to declare a variable of CPen type, which can be done using the default constructor as follows:

```
CPen NewPen;
```

To create a pen, you must specify the desired characteristics. This can be done with another CPen constructor declared as follows:

```
CPen(int nPenStyle, int nWidth, COLORREF crColor);
```

Alternatively, if you want to use a variable declared using the default constructor, you can then call the CPen::CreatePen() method. Its syntax is:

```
BOOL CreatePen(int nPenStyle, int nWidth, COLORREF crColor);
```

The arguments of the second constructor and the CreatePen() method are used to specify the properties that the pen should have:

The **Style**: This characteristic is passed as the nPenStyle argument. The possible values of this argument are:

Value	Illustration	Description
PS_SOLID	———————	A continuous solid line
PS_DASH	·– – – – – – – – – – –	A continuous line with dashed interruptions
PS_DOT	······························	A line with a dot interruption at every other pixel
PS_DASHDOT	·–·–·–·–·–·–·–·–·–	A combination of alternating dashed and dotted points
PS_DASHDOTDOT	—··—··—··—··—··	A combination of dash and double dotted interruptions
PS_NULL		No visible line
PS_INSIDEFRAME	———————	A line drawn just inside of the border of a closed shape

To specify the type of pen you are creating, as cosmetic or geometric, use the bitwise OR operator to combine one of the above styles with one of the following:

?? **PS_COSMETIC**: used to create a cosmetic pen

?? **PS_GEOMTERIC**: used to create a geometric pen

If you are creating a cosmetic pen, you can also add (bitwise OR) the **PS_ALTERNATE** style to to set the pen at every other pixel.

The **Width**: The nWidth argument is the width used to draw the lines or borders of a closed shape. A cosmetic pen can have a width of only 1 pixel. If you specify a higher width, it would be ignored. A geometric pen can have a width of 1 or more pixels but the line can only be solid or null. This means that, if you specify the style as **PS_DASH**, **PS_DOT**, **PS_DASHDOT**, or **PS_DASHDOTDOT** but set a width higher than 1, the line would be drawn as **PS_SOLID**.

The **Color**: The default color of pen on the device context is black. If you want to control the color, specify the desired value for the crColor argument.

Based this, using the second constructor, you can declare and initialize a CPen variable as follows:

```
CPen NewPen(PS_DASHDOTDOT, 1, RGB(255, 25, 5));
```

The same pen can be created using the CreatePen() method as follows:

```
CPen NewPen;

NewPen.CreatePen(PS_DASHDOTDOT, 1, RGB(255, 25, 5));
```

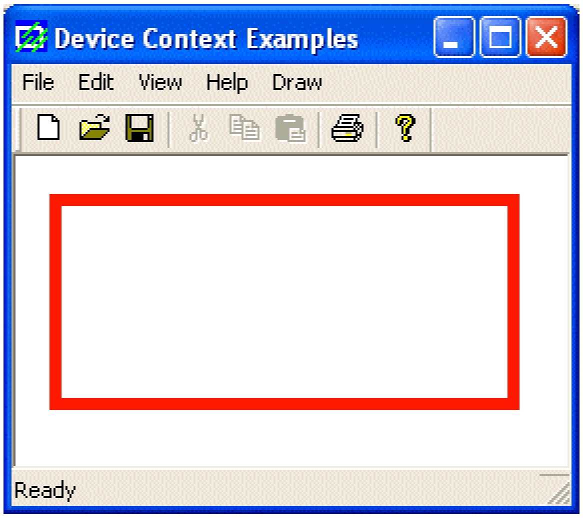
After creating a pen, you can select it into the desired device context variable and then use it as you see fit, such as drawing a rectangle. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen NewPen;

	NewPen.CreatePen(PS_DASHDOTDOT, 1, RGB(255, 25, 5));

	pDC->SelectObject(&NewPen);

	pDC->Rectangle(20, 22, 250, 125);
}
```

Once a pen has been selected, any drawing performed and that uses a pen would use the currently selected pen. If you want to use a different pen, you can either create a new pen or change the characteristics of the current pen.

After using a pen, between exiting the function or event that created it, you should get rid of it and restore the pen that was selected previously. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen NewPen;

	NewPen.CreatePen(PS_DASHDOTDOT, 6, RGB(255, 25, 5));

	CPen* pPen = pDC->SelectObject(&NewPen);

	pDC->Rectangle(20, 22, 250, 125);

	// Restore the previous pen
	pDC->SelectObject(pPen);
}
```



The Win32 API provides the LOGPEN structure that you can use to individually specify each characteristics of a pen. The LOGPEN structure is created as follows:

```
typedef struct tagLOGPEN {
    UINT      lopnStyle;
    POINT     lopnWidth;
    COLORREF lopnColor;
} LOGPEN, *PLOGPEN;
```

To use this structure, declare a variable of LOGPEN type or a pointer. Then initialize each member of the structure. If you do not, its default values would be used and the line not be visible.

The lopnStyle argument follows the same rules we reviewed for the nPenStyle argument of the second constructor and the CreatePen() method.

The lopnWidth argument is provided as a POINT or a CPoint value. Only the POINT::x or the CPoint::x value is considered.

The lopnColor argument is a color and can be provided following the rules we reviewed for colors.

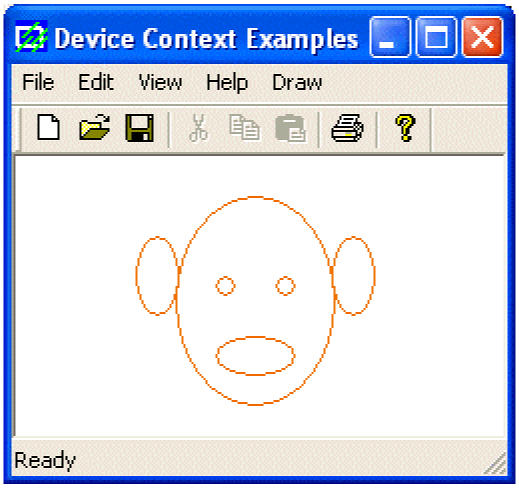
After initializing the LOGPEN variable, call the CPen::CreatePenIndirect() member function to create a pen. The syntax of the CreatePenIndirect() method is:

```
BOOL CreatePenIndirect(LPLOGPEN lpLogPen);
```

The LOGPEN value is passed to this method as a pointer. After this call, the new pen is available and can be selected into a device context variable for use. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen NewPen;
	LOGPEN LogPen;

	LogPen.lopnStyle = PS_SOLID;
```

```
        LogPen.lopnWidth = CPoint(1, 105);
        LogPen.lopnColor = RGB(235, 115, 5);

        NewPen.CreatePenIndirect(&LogPen);

        CPen* pPen = pDC->SelectObject(&NewPen);

        pDC->Ellipse(60, 40, 82, 80);
        pDC->Ellipse(80, 20, 160, 125);
        pDC->Ellipse(158, 40, 180, 80);

        pDC->Ellipse(100, 60, 110, 70);
        pDC->Ellipse(130, 60, 140, 70);
        pDC->Ellipse(100, 90, 140, 110);

        // Restore the previous pen
        pDC->SelectObject(pPen);
}
```



7.5.4 Retrieving a Pen

If you want to know the currently selected pen used on a device context, you can call the CPen::GetLogPen() member function. Its syntax is:

```
int GetLogPen(LOGPEN* pLogPen);
```

To get the characteristics of the current pen, pass a pointer to the LOGPEN structure to this GetLogPen() method. The returned pLogPen value would give you the style, the width, and the color of the pen.

7.6 Brushes

7.6.1 Introduction

A brush is a drawing tool used to fill out closed shaped or the interior of lines. A brush behaves like picking up a bucket of paint and pouring it somewhere. In the case of computer graphics, the area where you position the brush is called the brush origin. The color (or picture) that the brush holds would be used to fill the whole area until the brush finds a limit set by some rule.

A brush can be characterized by its color (if used), its pattern used to fill the area, or a picture (bitmap) used as the brush.

To create a brush, the MFC provides the CBrush class. Therefore, to start, you can declare a variable of this type using the default constructor as follows:

```
CBrush NewBrush;
```

Because there can be so many variations of brushes, there are different member functions for the various possible types of brushes you would need. The easiest brush you can create is made of a color.

7.6.2 Solid Brushes

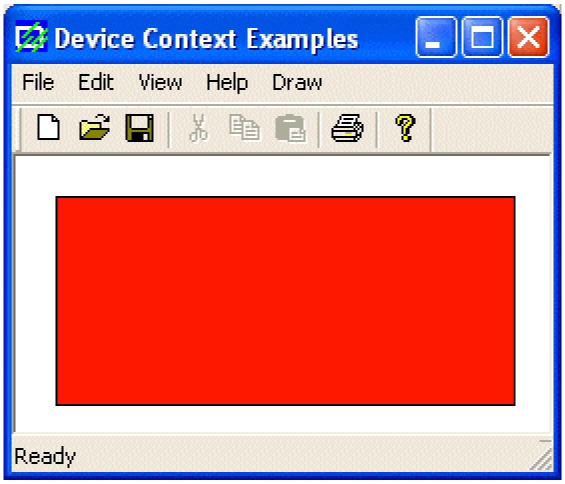
A brush is referred to as solid if it is made of a color simply used to fill a closed shaped. To create a solid brush, you can use the following constructor:

```
CBrush(COLORREF crColor);
```

The color to provide as the crColor argument follows the rules we reviewed for colors.

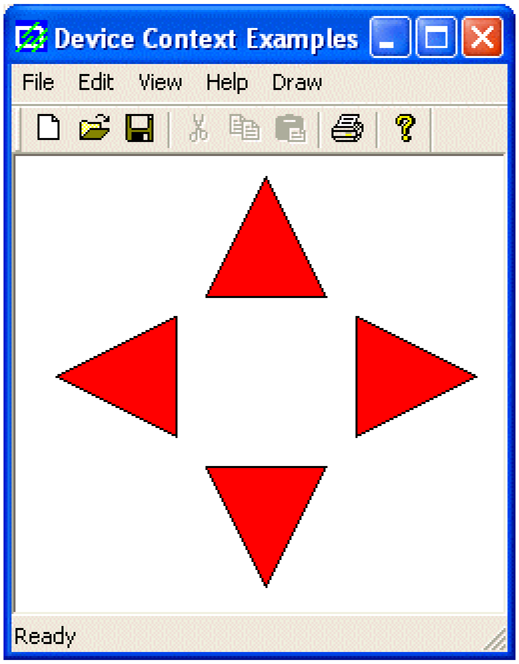
To use the newly created brush, you can select it into the device context by calling the CDC::SelectObject(). Once this is done. Any closed shape you draw (ellipse, rectangle, polygon) would be filled with the color specified. After using the brush, you can dismiss it and restore the previous brush. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush NewBrush(RGB(250, 25, 5));

	CBrush *pBrush = pDC->SelectObject(&NewBrush);
	pDC->Rectangle(20, 20, 250, 125);

	pDC->SelectObject(pBrush);
}
```



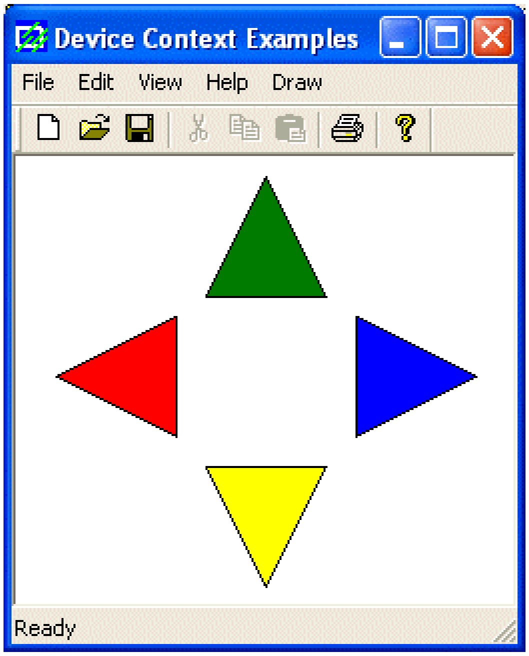
Once a brush has been selected, it would be used on all shapes that are drawn under it, until you delete or change it. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush NewBrush(RGB(255, 2, 5));
	CBrush *pBrush;

	CPoint Pt[3];

	// Top Triangle
	Pt[0] = CPoint(125,  10);
	Pt[1] = CPoint( 95,  70);
	Pt[2] = CPoint(155,  70);

	pBrush = pDC->SelectObject(&NewBrush);
	pDC->Polygon(Pt, 3);

	// Left Triangle
	Pt[0] = CPoint( 80,  80);
	Pt[1] = CPoint( 20, 110);
	Pt[2] = CPoint( 80, 140);

	pDC->Polygon(Pt, 3);

	// Bottom Triangle
	Pt[0] = CPoint( 95, 155);
	Pt[1] = CPoint(125, 215);
	Pt[2] = CPoint(155, 155);

	pDC->Polygon(Pt, 3);

	// Right Triangle
	Pt[0] = CPoint(170,  80);
	Pt[1] = CPoint(170, 140);
	Pt[2] = CPoint(230, 110);

	pDC->Polygon(Pt, 3);
	pDC->SelectObject(pBrush);
}
```



If you want to use a different brush, you should create a new one. Here is an examp le:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush BrushGreen(RGB(0, 125, 5));
	CBrush BrushRed(RGB(255, 2, 5));
	CBrush BrushYellow(RGB(250, 255, 5));
	CBrush BrushBlue(RGB(0, 2, 255));
	CBrush *pBrush;

	CPoint Pt[3];

	// Top Triangle
	Pt[0] = CPoint(125,  10);
	Pt[1] = CPoint( 95,  70);
	Pt[2] = CPoint(155,  70);

	pBrush = pDC->SelectObject(&BrushGreen);
	pDC->Polygon(Pt, 3);

	// Left Triangle
	Pt[0] = CPoint( 80,  80);
	Pt[1] = CPoint( 20, 110);
	Pt[2] = CPoint( 80, 140);

	pBrush = pDC->SelectObject(&BrushRed);
	pDC->Polygon(Pt, 3);
	// Bottom Triangle
	Pt[0] = CPoint( 95, 155);
	Pt[1] = CPoint(125, 215);
	Pt[2] = CPoint(155, 155);

	pBrush = pDC->SelectObject(&BrushYellow);
	pDC->Polygon(Pt, 3);

	// Right Triangle
	Pt[0] = CPoint(170,  80);
	Pt[1] = CPoint(170, 140);
	Pt[2] = CPoint(230, 110);

	pBrush = pDC->SelectObject(&BrushBlue);
	pDC->Polygon(Pt, 3);

	pDC->SelectObject(pBrush);
}
```

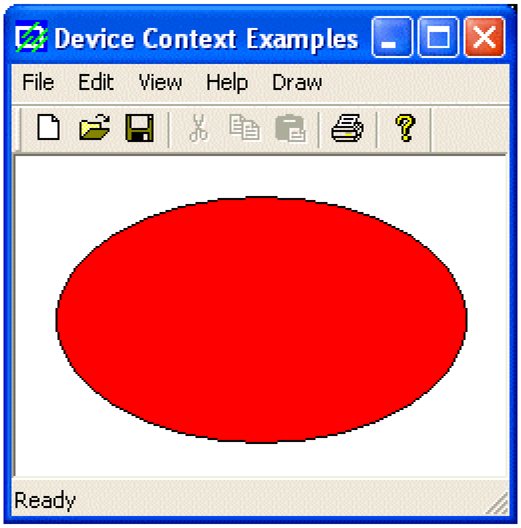


If you had declared a CBrush variable using the default constructor, you can initialize it with a color by calling the CBrush::CreateSolidBrush() method. Its syntax is:

```
BOOL CreateSolidBrush(COLORREF crColor);
```

This member function can be used in place of the second constructor. It takes the same type of argument, a color as crColor and produces the same result. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush BrushOlive;
	CBrush *pBrush;
	BrushOlive.CreateSolidBrush(RGB(255, 2, 5));
	pBrush = pDC->SelectObject(&BrushOlive);

	pDC->Ellipse(20, 20, 226, 144);
	pDC->SelectObject(pBrush);
}
```



7.6.3 Hatched Brushes

A hatch brush is one that uses a drawn pattern to regularly fill an area. Microsoft Windows provides 6 preset patterns for such a brush. To create a hatched brush, you can use the following constructor:

```
CBrush(int nIndex, COLORREF crColor);
```

If you had declared a CBrush variable using the default constructor, you can call the CreateHatchBrush() member function to initialize it. The syntax of this method is:

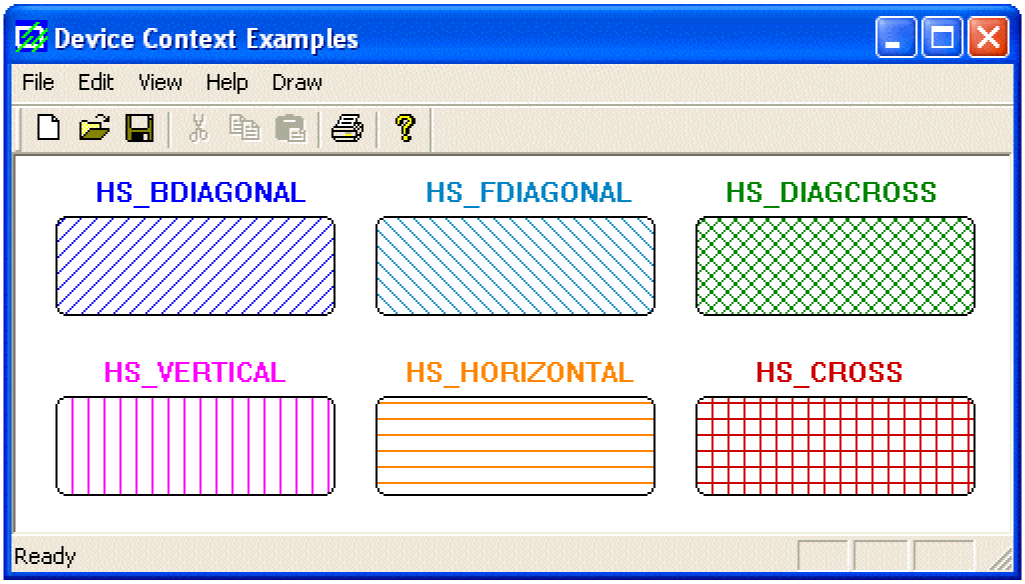
```
BOOL CreateHatchBrush(int nIndex, COLORREF crColor);
```

In both cases, the nIndex argument specifies the hatch pattern that must be used to fill the area. The possible values to use are HS_BDIAGONAL, HS_CROSS, HS_DIAGCROSS, HS_FDIAGONAL, HS_HORIZONTAL, or HS_VERTICAL.

The crColor argument specifies the color applied on the drawn pattern.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush brBDiagonal(HS_BDIAGONAL, RGB(0, 0, 255));
	CBrush brCross;
	CBrush brDiagCross(HS_DIAGCROSS, RGB(0, 128, 0));
	CBrush brFDiagonal;
	CBrush brHorizontal(HS_HORIZONTAL, RGB(255, 128, 0));
	CBrush brVertical;

	CBrush *pBrush;
	pBrush = pDC->SelectObject(&brBDiagonal);
	pDC->RoundRect( 20,  30, 160, 80, 10, 10);

	brFDiagonal.CreateHatchBrush(HS_FDIAGONAL, RGB(0, 128, 192));
	pBrush = pDC->SelectObject(&brFDiagonal);
	pDC->RoundRect(180, 30, 320, 80, 10, 10);

	pBrush = pDC->SelectObject(&brDiagCross);
	pDC->RoundRect(340, 30, 480, 80, 10, 10);

	brVertical.CreateHatchBrush(HS_VERTICAL, RGB(255, 0, 255));
	pBrush = pDC->SelectObject(&brVertical);
	pDC->RoundRect(20, 120, 160, 170, 10, 10);

	pBrush = pDC->SelectObject(&brHorizontal);
	pDC->RoundRect(180, 120, 320, 170, 10, 10);

	brCross.CreateHatchBrush(HS_CROSS, RGB(200, 0, 0));
	pBrush = pDC->SelectObject(&brCross);
	pDC->RoundRect(340, 120, 480, 170, 10, 10);

	pDC->SetTextColor(RGB(0, 0, 255));
	pDC->TextOut(40, 10, "HS_BDIAGONAL", 12);
	pDC->SetTextColor(RGB(0, 128, 192));
	pDC->TextOut(205, 10, "HS_FDIAGONAL", 12);
	pDC->SetTextColor(RGB(0, 128, 0));
	pDC->TextOut(355, 10, "HS_DIAGCROSS", 12);
	pDC->SetTextColor(RGB(255, 0, 255));
	pDC->TextOut(44, 100, "HS_VERTICAL", 11);
	pDC->SetTextColor(RGB(255, 128, 0));
	pDC->TextOut(195, 100, "HS_HORIZONTAL", 13);
	pDC->SetTextColor(RGB(200, 0, 0));
	pDC->TextOut(370, 100, "HS_CROSS", 8);

	pDC->SelectObject(pBrush);
}
```



7.6.4 Patterned Brushes

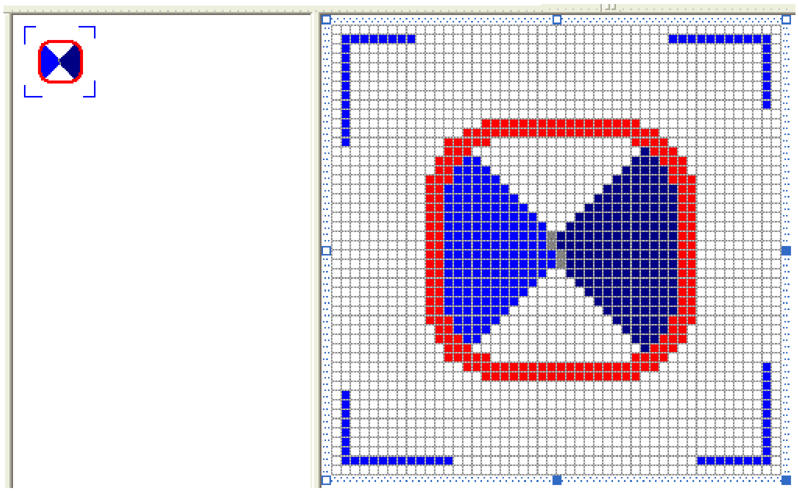
A pattern brush is one that uses a bitmap or (small) picture to fill out an area. To create DDB bitmap, you can first create an array of WORD values. Then call the **CBitmap::CreateBitmap()** method to initialize it. As this makes the bitmap ready, call the **CBrush::CreatePatternBrush()** method to initialize the brush. The syntax of this member function is:

```
BOOL CreatePatternBrush(CBitmap* pBitmap);
```

Once the brush has been defined, you can select in into a device context and use it as you see fit. For example, you can use it to fill a shape. Here is an example:

```
void CCView4View::OnDraw(CDC* pDC)
{
	CCView4Doc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBitmap Bmp;
	CBrush brBits;
	WORD wBits[] = { 0x00, 0x22, 0x44, 0x88, 0x00, 0x22, 0x44, 0x88,
	                 0x22, 0x44, 0x88, 0x00, 0x22, 0x44, 0x88, 0x00,
	                 0x44, 0x88, 0x00, 0x22, 0x44, 0x88, 0x00, 0x22,
	                 0x88, 0x00, 0x22, 0x44, 0x88, 0x00, 0x22, 0x44 };

	Bmp.CreateBitmap(32, 32, 1, 1, wBits);

	brBits.CreatePatternBrush(&Bmp);
	CBrush* pOldBrush = (CBrush*)pDC->SelectObject(&brBits);

	pDC->Rectangle(20, 20, 400, 400);

	pDC->SelectObject(&Bmp);
}
```

Another technique you can use to create a pattern brush consists of using a bitmap resource. Before creating a pattern, you must first have a picture, which can be done by creating a bitmap. For example, imagine you create the following bitmap:



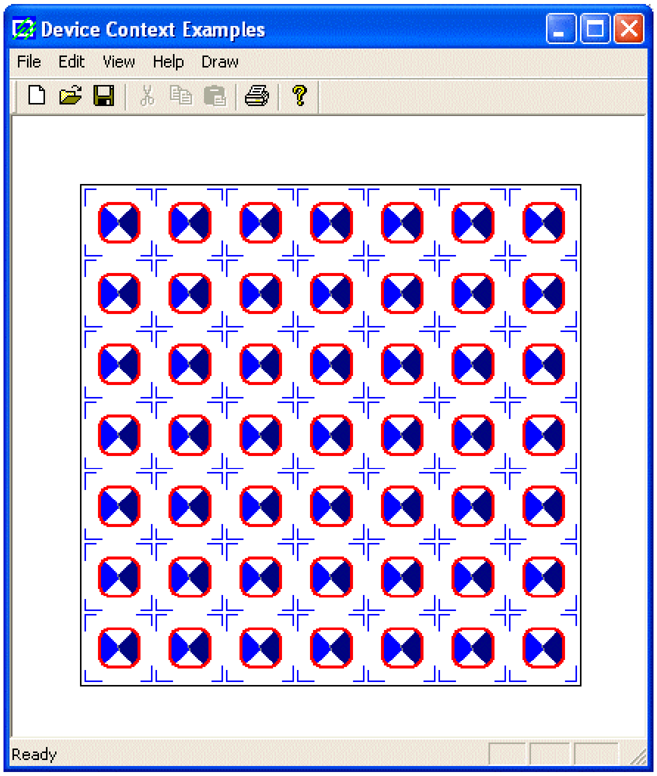
To create a brush based on a bitmap, you can use the following constructor:

```
CBrush(CBitmap* pBitmap);
```

If you had declared a CBrush variable using the default constructor, you can call the CBrush::CreatePatternBrush() member function to initialize it. Its syntax is:
BOOL CreatePatternBrush(CBitmap* pBitmap);

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush brPattern;
	CBitmap Bmp;
	CBrush *pBrush;

	Bmp.LoadBitmap(IDB_BITMAP1);//"C:\\Programs\\woman2.bmp");

	brPattern.CreatePatternBrush(&Bmp);

	pBrush = pDC->SelectObject(&brPattern);
	pDC->Rectangle(46, 46, 386, 386);

	pDC->SelectObject(pBrush);
}
```



7.6.5 Logical Brushes

The Win32 library provides the LOGBRUSH structure that can be used to create a brush by specifying its characteristics. LOGBRUSH is defined as follows:

```
typedef struct tagLOGBRUSH {
  UINT      lbStyle;
  COLORREF lbColor;
  LONG      lbHatch;
} LOGBRUSH, *PLOGBRUSH;
```

The lbStyle member variable specifies the style applied on the brush.
The lbColor is specified as a COLORREF value.
The lbHatch value represents the hatch pattern used on the brush.
Here is an example:

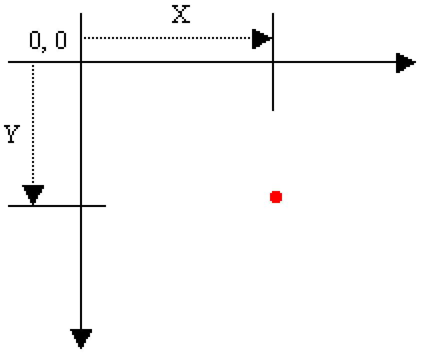
```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush *pBrush;
	CBrush brLogBrush;
	LOGBRUSH LogBrush;

	LogBrush.lbStyle = BS_HATCHED;
	LogBrush.lbColor = RGB(255, 0, 255);
	LogBrush.lbHatch = HS_DIAGCROSS;

	brLogBrush.CreateBrushIndirect(&LogBrush);
	pBrush = pDC->SelectObject(&brLogBrush);

	pDC->Rectangle(20, 12, 250, 175);

	pDC->SelectObject(pBrush);
}
```

Chapter 8: GDI Orientation and Transformations

8.1 The Default Coordinate System

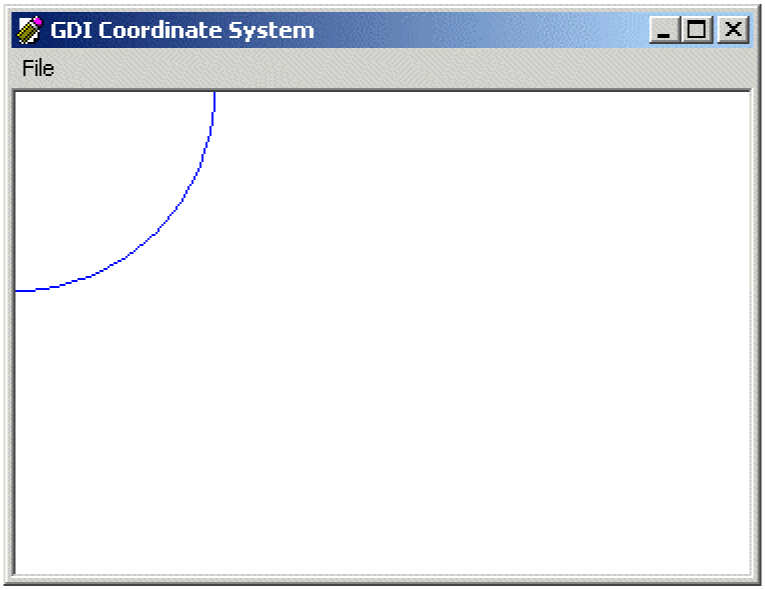
8.1.1 Introduction

When drawing on Microsoft Windows, the coordinates of the drawing area are located on the upper-left corner of the screen. Everything positioned on the screen takes its reference on that point, as we have seen in Lesson 6. That point can be illustrated in a Cartesian coordinate system as (0,0) where the horizontal axis moves from (0,0) to the right and the vertical axis moves from (0,0) down:



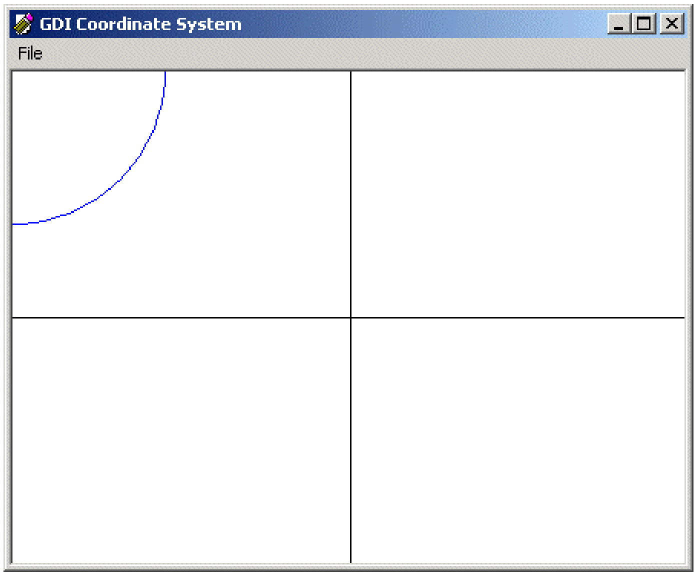
This starting origin is only the default coordinate system of the operating system. Therefore, if you draw a shape with the following call, Ellipse(-100, -100, 100, 100), you would get a circle whose center is positioned on the top-left corner of the screen. In this case, only the lower-right 3/4 of the circle would be seen:

```
void CExoView::OnDraw(CDC* pDC)
{
        CExoDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        CPen PenBlue;

        // Blue solid pen width = 1
        PenBlue.CreatePen(PS_SOLID, 1, RGB(0, 12, 255));

        CPen *pOld = pDC->SelectObject(&PenBlue);
        pDC->Ellipse(-100, -100, 100, 100);
        pDC->SelectObject(pOld);
}
```



In the same way, you can draw any geometric or non-geometric figure you want, using one of the CDC methods or creating functions of your choice. For example, the following code draws a vertical and a horizontal lines that cross each other in the center middle of the view:

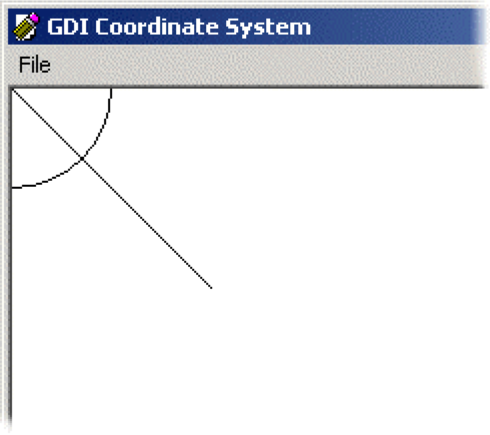
```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen  PenBlue;
	CPen *pOld;

	PenBlue.CreatePen(PS_SOLID, 1, RGB(0, 12, 255));
	pOld = pDC->SelectObject(&PenBlue);
	pDC->Ellipse(-100, -100, 100, 100);

	CPen PenBlack;
	PenBlack.CreatePen(PS_SOLID, 1, BLACK_PEN);
	pOld = pDC->SelectObject(&PenBlack);

	pDC->MoveTo(220, 0);
	pDC->LineTo(220, 370);
	pDC->MoveTo(0, 160);
	pDC->LineTo(460, 160);

	pDC->SelectObject(pOld);
}
```



8.1.2 Changing the Coordinate System

As seen above, the default coordinate system has its origin set on the top-left section of the screen. The horizontal axis moves positively from the origin to the right direction. The vertical axis moves from the origin to the bottom direction. To illustrate this, we will draw a circle with a radius whose center is at the origin (0, 0) with a radius of 50 units. We will also draw a line from the origin (0, 0) to (100, 100):

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	// A circle whose center is at the origin (0, 0)
	pDC->Ellipse(-50, -50, 50, 50);

	// A line that starts at (0, 0) and ends at (100, 100)
	pDC->MoveTo(0, 0);
	pDC->LineTo(100, 100);
}
```



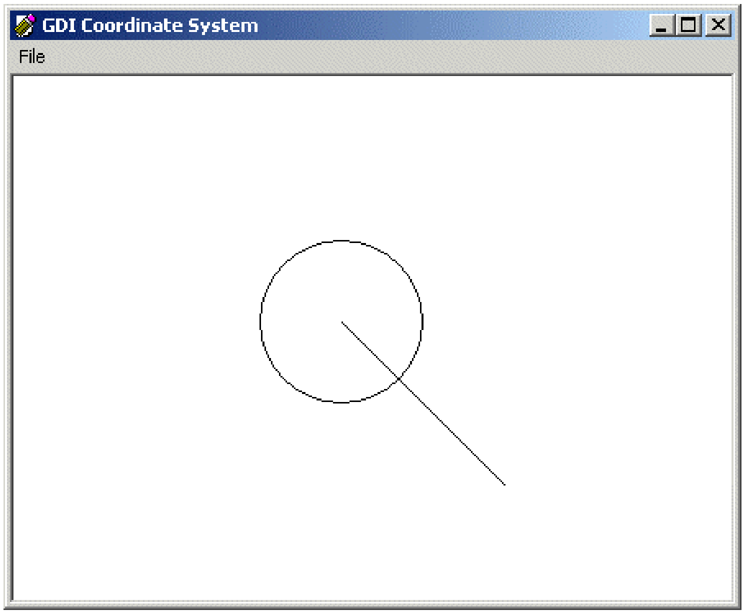
This default origin is fine for most, if not all regular, operations performed on graphics applications. Sometimes, you will need to control the position of the origin of the coordinate system. For example, most CAD applications, including AutoCAD, allow the user to set this origin.

The MFC library provides various functions to deal with the coordinates positions and extents of the drawing area, including functions used to set the origin of the coordinate system anywhere you want on the screen. Since you are drawing on a device context, all you need to do is simply call the CDC::SetViewportOrg() method. It is overloaded with two versions, which allow you to use either the X and the Y coordinates or a defined point. The syntaxes of this method are:

```
SetViewportOrg(int X, int Y);
SetViewportOrg(CPoint Pt);
```

When calling this member function, simply specify where you want the new origin to be. If using the second version, the argument can be a Win32 POINT structure or an MFC CPoint class. To see the effect of this function, we will move the origin 200 units in the positive direction of the X axis and 150 units in the positive direction of the vertical axis without changing the circle and the line. Our OnDraw() method would look like this:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetViewportOrg(200, 150);

	// A circle whose center is at the origin (0, 0)
	pDC->Ellipse(-50, -50, 50, 50);

	// A line that starts at (0, 0) and ends at (100, 100)
	pDC->MoveTo(0, 0);
	pDC->LineTo(100, 100);
}
```

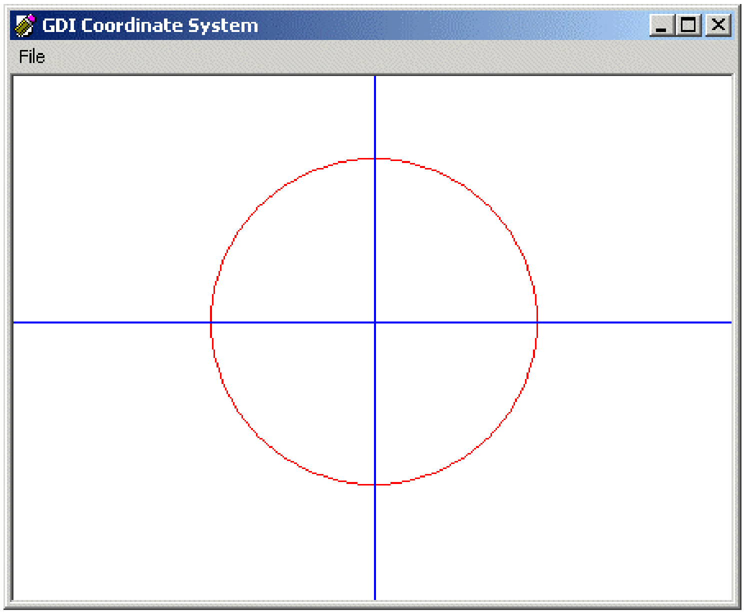


By the way, as you probably know already, you can change the dimensions of the window with a code similar to the following:

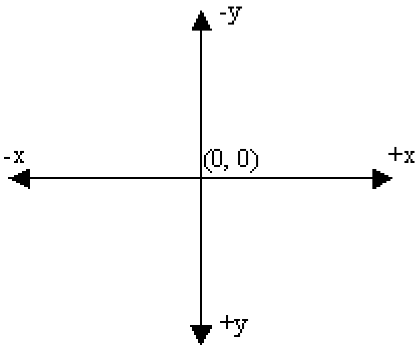
```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreateWindow(cs) )
		return FALSE;

	// The new width of the window's frame
	cs.cx = 450;
	// The new height of the window's frame
	cs.cy = 370;
	// Remove the Untitled thing
	cs.style &= ~FWS_ADDTOTITLE;

	return TRUE;
}
```

Now that we know how to control the origin, we will position it at a fixed point, 320 units to the right and 220 units down. We can also easily draw the (Cartesian) axes now:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetViewportOrg(220, 150);

	// Use a red pen
	CPen PenRed(PS_SOLID, 1, RGB(255, 0, 0));
	CPen *pOld = pDC->SelectObject(&PenRed);

	// A circle whose center is at the origin (0, 0)
	pDC->Ellipse(-100, -100, 100, 100);

	// Use a blue pen
	CPen PenBlue(PS_SOLID, 1, RGB(0, 0, 255));
	pOld = pDC->SelectObject(&PenBlue);

	// Horizontal axis
	pDC->MoveTo(-320,    0);
	pDC->LineTo( 320,    0);
	// Vertical axis
	pDC->MoveTo(   0, -220);
	pDC->LineTo(   0,  220);

	pDC->SelectObject(pOld);
}
```



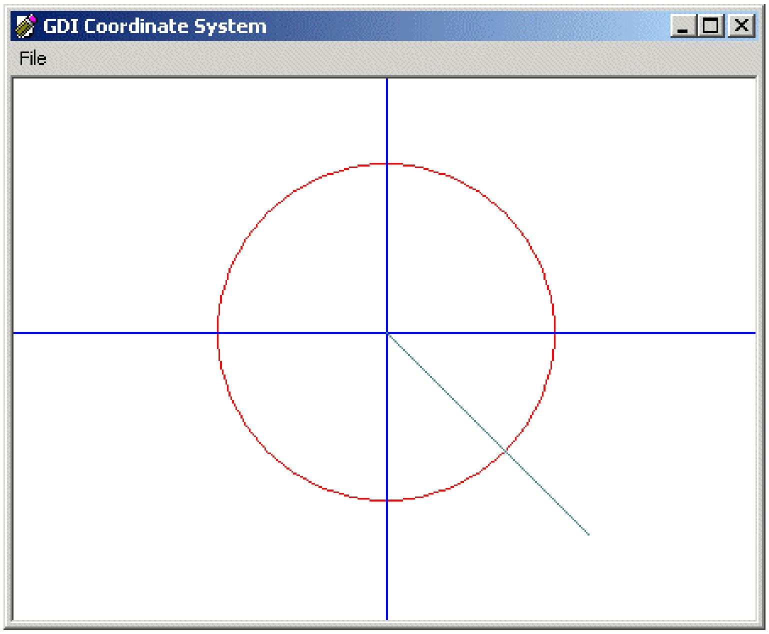
As seen already, the SetViewportOrg() member function can be used to change the origin of the device context. It also uses an orientation of axes so that the horizontal axis moves positively from (0, 0) to the right. The vertical axis moves positively from (0, 0) down:



To illustrate this, we will draw a green line at 45° from the origin:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetViewportOrg(220, 150);

	// Use a red pen
	CPen PenRed(PS_SOLID, 1, RGB(255, 0, 0));
	CPen *pOld = pDC->SelectObject(&PenRed);

	// A circle whose center is at the origin (0, 0)
	pDC->Ellipse(-100, -100, 100, 100);

	// Use a blue pen
	CPen PenBlue(PS_SOLID, 1, RGB(0, 0, 255));
	pOld = pDC->SelectObject(&PenBlue);
```

```
        // Horizontal axis
        pDC->MoveTo(-320,    0);
        pDC->LineTo( 320,    0);
        // Vertical axis
        pDC->MoveTo(   0, -220);
        pDC->LineTo(   0,  220);

        // An orange pen
        CPen PenGreen(PS_SOLID, 1, RGB(64, 128, 128));
        pOld = pDC->SelectObject(&PenGreen);

        // A diagonal line at 45 degrees
        pDC->MoveTo(0, 0);
        pDC->LineTo(120, 120);

        pDC->SelectObject(pOld);
}
```



As you can see, our line is not at 45º. Instead of being in the first quadrant, it is in the fourth. This is due to the default orientation of the coordinate system.

8.2 The Mapping Modes

8.2.1 Mapping Mode Choices

To control the orientation of the axes of the device context, you use a member function of the CDC class called SetMapMode(). Its syntax is:

```
int SetMapMode(int nMapMode);
```

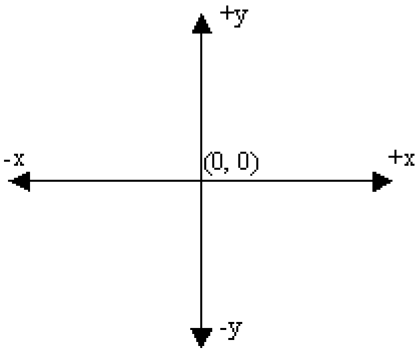
As you are about to see, this member function can be used to do two things, depending on the value of the argument. It can control the orientation of the coordinate system you want to use for your application. It also helps with the unit system you would prefer to use.

The argument of this member function is a constant integer that species the mapping mode used. The possible values are MM_TEXT, MM_LOENGLISH, MM_HIENGLISH, MM_ANISOTROPIC, MM_HIMETRIC, MM_ISOTROPIC, MM_LOMETRIC, and MM_TWIPS.

The default map mode used is the MM_TEXT. In other words, if you do not specify another, this is the one your application would use. With this map mode, the dimensions or measurements you specify in your CDC member functions are respected and kept "as is". Also, the axes are oriented so the horizontal axis moves from (0, 0) to the right and the vertical axis moves from (0, 0) down. For example, the above OnDraw() method can be re-written as follows and produce the same result:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetMapMode(MM_TEXT);
	pDC->SetViewportOrg(220, 150);

	// Use a red pen
	CPen PenRed(PS_SOLID, 1, RGB(255, 0, 0));
	CPen *pOld = pDC->SelectObject(&PenRed);

	. . . No Change

	pDC->SelectObject(pOld);
}
```

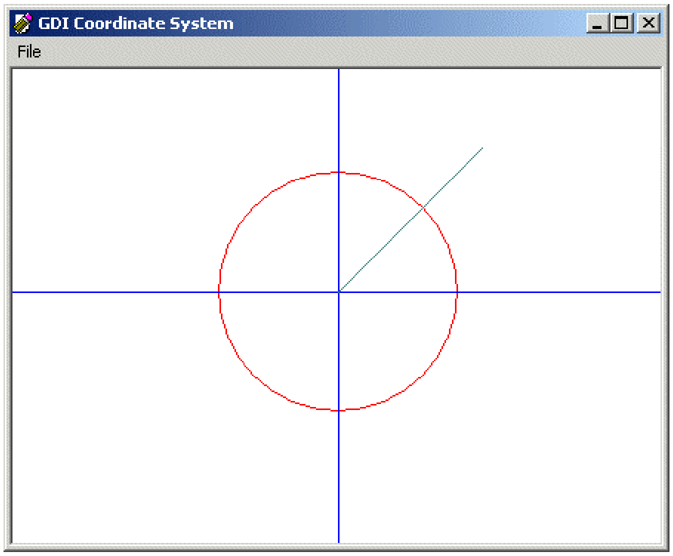
The **MM_LOENGLISH**, like some of the other mapping modes (excluding MM_TEXT as seen above), performs two actions. It changes the orientation of the vertical axis: the positive y axis would move from (0, 0) up:



Also, each unit of measure is multiplied by 0.01 inch, which means that each unit you provide is divided by 100 (unit/100). This also means that the units are reduced from their stated measures by a 100th. Observe the effect of the MM_LOENGLISH map mode on the above OnPaint() event :

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
```

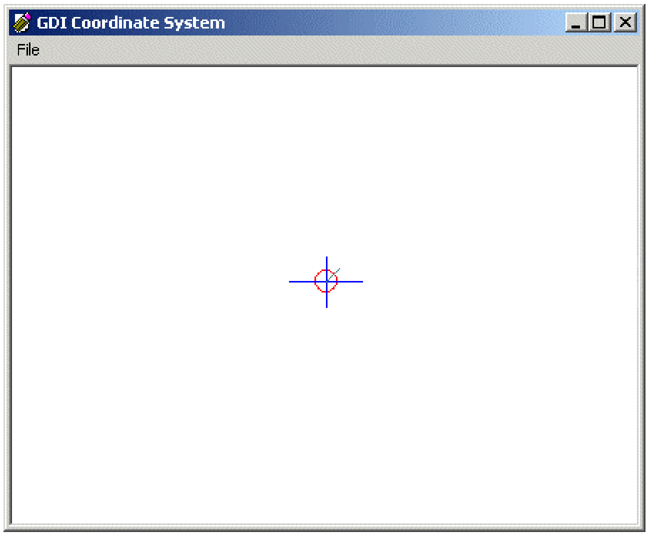
```
        ASSERT_VALID(pDoc);

        pDC->SetMapMode(MM_LOENGLISH);
        pDC->SetViewportOrg(220, 150);

        . . . No Change

        pDC->SelectObject(pOld);
}
```



As you can see, now the lines are drawn respecting the positive and the negative orientations of the axes, fulfilling the requirements of a Cartesian coordinate system. At the same time, the lengths we used have been reduced: the circle is smaller and the lines are shorter.

Like the MM_LOENGLISH map mode, the MM_HIENGLISH sets the orientation so the vertical axis moves from (0, 0) up. Also, unlike the MM_LOENGLISH, the MM_HIENGLISH map mode reduces each unit by a factor of 0.001 inch. This means that each unit is divided by 1000 (1/1000 = 1000th) which is significant and can change the display of a drawing. Here is its effect:

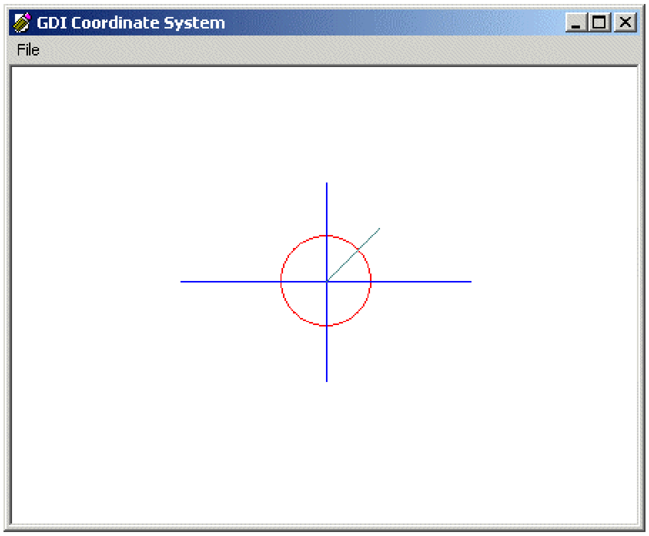
```
void CExoView::OnDraw(CDC* pDC)
{
        CExoDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        pDC->SetMapMode(MM_HIENGLISH);
        pDC->SetViewportOrg(220, 150);

        . . . No Change

        pDC->SelectObject(pOld);
}
```



Notice that we are still using the same dimensions for our lines and circle.

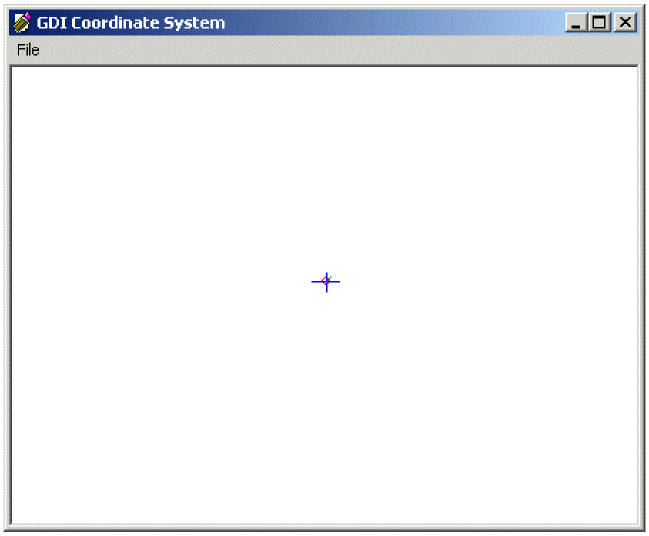
The MM_LOMETRIC map mode uses the same axes orientation as the previous two modes. By contrast, the MM_LOMETRIC multiplies each unit by 0.1 millimeter. This means that each unit is reduced by 10%. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetMapMode(MM_LOMETRIC);
	pDC->SetViewportOrg(220, 150);

	. . . No Change

	pDC->SelectObject(pOld);
}
```



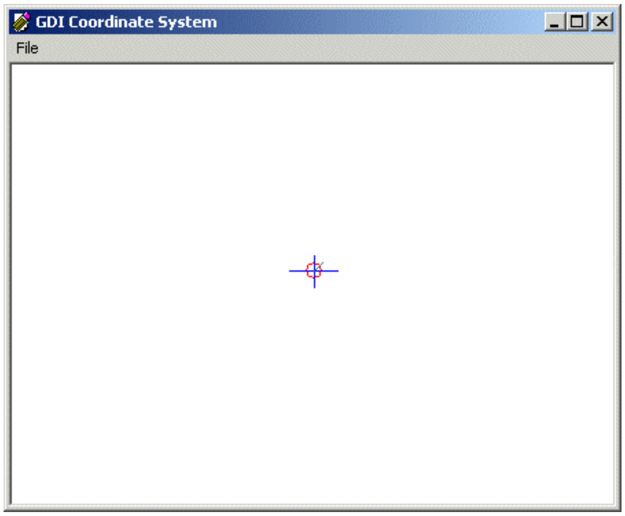
The MM_HIMETRIC mapping mode uses the same axes orientation as the above three modes. Its units are gotten by multiplying each of the given units by 0.01 millimeter. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	pDC->SetMapMode(MM_HIMETRIC);
	pDC->SetViewportOrg(220, 150);

	. . . No Change

	pDC->SelectObject(pOld);
}
```



The MM_TWIPS map mode divides each logical unit by 20. Actually a twip is equivalent to 1/1440 inch. Besides this unit conversion, the axes are oriented so the horizontal axis moves from the origin (0, 0) to the right while the vertical axis moves from the origin (0, 0) up. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDo c);

	pDC->SetMapMode(MM_TWIPS);
	pDC->SetViewportOrg(220, 150);

	. . . No Change

	pDC->SelectObject(pOld);
}
```



8.2.2 Unit and Coordinate Systems Options

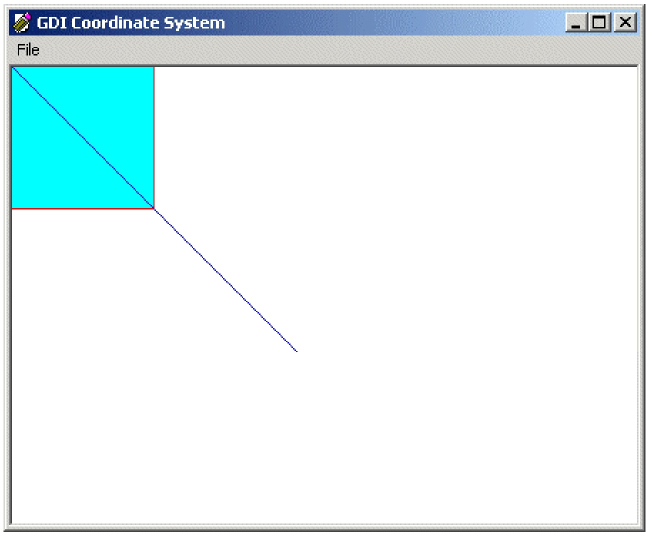
The mapping modes we have used so far allowed us to select the orientation of the axes, especially the y axis. Nevertheless, we could not influence any conversion unit for the dimensions we specified on our drawings. This is because each one of these mapping modes (MM_TEXT, MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, and MM_TWIPS) has a fixed set of attributes such as the orientation of its axes and the conversion used on the provided dimensions.

Consider the following OnDraw() method. It draws a 200x200 pixels square with a red border and an aqua background. The square starts at 100x100 pixels on the negative sides of both axes and it continues 100x100 pixels on the positive sides of both axes. For better illustration, the event also draws a diagonal line at 45º starting at the origin (0, 0):

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen      PenRed(PS_SOLID, 1, RGB(255, 0, 0));
	CBrush   BrushAqua(RGB(0, 255, 255));
	CBrush *brOld;
	CPen   *pnOld;

	pnOld = pDC->SelectObject(&PenRed);
	brOld = pDC->SelectObject(&BrushAqua);

	// Draw a square with a red border and an aqua background
	pDC->Rectangle(-100, -100, 100, 100);

	CPen BluePen(PS_SOLID, 1, RGB(0, 0, 255));
	pnOld = pDC->SelectObject(&BluePen);

	// Diagonal line at 45 degrees starting at the origin (0, 0)
	pDC->MoveTo(0, 0);
	pDC->LineTo(200, 200);

	pDC->SelectObject(pnOld);
```

```
        pDC->SelectObject(brOld);
}
```

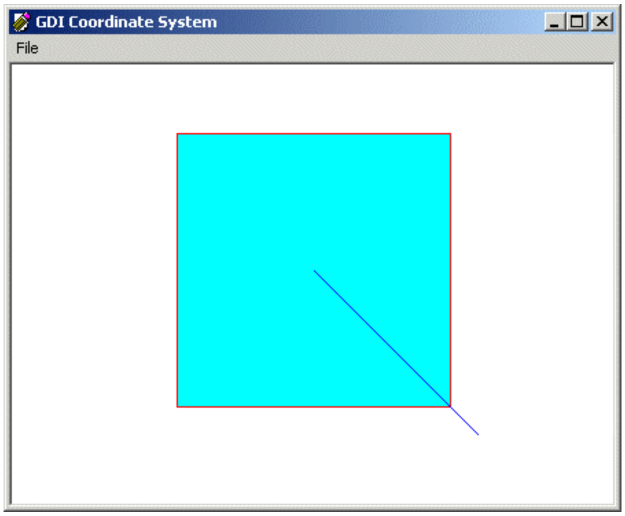
This would produce:



As you can see, we get only the the lower-right 3/4 portion of the square and the line is pointing in the 3 to 6 quadrant of a clock .

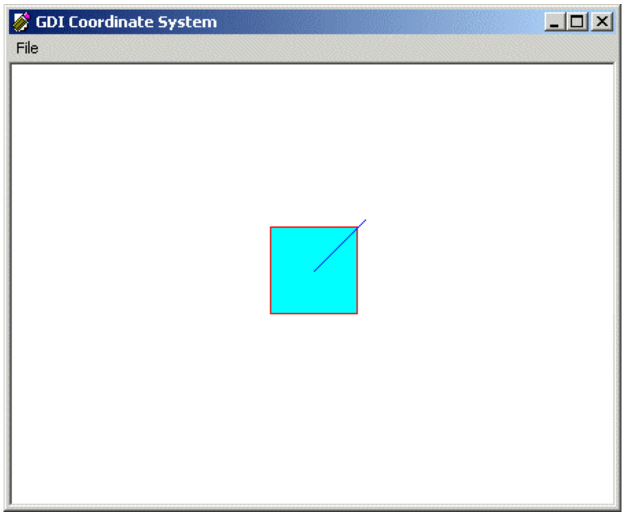
Imagine that you want the origin (0, 0) to be positioned at the (220, 150) point. We saw already that you can use the CDC::SetViewportOrg() (keep in mind that this member function only changes the origin of the coordinate system; it does not influence the orientation of the axes nor does it control the units or dimensions) method to specify the origin. Here is an example (we are not specifying the mapping mode because MM_TEXT can be used as the default):

```
void CExoView::OnDraw(CDC* pDC)
{
        CExoDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        CPen      PenRed(PS_SOLID, 1, RGB(255, 0, 0));
        CBrush    BrushAqua(RGB(0, 255, 255));
        CBrush *brOld;
        CPen    *pnOld;

        pnOld = pDC->SelectObject(&PenRed);
        brOld = pDC->SelectObject(&BrushAqua);

        pDC->SetViewportOrg(220, 150);

        // Draw a square with a red border and an aqua background
        pDC->Rectangle(-100, -100, 100, 100);

        CPen BluePen(PS_SOLID, 1, RGB(0, 0, 255));
        pnOld = pDC->SelectObject(&BluePen);

        // Diagonal line at 45 degrees starting at the origin (0, 0)
        pDC->MoveTo(0, 0);
        pDC->LineTo(120, 120);
        pDC->SelectObject(pnOld);
        pDC->SelectObject(brOld);
}
```



To control your own unit system, the orientation of the axes or how the application converts the units used on your application, use either the MM_ISOTROPIC or the MM_ANISOTROPIC mapping modes. The first thing you should do is to call the CDC::SetMapMode() member function and specify one of these two constants (either MM_ISOTROPIC or MM_ANISOTROPIC). Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
        CExoDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        CPen     PenRed(PS_SOLID, 1, RGB(255, 0, 0));
        CBrush   BrushAqua(RGB(0, 255, 255));
        CBrush *brOld;
        CPen   *pnOld;

        pnOld = pDC->SelectObject(&PenRed);
        brOld = pDC->SelectObject(&BrushAqua);

        pDC->SetMapMode(MM_ISOTROPIC);
        pDC->SetViewportOrg(220, 150);

        // Draw a square with a red border and an aqua background
        pDC->Rectangle(-100, -100, 100, 100);

        CPen BluePen(PS_SOLID, 1, RGB(0, 0, 255));
        pnOld = pDC->SelectObject(&BluePen);

        // Diagonal line at 45 degrees starting at the origin (0, 0)
        pDC->MoveTo(0, 0);
        pDC->LineTo(120, 120);

        pDC->SelectObject(pnOld);
        pDC->SelectObject(brOld);
}
```



You should not rely on the above picture. After calling the CDC::SetMapMode() function with MM_ISOTROPIC (or MM_ANISOTROPIC) as argument, you are not supposed to stop there. The purpose of these two map modes is to let you control the orientation of the axes and the conversion of the units.

The difference between both mapping modes is that, when using the MM_ISOTROPIC map mode, one unit in the horizontal axis is equivalent to one unit in the vertical axis. This is not the case for the MM_ANISOTROPIC map mode which allows you to control however the units should be converted on each individual axis.

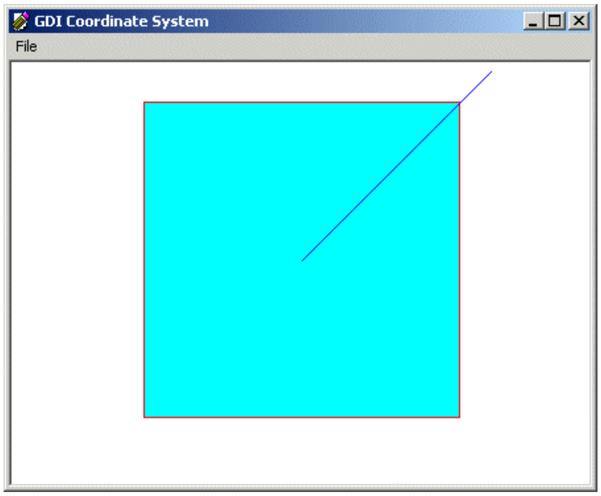
Therefore, after calling SetMapMode() and specifying the MM_ISOTROPIC (or MM_ANISOTROPIC), you must call the CDC:SetWindowExt() member function. This method specifies how much each new unit will be multiplied by the old or default unit system. The CDC::SetWindowExEx() member function comes in two versions with the following syntaxes:

```
CSize SetWindowExt(int cx, int cy);
CSize SetWindowExt(SIZE size);
```

If using the first version, the first argument to this function, cx, specifies the logical conversion mu ltiplier used for each unit on the horizontal axis. The second argument, cy, specifies the logical conversion multiplier used for each unit on the vertical axis.

The second version of the method can be used if you know the desired logical width and height as a SIZE structure. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen	PenRed(PS_SOLID, 1, RGB(255, 0, 0));
	CBrush	BrushAqua(RGB(0, 255, 255));
	CBrush *brOld;
	CPen   *pnOld;

	pnOld = pDC->SelectObject(&PenRed);
	brOld = pDC->SelectObject(&BrushAqua);
	pDC->SetMapMode(MM_ISOTROPIC);
	pDC->SetViewportOrg(220, 150);
	pDC->SetWindowExt(640, 640);

	// Draw a square with a red border and an aqua background
	pDC->Rectangle(-100, -100, 100, 100);

	CPen BluePen(PS_SOLID, 1, RGB(0, 0, 255));
	pnOld = pDC->SelectObject(&BluePen);

	// Diagonal line at 45 degrees starting at the origin (0, 0)
	pDC->MoveTo(0, 0);
	pDC->LineTo(120, 120);

	pDC->SelectObject(pnOld);
	pDC->SelectObject(brOld);
}
```



After calling the SetWindowExt() member function, you should call the SetViewportExt() method. Its job is to specify the horizontal and vertical units of the device context being used. It comes in two versions with the following syntaxes:

```
CSize SetViewportExt(int cx, int cy);
CSize SetViewportExt(SIZE size);
```

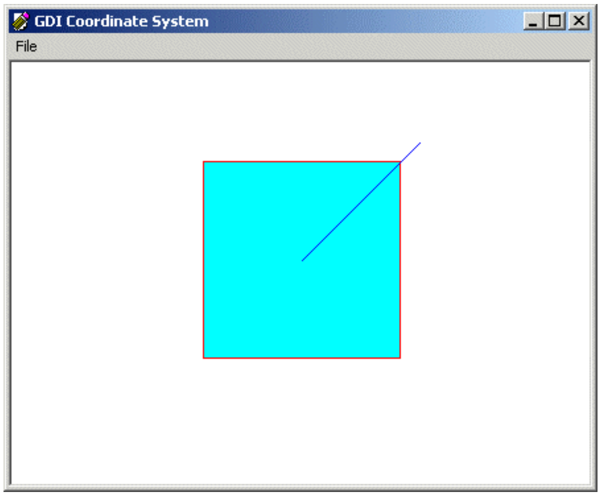
To use the first version of this function, you must provide the units of device conversion as cx for the horizontal axis and as cy for the vertical axis.

If you know the size as a width/height combination of the device unit conversion, you can use the second version of the function and supply this size argument.

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPen	PenRed(PS_SOLID, 1, RGB(255, 0, 0));
```

```
            CBrush   BrushAqua(RGB(0, 255, 255));
            CBrush *brOld;
            CPen   *pnOld;

            pnOld = pDC->SelectObject(&PenRed);
            brOld = pDC->SelectObject(&BrushAqua);

            pDC->SetMapMode(MM_ISOTROPIC);
            pDC->SetViewportOrg(220, 150);
            pDC->SetWindowExt(640, 640);
            pDC->SetViewportExt(480, -480);

            // Draw a square with a red border and an aqua background
            pDC->Rectangle(-100, -100, 100, 100);

            CPen BluePen(PS_SOLID, 1, RGB(0, 0, 255));
            pnOld = pDC->SelectObject(&BluePen);

            // Diagonal line at 45 degrees starting at the origin (0, 0)
            pDC->MoveTo(0, 0);
            pDC->LineTo(120, 120);

            pDC->SelectObject(pnOld);
            pDC->SelectObject(brOld);
}
```



Chapter 9: Strings

- ✍ Fundamentals of Strings
- ✍ Operations on Strings
- ✍ Characters of a String
- ☞ The CString Class
- ☞ Working with Individual Characters
- ☞ Sub Strings

9.1 Fundamentals of Strings

9.1.1 Null-Terminated Strings

A string is a group of printable letters or symbols. The symbol can be used as a single character or some symbols can be combined to produce a word or a sentence. The symbols are aligned in the computer memory in a consecutive manner so the last symbol is the null character. Here is an example:

M	a	t	h	e	m	a	t	i	c	s	\0

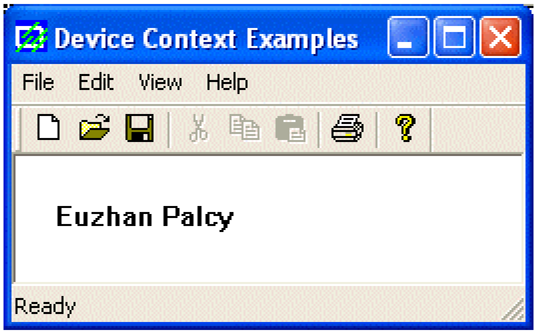
Such a value is called a null-terminated string because the last symbol is the null-terminated character "\0". When allocating space for such a string, the space must be provided as the number of its characters + 1.

Like most other variables, to use a string, you can first declare a variable that would hold it. To declare a string variable, you can use the **char** data type and create an array of symbols. Here are two examples:

```
char StrName[20];
char *Sentence;
```

The easiest way to initialize one of these variables is to give it a value when the variable is declared. Once it has been initialized, such a variable can be used in any function that can take advantage of it. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
    char *StrName = "Euzhan Palcy";

    pDC->TextOut(20, 22, StrName, 12);
}
```



Euzhan Palcy

When a string variable has been initialized, the string the variable holds is called its value. In the example above, Euzhan Palcy is the string value, or simply the value, of the StrName variable.

When writing applications for Microsoft Windows, you can also declare a string using the **CHAR** data type. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
    CHAR *StrName = "Euzhan Palcy";

    pDC->TextOut(20, 22, StrName, 12);
}
```

If you are writing for an international audience and involve unicode in your application, you can declare the string using the **TCHAR** data type:

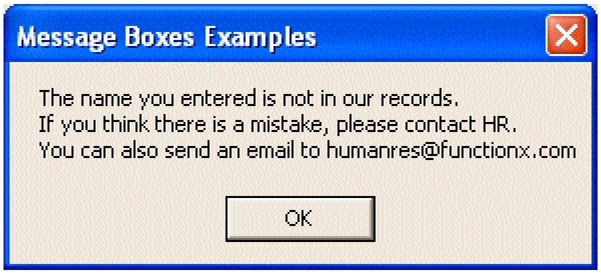
```
void CExoView::OnDraw(CDC* pDC)
{
    TCHAR *StrName = "Euzhan Palcy";

    pDC->TextOut(20, 22, StrName, 12);
}
```

Alternatively, you can use **_TCHAR** to declare such a variable. Normally, to initialize a null-terminated string variable for an international audience, you should use the **_T**, the **TEXT** or the **_TEXT** macros. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
    char *StrName = _T("Euzhan Palcy");

    pDC->TextOut(20, 22, StrName, 12);
}
```

You can also initialize a string variable with a value that spans various lines. To do this, when creating the string, end each line with a double-quote and start the other with another double-quotes. Here is an example:

```
void CExerciseDlg::OnMsgBox()
{
            // TODO: Add your control notification handler code here
            char Sentence[] = "The name you entered is not in our records.\n"
                              "If you think there is a mistake, please contact HR.\n"
                              "You can also send an email to humanres@functionx.com";
            MessageBox(Sentence);
}
```



The name you entered is not in our records.
If you think there is a mistake, please contact HR.
You can also send an email to humanres@functionx.com

9.1.2 The Standard string Class

The Standard Template Library (STL) provides a class that can be used for string manipulation. This class is called **basic_string** and it was type-defined as **string**. To use this class, on top of the file where you will need access to it, include the **string** library and call the **std namespace**. To do this, you would type:

```
#include <string>
using namespace std;
```

Therefore, to declare a variable using the STL's string class, use one of its constructors. For example, you can use the default constructor as follows:

```
string FirstName;
```

To initialize a string variable, you can provide its value between parentheses. Here is an example:

```
string FullName(“Jules Andong”);
```

The STL's **string** class is not natively supported in MFC applications. Therefore, in order to use it, you must convert its value to a null-terminated string. To allow this, the string class provides the **c_str()** member function. Its syntax is:

```
const E *c_str() const;
```

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	string StrName("Central African Republic");

	pDC->TextOut(20, 22, StrName.c_str(), 24);
}
```

9.1.3 The Length of a String

The length of a string is the number of characters that the string contains. To get the length of a string declared using the **char**, the **CHAR**, the **TCHAR**, the **_TCHAR** or one of the Win32 API's data types, you can use the **strlen()** function. Its syntax is:

```
size_t strlen(const char *String);
```

This function returns the number of characters of the *String* argument. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	char StrName[] = "Euzhan Palcy";
	int Len = strlen(StrName);

	pDC->TextOut(20, 22, StrName, Len);
}
```

To get the length of a string declared using the **string** class, call the **length()** member function. Its syntax is:

```
size_type length() const;
```

You can also get the same result by calling the **size()** method whose syntax is:

```
size_type size() const;
```

Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	string StrName("Central African Republic");

	int Len = StrName.length();
	pDC->TextOut(20, 22, StrName.c_str(), Len);
}
```

9.1.4 String Formatting

Formatting a string consists of telling the compiler how to adjust, before using such as displaying, the string. Imagine you declare a null-terminated string as follows:

```
char Represents[20];
```

To format a null-terminated string, you can call the **sprintf()** function. Its syntax is:

```
int sprintf(char* Buffer, const char* S, Arguments...);
```

This function takes at least two arguments but can take as manu as necessary. The first argument, *Buffer*, is a null-terminated string variable:

```
char Represents[20];
sprintf(Represents,
```

To display a value as part of the *Buffer* variable, provide the second argument starting it with a double-quote followed by a string if any, followed by the **%** sign, followed by one of the following characters:

Character	Type of Data	Used for
c	char	A single character
d	int	Any signed decimal integer (int, short)
i	int	Any signed decimal integer (int, short)
o	Signed Integers	An unsigned octal integer
u	Unsigned	Any unsigned decimal integer (unsigned, unsigned int)
x	Unsigned	Any unsigned lowercase hexadecimal integer (abcdef)
X	Unsigned	Any unsigned uppercase hexadecimal integer (ABCDEF)
e	double	Scientific representation of a signed value used to display the lowercase exponent
E	double	Scientific representation of a signed value used to display the uppercase exponent
f	double	Representation of a floating point number with a specific number of characters to the right of

		the separator
g	double	Representation of a number using either the e or the f format
G	double	Representation of a number using either the E or the f format
s	String	String representation
S	String	String representation

An example would be:

```
char Represents[20];
sprintf(Represents, "%f", );
```

If you want to display your own, unformatted words as part of the new string, you can type anything before the % operator or after the formatting character:

```
char Represents[20];
sprintf(Represents, "The number is %f", );
```

If you are formatting a floating point number, you can specify the number of digits to appear as the separator. To do this, type a period on the right side of the % symbol followed by a number. For example, to produce a number with a precision of 4 digits, you would type:

```
sprintf(Represents, "The number is %.4f", );
```

To create a longer or more advanced string, you can add as many combinations of % and characters as necessary.

The % symbol used in the S argument is used as a place holder, informing the compiler that an external value will be needed for that position. To specify a value for a place holder, provide it as a third argument. Here is an example:

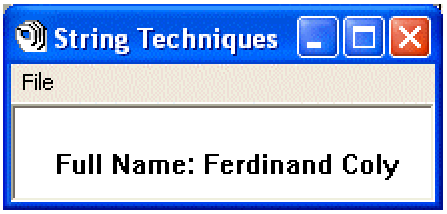
```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	char Represents[20];
	double Number = 125.55;
	sprintf(Represents, "The number is %.4f", Number);
}
```

If you created more than one combination of % and character, you can provide a value for each, separated by a comma. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	char FirstName[] = "Ferdinand";
	char LastName[]  = "Coly";
	char FullName[40];

	sprintf(FullName, "Full Name: %s %s", FirstName, LastName);
}
```

After formatting the string, the *Buffer* argument holds the new string. You can then use it as you see fit. For example, you can pass it to another function such as displaying it on a view as follows:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	char FirstName[] = "Ferdinand";
	char LastName[]  = "Coly";
	char FullName[40];

	sprintf(FullName, "Full Name: %s %s", FirstName, LastName);
	int Len = strlen(FullName);

	pDC->TextOut(20, 20, FullName, Len);
}
```



Full Name: Ferdinand Coly

9.2 Operations of Strings

9.2.1 String Copy

Copying a string is equivalent to assigning all of its characters to another string. After copying, both strings would have the same value. To copy a null-terminated string, you can use the **strcpy()** function. Its syntax is:

char *strcpy(char **strDestination*, const char **strSource*);

The *strDestination* argument is the target string. The *strSource* argument holds the value of the string that needs to be copied. During execution, the *strSource* value would be assigned to the *strDestination* variable. After execution, this function returns another string that has the same value as *strDestination*.

Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	char Artist[] = _T("Arsene Wendt");
	char Producer[40];
```

```
        strcpy(Producer, Artist);

        int Len = strlen(Producer);
        pDC->TextOut(20, 20, Producer, Len);
}
```

To make a copy of a **string** variable, you can use the assignment operator. The primary technique applied to this operator can be used to initialize a string variable declared using the default constructor. Here is an example:

```
void CCView1View::OnDraw(CDC* pDC)
{
        CCView1Doc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // TODO: add draw code for native data here
        string StrName = "Central African Republic";

        int Len = StrName.length();
        pDC->TextOut(20, 22, StrName.c_str(), Len);
}
```

Using the assignment operator, you can copy one **string** variable into another. To do this, simply assign one variable to another. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
        CExerciseDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        string Artist = _T("Arsene Wendt");
        string Producer;

        Producer = Artist;

        int Len = Producer.length();
        pDC->TextOut(20, 20, Producer.c_str(), Len);
}
```

Besides the assignment operator, to copy a **string** variable or to assign the value of one string to another, you can also use the **string::assign()** method to assign a string. It is provides in various syntaxes as follows:

```
basic_string& assign(const E *s);
basic_string& assign(const E *s, size_type n);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
basic_string& assign(const basic_string& str);
basic_string& assign(size_type n, E c);
basic_string& assign(const_iterator first, const_iterator last);
```

Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
        CExerciseDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        string Artist = _T("Arsene Wendt");
        string Producer;

        Producer.assign(Artist);

        int Len = Producer.length();

        pDC->TextOut(20, 20, Producer.c_str(), Len);
}
```

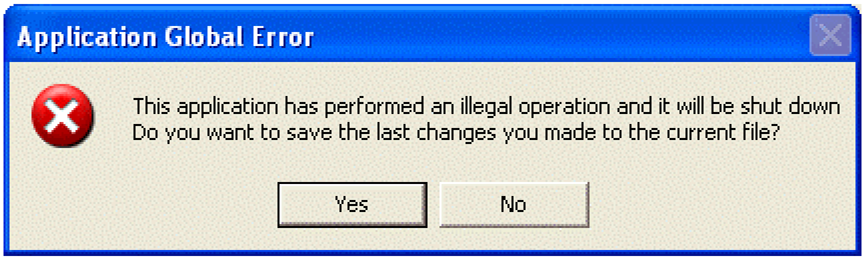
9.2.2 String Concatenation

String concatenation consists of adding one string to another. To add one null-terminated string to another, you can use the **strcat()** function. Its syntax is:

```
char *strcat(char *Destination, const char *Source);
```

The **strcat()** function takes two arguments. The second argument, *Source*, is the string to add to the right side of the first string, *Destination*. Here is an example:

```
void CExerciseDlg::OnMsgBox()
{
    // TODO: Add your control notification handler code here
    static char Msg[] = "This application has performed an illegal operation ";
    strcat(Msg, "and it will be shut down");
    strcat(Msg, "\nDo you want to save the last changes you made to the current file?");
    const char Title[] = "Application Global Error";
    UINT Option = MB_YESNO | MB_ICONSTOP;

    ::MessageBox(NULL, Msg, Title, Option);
}
```


Application Global Error

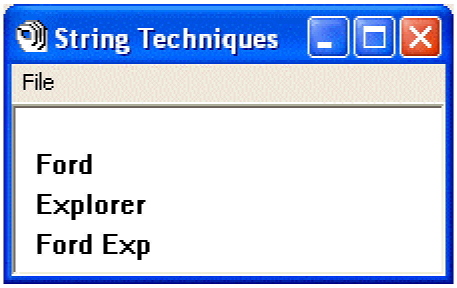
This application has performed an illegal operation and it will be shut down
Do you want to save the last changes you made to the current file?

Like the **strcat()** function, the **strncat()** function is used to append one string to another. The difference is that, while the **strcat()** considers all characters of the source string, the **strncat()** function allows you to specify the number of characters from the source string that you want to append to the destination string. This means that, if the source string has 12 characters, you can decide to append only a set number of its characters. The syntax of the **strncat()** function is:

```
char* strncat(char* Destination, const char* Source, int Number);
```

Besides the same arguments as the **strcat()** function, the *Number* argument is used to specify the number of characters considered from *Source*. To perform the concatenation, the compiler would count *Number* characters from left to right of the *Source* string. These characters would be added to the right of the *Destination* string. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
        CExerciseDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        char Make[]      = "Ford ";
        int LengthMake  = strlen(Make);
        char Model[]     = "Explorer";
        int LengthModel = strlen(Model);

        char *Car = strncat(Make, Model, 3);
        int LengthCar = strlen(Car);

        pDC->TextOut(10, 20, Make, LengthMake);
        pDC->TextOut(10, 40, Model, LengthModel);
        pDC->TextOut(10, 60, Car, LengthCar);
}
```



```
Ford
Explorer
Ford Exp
```

The **string** class provides its own mechanism to add two strings. It uses the addition operator. To do this, simply add one **string** value or variable to another **string** value or variable and assign the result to the target**string** variable. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
        CExerciseDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        string Make  = "Ford ";
        string Model = "Explorer";

        string Car = Make + Model;
        int LengthCar = Car.length();

        int LengthMake  = Make.length();
        int LengthModel = Model.length();

        pDC->TextOut(10, 10, Make.c_str(), LengthMake);
        pDC->TextOut(10, 30, Model.c_str(), LengthModel);
        pDC->TextOut(10, 50, Car.c_str(), LengthCar);
}
```

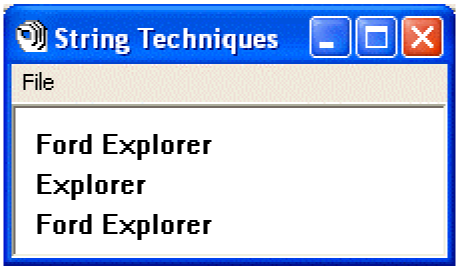
Alternatively, to add two strings, you can call the **append()** method. It comes in various versions as follows:

```
basic_string& append(const E *s);
basic_string& append(const E *s, size_type n);
basic_string& append(const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const basic_string& str);
```

```
basic_string& append(size_type n, E c);
basic_string& append(const_iterator first, const_iterator last);
```

In the following example, we use the first versions that takes a **string** variable as argument and add the value of that argument to the right side of the **string** variable that called it. After this version executes, it modifies the value of the string that made the call:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	string Make  = "Ford ";
	string Model = "Explorer";

	string Car = Make.append(Model);
	int Le ngthCar = Car.length();

	int LengthMake  = Make.length();
	int LengthModel = Model.length();

	pDC->TextOut(10, 10, Make.c_str(), LengthMake);
	pDC->TextOut(10, 30, Model.c_str(), LengthModel);
	pDC->TextOut(10, 50, Car.c_str(), LengthCar);
}
```



```
Ford Explorer
Explorer
Ford Explorer
```

If you want to append only a specific number of characters, use the second version of the **append()** method.

9.3 The Characters of a String

9.3.1 Access to Characters

As mentioned earlier, the characters of a null-terminated string are stored in an array. To access an individual character of a null-terminated string, use the square brackets operator. For example, if you had declared and initialized a string using

char FirstName[] = _T("Arsene");

you can access the third character using FirstName[2];

The **strchr()** function looks for the first occurrence of a certain character in a null-terminated string. Its syntax is:

```
char* strchr(const char* S, char c);
```

This function takes two arguments. The second argument, c, specifies what character to look for in the first argument, S, which is a string. If character c appears in string S, the function would return a new string whose value starts at the first occurrence of c in S. If the character c does not appear in the string S, then the function would return NULL.

The **strrchr()** function examines a string starting at the end (right side) of the string and looks for the first occurrence of a certain character. Its syntax is:

```
char* strrchr(const char* S, char c);
```

The first argument is the string that needs to be examined. The function will scan string S from right to left. Once it finds the first appearance of the character c in the string, it would return a new string whose value starts at that first occurrence. If the character c does not appear in the string S, then the function would return NULL.

9.3.2 Sub-Strings

The **strstr()** function looks for the first occurrence of a sub-string in another string and returns a new string as the remaining string. Its syntax is:

```
char* strstr(const char* Main, const char *Sub);
```

The first argument of the function is the main string that would be examined. The function would look for the second argument, the *Sub* string appearance in the *Main* string. If the *Sub* string is part of the *Main* string, then the function would return a string whose value starts at the first appearance of *Sub* and make it a new string. If *Sub* is not part of the *Main* string, the function would return a NULL value.

9.4 The CString Class

9.4.1 Introduction

The MFC library provides its own data types and a class for string manipulation. The class is called **CString**. To declare a string with the **CString** class, you can use one of its constructors. To initialize a string when declaring it, you can pass a string to one of the following constructors:

```
CString(const unsigned char* psz);
CString(LPCWSTR lpsz);
CString(LPCSTR lpsz);
```

These constructors take a null-terminated string as argument. Once the variable has been declared, an amount of memory space is assigned for its use. If for any reason the memory or enough memory could not be allocated, an exception would occur. If the space allocated for the variable is higher than necessary, at any time, you can reduce it by calling the **CString::FreeExtra()** method. Its syntax is:

```
void FreeExtra();
```

9.4.2 String Initialization

So, to deaclare and initialize a **CString** variable, you can use one of the above constructors. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Term("Information Superhighway");

	pDC->TextOut(20, 20, Term);
}
```

The above constructors mainly allow you to supply a null-terminated string as the initial value of the variable. In fact, you can first declare a null-terminated C string and pass it as argument to the constructor to initialize the string. Here is an exa mple:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	const char *Ville("Antananarivo");
	CString Capital(Ville);

	pDC->TextOut(10, 20, Ville);
	pDC->TextOut(10, 40, Capital);
}
```

You can also create a string using the String Table and specifying its value. To initialize a CString variable for such a string, call the CString::LoadString() method. Its syntax is:

BOOL LoadString(UINT *nID*);

The *nID* argument is the identifier of the item created in the String Table

9.4.3 The String and its Length

After initializing a string, it assumes a length that represents its number of characters. To get the length of a string, you can call the **CString::GetLength()** method whose syntax is:

```
int GetLength( ) const;
```

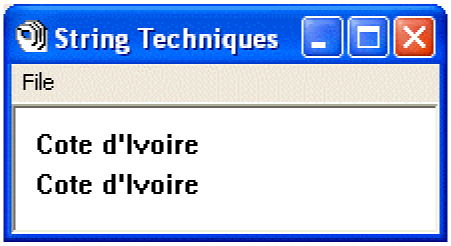
However you have declared and initialized the CString variable using one of the above constructors, if you want to retrieve the value stored in the variable, you can call the **CString::GetBuffer()** method. Its syntax is:

```
LPTSTR GetBuffer( int nMinBufLength );
```

The *nMinBufLength* argument specifies the minimum number of characters to consider from the string. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
```

```
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Country("Cote d'Ivoire");
	const char *Area = Country.GetBuffer(2);

	pDC->TextOut(10, 10, Country);
	pDC->TextOut(10, 30, Area);
}
```



```
Cote d'Ivoire
Cote d'Ivoire
```

If you want to specify the number of characters to consider when retrieving the buffer, use the CString::GetBufferSetLength() method. Its syntax is:

LPTSTR GetBufferSetLength(int *nNewLength*);

The *nNewLength* argument specifies the new length of the buffer.

To declare a **CString** variable that has a single character, use the following constructor:

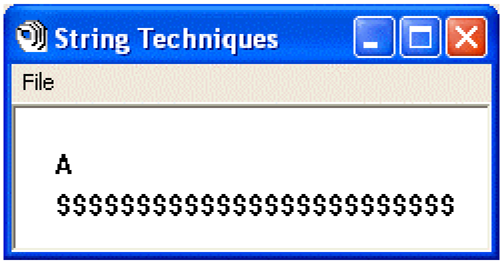
CString(TCHAR *ch*, int *nRepeat* = 1);

The character can be included between single-quotes. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Letter('A');

	pDC->TextOut(20, 22, Letter);
}
```

If you want the character to be repeated more than once, you can pass a second argument as nRepeat that holds the number of times that the ch character must be repeated. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Letter('A');
	pDC->TextOut(20, 20, Letter);

	CString Character('$', 25);
	pDC->TextOut(20, 40, Character);
}
```



```
A
$$$$$$$$$$$$$$$$$$$$$$$$$
```

9.5 Working with Individual Characters

9.5.1 Character Indexing

The characters or symbols that compose a **CString** variable are stored in an array of characters. To get to the character or symbol at a specific position, you can use the square brackets, the same way you would proceed for an array. This is possible because the square bracket operator is loaded for the **CString** class:

```
TCHAR operator []( int nIndex ) const;
```

Like an array, the characters of a **CString** value are stored in a 0-based index, meaning the first character is at index 0, the second at 2, etc. Imagine you have a string declared and initialized as:

```
CString Capital("Antananarivo");
```

To get the 5th character of this string, you could write:

```
CString Fifth = Capital[4];
```

Here si an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Capital("Antananarivo");
	CString Fifth = Capital[4];

	pDC->TextOut(10, 20, Capital);
	pDC->TextOut(10, 40, Fifth);
}
```

Besides the square brackets, to access the character stored at a certain position in a string, you can call the **GetAt().** Its syntax is:

```
TCHAR GetAt( int nIndex ) const;
```

Here is an example of using it:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Capital("Antananarivo");
	CString Fifth = Capital.GetAt(4);

	pDC->TextOut(10, 20, Capital);
	pDC->TextOut(10, 40, Fifth);
}
```

9.5.2 Character Insertion

As stated already, a **CString** value is in fact an array of characters. This allows you to locate a position in the string and change its character. To do this, you can use the **SetAt()** method. Its syntax is:

```
void SetAt(int nIndex, TCHAR ch);
```

Here is an example:

```
#include <afxwin.h>
#include <iostream>
using namespace std;

int main()
{
	CString Object("Wall");

	cout << "Object: " << (LPCTSTR)Object << endl;
	Object.SetAt(1, 'e');
	cout << "Name:   " << (LPCTSTR)Object << endl;

	return 0;
}
```

9.5.3 Finding a Character

Scanning a string consists of visiting or examining each one of its characters. One of the reasons you would do this is to look for a particular character that may be part of the string.

To scan a string for a character, you can call the **CString::Find()** method. It is overloaded as follows:

```
int Find(TCHAR ch) const;
int Find(TCHAR ch, int nStart) const;
```

To look for a single character in the string, pass a character value or variable as argument. In this case, the string would be examined from the most left character to the right. If you want the scanning to start at a certain position instead of from the most left character, besides the character to look for, pass a second argument as *nStart*. If the character is found in the string, this method returns the position of its first occurrence. If you want the get the last occurrence of the character, call the **ReverseFind()** instead. Its syntax is:

```
int ReverseFind(TCHAR ch) const;
```

These two versions of the **Find()** method are used to find a particular character in a string. Alternatively, you can specify a group of characters and try to find, at least, any one of them in the string. This operation is performed using the **FindOneOf()** method. Its syntax is:

```
int FindOneOf(LPCTSTR lpszCharSet) const;
```

The *lpszCharSet* argument is passed as a string which is a group of characters in any order. If the compiler finds any character of the *lpszCharSet* argument in the string, it returns its position. If none of the *lpszCharSet* characters is in the string, this method returns –1.

9.5.4 Character Identification

Every time the user types a character in a text-based control, this is referred to as a keystroke. When performing data entry, even if the user presses two keys simultaneously (to enter an uppercase letter or to type a special character), only one character is entered at a time. You can find out what character the user had entered in your application using appropriate functions. Some functions are used to categorize the types of characters on the keyboard. The functions used for these validations are as follows:

Function	Meaning
int isalpha(int c);	Returns true if c is an alphabetic character. Otherwise, returns false
int islower(int c);	Returns true if c is an alphabetic character in lowercase. Otherwise, returns false.
int isupper(int c);	Returns true if c is an alphabetic character in uppercase. Otherwise, returns false
int isdigit(int c);	Returns true if c is a digit. Otherwise, returns false
int isxdigit(int c);	Returns true if c is a hexadecimal digit. c must be one of the following 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, or F. Otherwise, returns false.
int isalnum(int c);	Returns true is c is between 0 and 9 or if c is between a and z or if c is between A and Z. Otherwise, returns false.
int isascii(int c);	Returns true if c is an ASCII character. Otherwise, returns false
int ispunct(int c);	Returns true if c is a punctuation character. Otherwise, returns false.
int isprint(int c);	Returns true if c is a printable character. Otherwise, returns false.
int isgraph(int c);	Returns true if c is a printable character and is not an empty space.
int isspace(int c);	Returns true if c is an empty space

9.5.5 Removing Characters

Besides the **CString::SetAt()** method, you can also use the **Replace()** method to replace a character of the string with another character. The syntax of the **Replace()** method is:

```
int Replace(TCHAR chOld, TCHAR chNew);
```

The **Replace()** method replaces each occurrence of the *chOld* argument. Whenever it finds it, it replaces it with the *chNew* character.

9.6 Sub Strings

9.6.1 Introduction

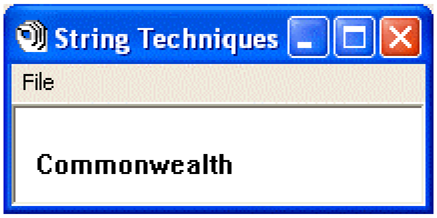
A sub string is a character or a group of characters that is part of another string. Therefore, to create a string, you must first have a string as basis.

When declaring and initializing a **CString** variable, although you can provide as many characters as posible, you may want the variable to actually hold only part of the string value. Based on this, to create a sub string when declaring a CString variable, you can use the following constructor:

```
CString(LPCTSTR lpch, int nLength);
```

When using this constructor, provide the whole string as a the null-terminated *lpch* string. To specify the number of characters for the sub string variable, provide an integer value for the *nLength* argument. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString CofV = "Commonwealth of Virginia";
	CString Sub(CofV, 12);

	pDC->TextOut(10, 20, Sub);
}
```



Commonwealth

As you may realize from this constructor, its sub string is built from the left side of the string and starts counting characters towards the right. Once it reaches the *nLength* number of characters, it creates the new string and ignores the characters beyond *nLength*.

If the string has already been initialized and you want to create a sub string made of some characters from its left side, you can call the **CString::Left()** method. Its syntax is:

```
CString Left(int nCount) const;
```

This method starts counting characters from the left to *nCount* characters and returns a new string made of those *nCount* characters. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
```

```
        CString Common = "Commonwealth of Virginia";
        CString Sub = Common.Left(12);

        pDC->TextOut(10, 20, Sub);
}
```

As the **Left()** method considers the characters from the left position, if you want the characters of the end of a string, you can call the **CString::Right()** to create a new string. Its syntax is:

```
CString Right( int nCount ) const;
```

Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
        CExerciseDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        CString Common = "Commonwealth of Virginia";
        CString Sub = Common.Right(8);

        pDC->TextOut(10, 20, Sub);
}
```

Alternatively, you can create a sub string using any range of characters from a string. To do this, call the CString::Mid() method whose syntaxes are:

```
CString Mid(int nFirst) const;
CString Mid(int nFirst, int nCount) const;
```

To create the sub string start from any character in the string, use the first version whose *nFirst* argument specifies where the new string would start. In this case, the method would return a string made of characters from the nFirst position to the last character of the string.

To create a sub string using a specific number of characters, use the second version. The nFirst argument specifies where the new string would start. The nCount argument is the number of characters that would be used from the main string to create the new string.

9.6.2 Finding a Sub String

To scan a string for a group of characters, you can call the following versions of the **CString::Find()** method:

```
int Find(LPCTSTR lpszSub) const;
int Find(LPCTSTR pstr, int nStart) const;
```

To find a sub string in a string, pass the desired sub string as the *lpszSub* argument. As done with the single character, you can specify the position to start looking by providing a second argument as *nStart* and use the fourth version of this method. If the sub string is found, this method returns the position of its first character in the string.

If the character or the sub string is not found, the method returns –1.

9.6.3 Character and String Removal

Besides the **CString(LPCTSTR *lpch*, int *nLength*)** constructor, the **Left(),** the **Mid(),** and the **Right()** method, another technique you can use to create a sub string consists of deleting characters from an existing string. This can be done by calling the **CString::Delete()** method. Its syntax is:

```
int Delete(int nIndex, int nCount = 1);
```

The *nIndex* argument specifies the first character of the range. If you pass only this argument, only the character at position *nIndex* will be deleted. If you want to delete more than one character, pass the desired number as the *nCount* argument. If the value of *nCount* is higher than the remaining number of characters from nIndex position, all characters to the right will be deleted.

If you want to remove all occurrences of a particular character in a string, you can call the **Remove()** method. Its syntax is:

```
int CString::Remove(TCHAR ch);
```

This method scans the string looking for the *ch* character. Whenever it finds it, it deletes it. This operation is performed using case sensitivity. This means that only the exact match with case of ch would be deleted.

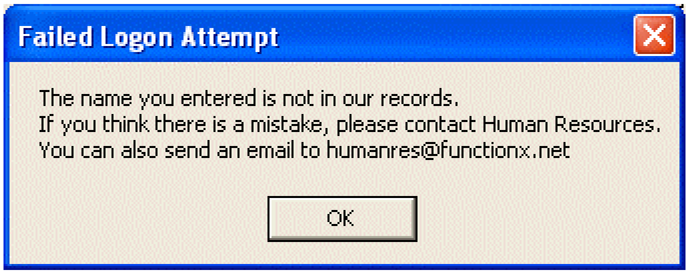
9.6.4 Replacing String Occurrences

Besides the **CString::SetAt()** method, you can also use the **Replace()** method to replace a character of the string with another character. The syntaxes of the **Replace()** method are:

```
int Replace(TCHAR chOld, TCHAR chNew);
int Replace(LPCTSTR lpszOld, LPCTSTR lpszNew);
```

The **Replace()** method replaces each occurrence of the *chOld* argument. Whenever it finds it, it replaces it with the *chNew* character. Here is an example:

```
void CExerciseDlg::OnMsgBox()
{
	// TODO: Add your control notification handler code here
	CString Msg("The name you entered is not in our records.\n"
			"If you think there is a mistake, please contact HR.\n"
			"You can also send an email to humanres@functionx.com");
	Msg.Replace("HR", "Human Resources");
	Msg.Replace(".com", ".net");

	MessageBox(Msg, "Failed Logon Attempt");
}
```


Failed Logon Attempt

The name you entered is not in our records.
If you think there is a mistake, please contact Human Resources.
You can also send an email to humanres@functionx.net

9.6.5 String Formatting

The **sprintf()** function we reviewed earlier is provided by the C/C++ language. The **CString** class provides its own function for formatting a string. This is done using the **CString::Format()** whose syntax are:

```
void Format(LPCTSTR lpszFormat, ...);
void Format(UINT nFormatID, ...);
```

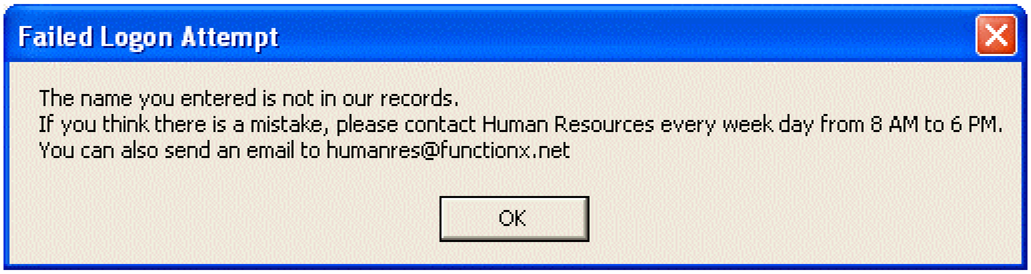
The *lpFormat* argument follows the exact same rules as the second argument to the **sprintf()** function seen earlier. The *nFormatID* argument is an identifier that can have been created using the String Table

The third and subsequent arguments follow the same rules as the … argument of the **sprintf()** function. There are many examples of the **CString::Format()** method in this book.

While the **sprintf()** function and the **CString::Format()** methods follow the order of the %Character combinations to format a message, the **CString** class provides a method that can also be used to format a string but it allows you to list the parameters if the order of your choice.

Here is an example:

```
void CExerciseDlg::OnMsgBox()
{
	// TODO: Add your control notification handler code here
	const char *DeptToContat = "Human Resources";
	const char  EmailContact[] = "humanres@functionx.net";
	const int   StartShift = 8;
	const int   EndShift   = 6;
	CString Msg;

	Msg.Format("The name you entered is not in our records.\n"
		         "If you think there is a mistake, please contact %s "
		         "every week day from %d AM to %d PM.\n"
		         "You can also send an email to %s",
		         DeptToContat, StartShift, EndShift, EmailContact);

	MessageBox(Msg, "Failed Logon Attempt");
}
```



The name you entered is not in our records.
If you think there is a mistake, please contact Human Resources every week day from 8 AM to 6 PM.
You can also send an email to humanres@functionx.net

9.7 Operations on Strings

9.7.1 About Constructing a String

The constructors we have used so far were meant to initialize a string when declaring it. Normally, if you do not have an initial value when declaring the string variable, you can use the default constructor:

```
CString();
```

The default constructor is used to declare an empty string that can be initialized later on. If thevariable already contains a value but you want it to be empty, you can call the **CString::Empty()** method. Its syntax is:

```
void Empty();
```

This method removes all characters, if there are any, from a string variable and restores the memory it was using. If you think the string contains empty spaces to its left side and you think this could compromise an operation on the string, to remove such empty spaces, call the **TrimLeft()** method whose syntax is:

```
void TrimLeft();
```

To remove empty spaces from the right side of a string, call the TrimRight() method. Its syntax is:

```
void TrimRight();
```

If at one time you want to find out whether a string is empty or not, call the CString::IsEmpty() method. Its syntax is:

```
BOOL IsEmpty() const
```

This method returns TRUE if the string is empty. Otherwise, it returns FALSE.

9.7.2 String Assignment

We saw earlier that you can declare an empty **CString** variable using the default constructor. To initialize it, you can assign it a string value of your choice. Here is an example:

```
void CExoView::OnDraw(CDC* pDC)
```

```
{
	CExoDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Term = "Information Superhighway";

	pDC->TextOut(20, 22, Term);
}
```

9.7.3 String Copy

Making a copy of a string consists of providing its value to another string. If you already have a **CString** initialized and you want to copy its value into another **CString** variable, you can use the following constructor:

CString(const CString& *stringSrc*);

This constructor is used to declare a CString variable and provide it with the value of another CString variable. Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString Term = "Information Superhighway";
	CString Description(Term);

	pDC->TextOut(20, 22, Description);

}
```

9.7.4 Strings and Their Cases

In US English, the following alphabetical letters are referred to as lowercase a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z. Their equivalents in uppercase are A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z. The digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and the other characters are considered “as is”.

To convert a lowercase CString variable to uppercase, you can call the **MakeUpper()** method. Its syntax is:

void MakeUpper();

When executed, this method would examine each character of the string. If a character is a lowercase letter, it would be converted to uppercase. If a character is already in uppercase, it be left in uppercase. If the character is not a letter, it would be left intact. On the other hand, to convert the characters of a string to lowercase, call the **MakeLower()** method. Its syntax is:

void MakeLower();

If a character is an uppercase letter, it would be converted to lowercase. If a character is already in lowercase, it be left in lowercase. The digits and other characters are left “as is”.

9.8 String Comparisons

9.8.1 Case Sensitivity

String comparison consists of finding out whether the values of two strings are identical or not. Toperform such a comparison, the **CString** class provides the **Collate()** method. Its syntax is:

```
int Collate(LPCTSTR lpsz) const;
```

The string that called this method is compared with the *lpsx* argument, character by character and with regards to the case of each combination of charcaters. To perform the comparison, this method refers to the Regional Settings of Control Panel on the user’s computer concerning the numeric, the date, the time, and the currency systems used

The comparison starts with the left character for most latin languages, including US English:

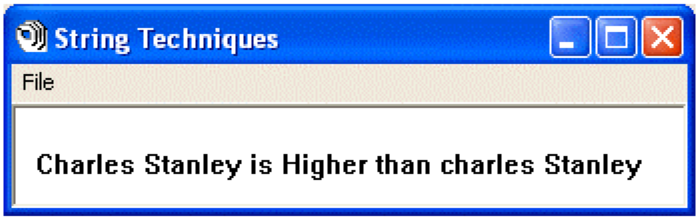
?? If the character at the first position ([0]) of the string (the string that called this method) has the same ANSI value as the first character at position [0] of the other string (the *lpsz* argument), the method skips this position and continues the comparison with the next position of characters. If all characters of the corresponding positions of both strings are exactly the same as set on the ANSI map, this method returns 0, meaning the strings are equal

?? If a character at one position x has an ANSI value higher than the corresponding character at the same position x on the other string, the method returns a positive value and stops the comparison

?? If a character at one position x has an ANSI value lower than the corresponding character at the same position x on the other string, the method returns a negative value and stops the comparison

Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString CS = "Charles Stanley";
	CString AS = "charles Stanley";

	int Result = CS.Collate(AS);

	if( Result > 0 )
		pDC->TextOut(10, 20, "Charles Stanley is Higher than charles Stanley");
	else if( Result == 0 )
		pDC->TextOut(10, 20, "Charles Stanley and charles Stanley are equal");
	else // if( Result < 0 )
		pDC->TextOut(10, 20, "Charles Stanley is Lower than charles Stanley");
}
```



Charles Stanley is Higher than charles Stanley

Besides the Collate() method, the CString class provides the Compare() method to perform a case-sensitive comparison of characters of two strings. Its syntax is:

int Compare(LPCTSTR *lpsz*) const;

This method compares the character at a position x to the character at the same position x of the other, *lpsz*, string. The approach used to perform the comparisons is the same with the difference that the Compare() method does not refer the user's Regional Settings.

9.8.2 Case Insensitivity

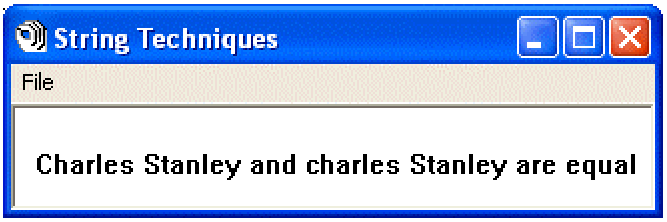
As seen above, the **Collate()** method considers the cases of characters during comparison and refers to the Regional Settings of the user. If you just want a simple comparison to find if two strings are identical regardless of the cases of their characters, you can call the **CString::CollateNoCase()** method. Its syntax is;

int CollateNoCase(LPCTSTR lpsz) const;

This member function considers that the alphabet is made of characters of a single case, implying that the character **a** is exactly the same as the character **A**. The other symbols and digits are considered "as is". It proceeds as follows:

?? The character at a position x of one string is compared with the corresponding character at the same position on the other string. If both characters are the same, the comparison continues with the next character. If all characters at the same positions on both strings are the same and the strings have the same length, this method returns 0, indicating that the strings are identical:

```
void CExerciseView::OnDraw(CDC* pDC)
{
    CExerciseDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    CString CS = "Charles Stanley";
    CString AS = "charles Stanley";

    int Result = CS.CollateNoCase(AS);

    if( Result > 0 )
            pDC->TextOut(10, 20, "Charles Stanley is Higher than charles Stanley");
    else if( Result == 0 )
            pDC->TextOut(10, 20, "Charles Stanley and charles Stanley are equal");
    else // if( Result < 0 )
            pDC->TextOut(10, 20, "Charles Stanley is Lower than charles Stanley");
}
```



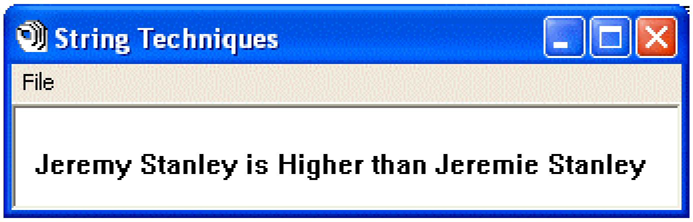
Charles Stanley and charles Stanley are equal

?? If a character at one position x has an ordered value higher than the corresponding character at the same position x on the other string, the method returns a positive value and stops the comparison

?? If a character at one position x has an ordered value lower than the corresponding character at the same position x on the other string, the method returns a negative value and stops the comparison

Here is an example:

```
void CExerciseView::OnDraw(CDC* pDC)
{
	CExerciseDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CString YS = "Jeremy Stanley";
	CString IS = "Jeremie Andy";

	int Result = YS.CollateNoCase(IS);

	if( Result > 0 )
		pDC->TextOut(10, 20, "Jeremy Stanley is Higher than Jeremie Stanley");
	else if( Result == 0 )
		pDC->TextOut(10, 20, "Jeremy Stanley and Jeremie Stanley are equal");
	else // if( Result < 0 )
		pDC->TextOut(10, 20, "Jeremy Stanley is Lower than Jeremie Stanley");
}
```



Jeremy Stanley is Higher than Jeremie Stanley

Alternatively, to perform case-insensitive string comparison on CString variables, you can use the CompareNoCase() method. Its syntax is:

```
int CompareNoCase(LPCTSTR lpsz) const;
```

The Comp areNoCase() method does not refer to the user's Regional Settings when performing its comparison.

Besides the **CollateNoCase()** and the **CompareNoCase()** methods, the **CString** class provides an easier and probably more familiar means of performing value comparisons. Character insensitive comparisons on **CString** variables can be performed using Boolean operators ==, !=, <, <=, >, >=. Any of these operators can be applied on two CString variables or a combination of a **CString** and a null-terminated string variables. The operators are overloaded as follows:

```
BOOL operator ==( const CString& s1, const CString& s2 );
BOOL operator ==( const CString& s1, LPCTSTR s2 );
BOOL operator ==( LPCTSTR s1, const CString& s2 );
BOOL operator !=( const CString& s1, const CString& s2 );
BOOL operator !=( const CString& s1, LPCTSTR s2 );
BOOL operator !=( LPCTSTR s1, const CString& s2 );
BOOL operator <( const CString& s1, const CString& s2 );
BOOL operator <( const CString& s1, LPCTSTR s2 );
BOOL operator <( LPCTSTR s1, const CString& s2 );
BOOL operator >( const CString& s1, const CString& s2 );
BOOL operator >( const CString& s1, LPCTSTR s2 );
BOOL operator >( LPCTSTR s1, const CString& s2 );
BOOL operator <=( const CString& s1, const CString& s2 );
BOOL operator <=( const CString& s1, LPCTSTR s2 );
BOOL operator <=( LPCTSTR s1, const CString& s2 );
BOOL operator >=( const CString& s1, const CString& s2 );
BOOL operator >=( const CString& s1, LPCTSTR s2 );
BOOL operator >=( LPCTSTR s1, const CString& s2 );
```

These operators work exactly as you are used to dealing with conditional statements except that the operands must be string values.

Chapter 10: Characteristics of a Window's Frame

✍ Overview of Windows Frames

✍ Window Creation

10.1 Introduction to Win32 Library

10.1.1 Overview

Win32 is a library made of data types, variables, constants, functions, and classes (mostly structures) that can be used to create applications for the Microsoft Windows operating systems. A typical application is made of at least two objects: a control and a host object on which the control is positioned.

To create a Win32 application using Microsoft Visual C++, display the New (5 and 6 versions) or New Project (7 version) dialog box and select Win32 application (5 and 6) or Win32 Project (7) item.

Just like a C++ program always has a main() function, a Win32 program uses a central function called WinMain. The syntax of that function is:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow );
```

Unlike the C++ main() function, the arguments of the WinMain() function are not optional. Your program will need them to communicate with the operating system.

To create an application using the Microsoft Foundation Class (MFC) library, we have seen that we can create a class derived from CWinApp. The CWinApp class implements the role of the WinMain() function. To provide functionality as complete as possible, this class is equipped with appropriate variables and methods.

Pratical Learning: Exploring the Win32 Library

1. Start Microsoft Visual C++ or Visual Studio and display either the New dialog box and its Projects property page (MSVC 6) or the New Project dialog box (MSVC 7)
2. Set the Project Type as either Win32 Application or Win32 Project
3. Set the name as Win32B and click OK
4. Set the project as An Empty Project for a Windows Application and click Finish
5. Create a C++ Source File named Exercise
6. In the empty file, type:

```
#include <windows.h>

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine, int nCmdShow )
{
    return 0;
}
```

7. Save All

10.1.2 The Framework

Instead of creating an application using "raw" Win32 classes and functions, the document/view architecture simplifies this process by providing a mechanism called the framework. The framework is a set of classes, functions, and techniques used to create an application as complete as possible with as few lines of code as possible. To provide all this functionality, the framework works behind the scenes with the CWinApp class to gather the necessary MFC classes and functions, to recognize and reconcile the Win32 classes that the application needs. This reconciliation is also made possible by a set of global functions. These functions have names that start with Afx... Some of these functions are:

?? AfxFormatString1
?? AfxFormatString2
?? AfxMessageBox
?? AfxFreeLibrary
?? AfxGetApp
?? AfxGetAppName
?? AfxGetInstanceHandle
?? AfxGetMainWnd
?? AfxGetResourceHandle
?? AfxInitRichEdit
?? AfxLoadLibrary
?? AfxMessageBox
?? AfxRegisterWndClass
?? AfxSocketInit
?? AfxSetResourceHandle
?? AfxRegisterClass
?? AfxBeginThread
?? AfxEndThread
?? AfxGetThread
?? AfxWinInit

So far, we have seen that, after an application is created by deriving a class from **CWinApp**, you must declare a global variable of your application class to make it available to the rest of your application. An example of declaring that variable is:

```
CExerciseApp theApp;
```

To get a pointer to this variable from anywhere in your application, call the **AfxGetApp()** function. Its syntax is:

```
CWinApp* AfxGetApp();
```

To implement the role of the Win32's **WinMain()** function, the framework uses its own implementation of this function and the MFC provides it as **AfxWinInit().** It is declared as follows:

```
BOOL AFXAPI AfxWinInit(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow);
```

As you can see, the Win32's **WinMain()** and the MFC's **AfxWinInit()** functions use the same arguments.

10.1.3 A Window's Instance

When you start an application such as Notepad, you are said to have created an instance of the application. In the same way, when you declare a variable of a class, an instance of the class is created and made available to the project. The **WinMain()** function also allows you to create an instance of an application, referred to as the hInstance argument of the **WinMain()** function. The instance is created as an **HINSTANCE**.

The **CWinApp** class provides a corresponding instance variable called **m_hInstance**. This variable can let you get a handle to the instance of your application. Alternatively, to get a handle to the instance of your application, you can call the **AfxGetInstanceHandle()** global function. Its syntax is:

```
HINSTANCE AfxGetInstanceHandle();
```

Even more, to get a handle to your application, you can call the Win32 API's **GetWindowLong()** function.

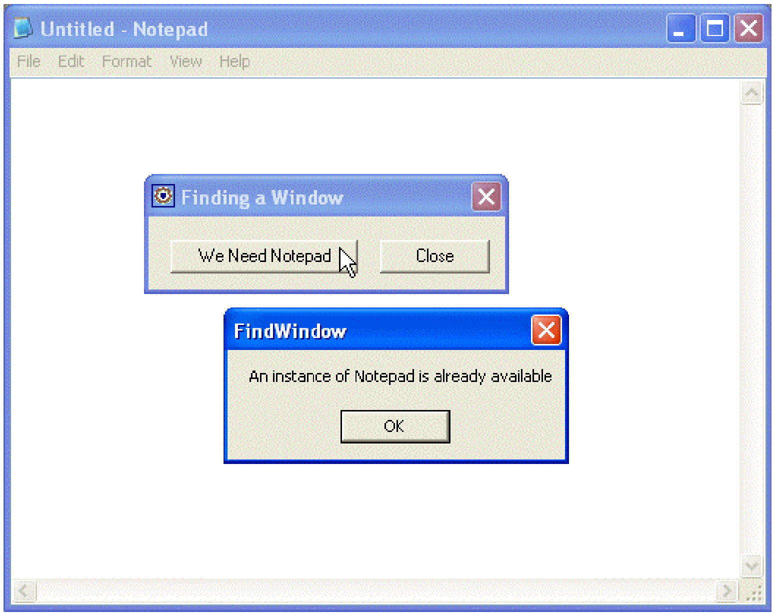
Suppose you have opened Notepad to view the source code of an HTML document. This is said that you have an instance of Notepad. Imagine that you want to open a text document using Notepad without closing the first instance of Notepad. To do this, you must open another copy of Notepad. This second copy of Notepad is another instance. In this case, the first instance is referred to as a previous instance. For a Win32 application, the previous instance would be the hPrevInstance argument of the **WinMain()** function. For a Win32 application, the hPrevInstance argument always has the NULL value. If you want to find out whether a previous instance of an application already exists, you can call the **CWnd::FindWindow()** method. Its syntax is:

```
static CWnd* PASCAL FindWindow(LPCTSTR lpszClassName, LPCTSTR lpszWindowName);
```

If you created the window or if it is a window you know for sure, in which case it could be a **WNDCLASS** or **WNDCLASSEX** object, specify it as the *lpszClassName* argument. If you do not know its name with certainty, set this argument as NULL.

The lpszWindowName argument is the possible caption of the window you are looking for. Imagine you position a button on a dialog box and you want the user to launch Notepad with that button and imagine that, if Notepad is already opened, there would be no reason to create another instance of it. In the following code, when the user clicks the button, the application checks if Notepad is already opened and displays a message accordingly:

```
void CFindWindowDlg::OnFindNotepad()
{
	// TODO: Add your control notification handler code here
	CWnd *ExistsAlready = FindWindow(NULL, "Untitled - Notepad");

	if( ExistsAlready )
	   AfxMessageBox("An instance of Notepad is already available");
	else
	   AfxMessageBox("No instace of Notepad could be found");
}
```



An instance of Notepad is already available

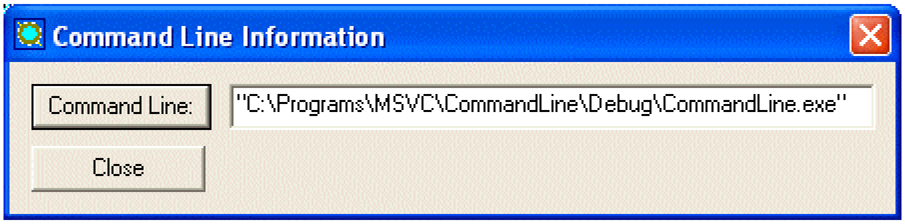
10.1.4 The Command Line

To execute a program, you must communicate its path and possibly some additional parameters to the compiler. This information is called the command line information and it is supplied as a string. You need to keep that in mind although all programs of this book will be compiled inside of Visual C++. The command line information is supplied to the compiler as the *lpCmdLine* argument of the **WinMain()** function. Internally, Visual C++ creates the path and communicates it to the compiler when you execute the program. If you want to find out what command line was used to execute your program, you can call the Win32's **GetCommandLine()** function. Its syntax is:

```
LPTSTR GetCommandLine(VOID);
```

This function takes no argument but returns the command line of an application as null-terminated string. Here is an example:

```
void CCommandLineDlg::OnBtnCmdLine()
{
	// TODO: Add your control notification handler code here
	char CmdLine[80];
	char CmdResult[80];

	strcpy(CmdLine, GetCommandLine());
	sprintf(CmdResult, "%s", CmdLine);
	m_CommandLine.Format("%s", CmdResult);

	UpdateData(FALSE);
}
```



"C:\Programs\MSVC\CommandLine\Debug\CommandLine.exe"

10.1.5 Frame Display Options

The *nCmdShow* argument of the **WinMain()** function specifies whether and how you want to display the window when the user attempts to open it. This is a constant value that is actually passed to a function that is in charge of displaying the window.

Its possible values are:

Value	Description
SW_SHOW	Displays a window and makes it visible
SW_SHOWNORMAL	Displays the window in its regular size. In most circumstances, the operating system keeps track of the last location and size a window such as Internet Explorer or My Computer had the last time it was displaying. This value allows the OS to restore it.
SW_SHOWMINIMIZED	Opens the window in its minimized state, representing it as a button on the taskbar
SW_SHOWMAXIMIZED	Opens the window in its maximized state
SW_SHOWMINNOACTIVE	Opens the window but displays only its icon. It does not make it active
SW_SHOWNA	As previous
SW_SHOWNOACTIVATE	Retrieves the window's previous size and location and displays it accordingly
SW_HIDE	Used to hide a window
SW_MINIMIZE	Shrinks the window and reduces it to a button on the taskbar
SW_MAXIMIZE	Maximizes the window to occupy the whole screen area
SW_RESTORE	If the window was minimized or maximized, it would be restored to its previous location and size

One of the ways you can use this value is to pass it to the WinExec() Win32 function which can be used to open an application. The syntax of this function is:

```
UINT WinExec(LPCSTR lpCmdLine, UINT nCmdShow);
```

The lpCmdLine argument is a null-terminated string that specifies either the name of the application or its complete path.

In the following example, the SW_MAXIMIZE nCmdShow value is passed to the WinExec() function to open Solitaire maximized:

```
void CWindowDlg::OnOpenSolitaire()
{
	WinExec("SOL.EXE", SW_MAXIMIZE);
}
```

10.1.6 Window Class Initialization

A win32 application is built using either the **WNDCLASS** or the **WNDCLASSEX** classes.

The WNDCLASS class is defined as follows:

```
typedef struct _WNDCLASS {
    UINT       style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

If you are creating an MFC application, you can declare a **WNDCLASS** variable in your frame constructor. Here is an example:

```
#include <afxwin.h>

// The application class
class CExerciseApp : public CWinApp
{
public:
            // Used to instantiate the application
            BOOL InitInstance();
};

// The class that displays the application's window
// and gives it "physical" presence (Real Estate)
class CMainFrame : public CFrameWnd
{
public:
            // The window class will be created in this constructor
            CMainFrame();
};

CMainFrame::CMainFrame()
{
            // Declare a window class variable
            WNDCLASS WndCls;
}

BOOL CExerciseApp::InitInstance()
{
            // Initialize the main window object
            m_pMainWnd = new CMainFrame();

            // Hoping everything is fine, return TRUE
            return TRUE;
}

// The global application object
CExerciseApp theApp;
```

Upon declaring a **WNDCLASS** variable, the compiler allocates an amount of memory space for it, as it does for all other variables. If you think you will need more memory than allocated, assign the number of extra bytes to the cbClsExtra member variable. Otherwise, the compiler initializes this variable to 0. If you do not need extra memory for your **WNDCLASS** variable, initialize this member with 0. If you are creating an MFC application, you can omit initializing the cbClsExtra member variable. Otherwise, you can do it as follows:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;

        WndCls.cbClsExtra      = 0;
}
```

Creating an application, as we saw earlier, is equivalent to creating an instance for it. To communicate to the WinMain() function that you want to create an instance for your application, which is, to make it available as a resource, assign the WinMain()'s hInstance argument to your **WNDCLASS** variable. We saw earlier that, to get an instance for your application, you can call the AfxGetInstanceHandle(). You can use the return value of this function to initialize the hInstance member variable of your **WNDCLASS** object:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;

        WndCls.cbClsExtra          = 0;
        WndCls.hInstance     = AfxGetInstanceHandle();
}
```

If you omit doing this, the framework would initialize it with the main instance of the application. For this reason, you do not have to initialize the WNDCLASS::hInstance variable.

When an application has been launched and is displaying on the screen, which means an instance of the application has been created, the operating system allocates an amount of memory space for that application to use. If you think that your application's instance would need more memory than that, you can request that extra memory bytes be allocated to it. Otherwise, you can let the operating system handle this instance memory issue and initialize the *cbWndExtra* member variable to 0. For an MFC application, if you want to specify the amount of extra memory your application's instance would need, assign the desired number the same way:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;

        WndCls.cbClsExtra    = 0;
        WndCls.cbWndExtra    = 0;
        WndCls.hInstance     = AfxGetInstanceHandle();
}
```

The style member variable specifies the primary operations applied on the window class. The actual available styles are constant values. For example, if a user moves a window or changes its size, you would need the window to be redrawn to get its previous characteristics. To redraw the window horizontally, you would apply the **CS_HREDRAW**. In the same way, to redraw the window vertically, you can apply the **CS_VREDRAW**.

The styles are combined using the bitwise operator OR (|). The **CS_HREDRAW** and the **CS_VREDRAW** styles can be combined and assigned to the style member variable as follows:

```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;

	WndCls.style          = CS_VREDRAW | CS_HREDRAW;
	WndCls.cbClsExtra     = 0;
	WndCls.cbWndExtra     = 0;
	WndCls.hInstance      = AfxGetInstanceHandle();
}
```

On a regular basis, while the application is running, its controls will receive instructions from the user. This happens when the user clicks a mouse button or presses a keyboard keys. These actions produce messages that must be sent to the operating system to do something. Since there can be various messages for different reasons at any time, the messages are processed in a global function pointer called a window procedure. To define this behavior, you can create a pointer to function, also called a callback function. In this case, the function must return a 32-bit value specially intended for window procedures. It is called **LRESULT**. The name of the function is not important but it must carry some required pieces of information that make a message relevant and complete. For a Win32 application, the message must provide the following four pieces of information:

?? The control that sent the message: Every object you will need in your program, just like everything in the computer, must have a name. The operating system needs this name to identify every object, for any reason. An object in Microsoft Windows is identified as a Handle. For Win32 controls, the handle is called **HWND**

?? The type of message: The object that sends a message must let the operating system know what message it is sending. As we saw in Lesson 4 on Messages, there are various types of messages for different circumstances. Nevertheless, to make matters a little easier, we saw that each message is a constant positive natural number (unsigned int) identified with a particular name. The message identifier is passed as **UINT**

?? Accompanying items: Because there are so many types of messages, you must provide two additional pieces of information to help process the message. These two items depend on the type of message and could be different from one type of message to another. The first accompanying item is a 32-bit type (unsigned int) identified as **WPARAM**. The second accompanying item is a 32-bit type of value (long) identified as **LPARAM**. Remember that these two can be different things for different messages. For a Win32 application, the messages can be carried in a function defined as follows:

```
LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam);
```

For a Win32 application, the *hWnd* argument is required because it specifies what Windows control sent the message. On an MFC application, the class that manages the controls knows what control sent the message, which means that you do not have to specify the window handle. Therefore, the window procedure would be declared as follows, omitting the **HWND** object because it is specified by the window that is sending the message:

```
virtual LRESULT WindowProcedure(UINT message, WPARAM wParam, LPARAM lParam);
```

To process the messages, and because there can be so many of them, the window procedure typically uses a switch control to list all necessary messages and process each one in turn (some of the messages are those we reviewed in Lesson 4). After processing a message, its case must return a value indicating whether the message was successfully processed or not and how the message was processed.

Regardless of the number of messages you process, there will still be messages that you did not deal with. It could be because they were not sent even though they are part of the Windows control(s) used in your application. If you did not process some messages, you should/must let the operating system take over and process it. This is done because the operating system is aware of all messages and it has a default behavior or processing for each one of them. Therefore, you should/must return a value for this to happen. The value returned is typically placed in the default section of the switch condition and must simply be a DefWindowProc() function. For a Win32 application, its syntax is:

```
LRESULT DefWindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

For an MFC application, the syntax used for this function is:

```
virtual LRESULT DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam);
```

This function is returned to Windows, saying "There are messages I couldn't process. Do what you want with them". The operating system would simply apply a default processing to them. The values returned by the DefWindowProc() function should be the same passed to the procedure.

The most basic message you can process is to make sure a user can close a window after using it. This can be done with a function called **PostQuitMessage().** Its syntax is:

```
VOID PostQuitMessage(int nExitCode);
```

This function takes one argument which is the value of the **LPARAM** argument. To close a window, you can pass the argument as **WM_QUIT**.

The name of the window procedure must be assigned to the *lpfnWndProc* member variable of the **WNDCLASS** variable.

Because we are using MFC to visually build our applications, you usually will not need to define a window procedure to process Windows messages, unless the control you are using is lacking a message that you find relevant. The Windows controls we will use in this book have messages and notifications that apply the most regular behaviors they need to offer. If you do not process all messages of a control, which will happen most of the time, their default behavior are part of the **AfxWndProc** procedure. Therefore, you can simply assign it to the lpfnWndProc member variable of your **WNDCLASS** variable:

```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;

	WndCls.style         = CS_VREDRAW | CS_HREDRAW;
	WndCls.lpfnWndProc   = AfxWndProc;
	WndCls.cbClsExtra    = 0;
	WndCls.cbWndExtra    = 0;
```

```
        WndCls.hInstance    = AfxGetInstanceHandle();
}
```

In Lesson 3, we saw that an icon can be used to represent an application in My Computer or Windows Explorer. To assign this small picture to your application, you can either use an existing icon or design your own. To make your programming a little faster, Microsoft Windows installs a few icons. The icon is assigned to the hIcon member variable using the **LoadIcon()** function. For a Win32 application, the syntax of this function is:

```
HICON LoadIcon(HINSTANCE hInstance, LPCTSTR lpIconName);
```

The hInstance argument is a handle to the file in which the icon was created. This file is usually stored in a library (DLL) of an executable program. If the icon was created as part of your application, you can use the hInstance of your application. If your are using one of the icons below, set this argument to NULL.

The *lpIconName* is the name of the icon to be loaded. This name is added to the resource file when you create the icon resource. It is added automatically if you add the icon as part of your resources; otherwise you can add it manually when creating your resource script. Normally, if you had created and designed an icon and gave it an identifier, you can pass it using the **MAKEINTRESOURCE** macro.

To make your programming a little faster, Microsoft Windows installs a few icons you can use for your application. These icons have identification names that you can pass to the LoadIcon() function as the lpIconName argument. The icons are:

ID	Picture
IDI_APPLICATION	□
IDI_INFORMATION	ℹ
IDI_ASTERISK	ℹ
IDI_QUESTION	?
IDI_WARNING	⚠
IDI_EXCLAMATION	⚠
IDI_HAND	✖
IDI_ERROR	✖

If you designed your own icon (you should make sure you design a 32x32 and a 16x16 versions, even for convenience), to use it, specify the hInstance argument of the LoadIcon() function to the instance of your application. Then use the MAKEINTRESOURCE macro to convert its identifier to a null-terminated string. This can be done as follows:

```
WndCls.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_STAPLE));
```

If you are creating an MFC application, to use a standard icon, you can call the CWinApp::LoadIcon() method. It is provided in two versions as follows:

```
HICON LoadIcon(LPCTSTR lpszResourceName) const;
HICON LoadIcon(UINT nIDResource) const;
```

The icon can be specified by its name, which would be a null-terminated string passed as *lpszResourceName*. If you had designed your icon and gave it an ID, you can pass this identifier to the **LoadIcon()** method.

The **LoadIcon()** member function returns an HICON object that you can assign to the hIcon member variable of your **WNDCLASS** object. Here is an example:

```
CMainFrame::CMainFrame()
{
        WNDCLASS WndCls;

        WndCls.style          = CS_VREDRAW | CS_HREDRAW;
        WndCls.lpfnWndProc    = AfxWndProc;
        WndCls.cbClsExtra     = 0;
        WndCls.cbWndExtra     = 0;
        WndCls.hInstance      = AfxGetInstanceHandle();
        WndCls.hIcon          = LoadIcon(NULL, IDI_WARNING);
}
```

You can also declare an HICON handle and initialize it with CWinApp::LoadStandardIcon() method as follows:

```
AfxGetApp()->LoadStandardIcon(StdIcon);
```

A cursor is used to locate the position of the mouse pointer on a document or the screen. To use a cursor, call the Win32 LoadCursor() function. Its syntax is:

```
HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursorName);
```

The hInstance argument is a handle to the file in which the cursor was created. This file is usually stored in a library (DLL) of an executable program. If the cursor was created as part of your application, you can use the hInstance of your application. If your are using one of the below cursors, set this argument to NULL.

When Microsoft Windows installs, it also installs various standard cursors you can use in your program. Each one of these cursors is recognized by an ID which is simply a constant integers. The available cursors are:

ID	Picture	Description
IDC_APPSTARTING	↖⌛	Used to show that something undetermined is going on or the application is not stable
IDC_ARROW	↖	This standard arrow is the most commonly used cursor
IDC_CROSS	+	The crosshair cursor is used in various circumstances such as drawing
IDC_HAND	☝	The Hand is standard only in Windows 2000. If you are using a previous operating system and need this cursor, you may have to create your own.
IDC_HELP	↖?	The combined arrow and question mark cursor is used when providing help on a specific item on a window object
IDC_IBEAM	I	The I-beam cursor is used on text-based object to show the position of the caret
IDC_ICON		This cursor is not used anymore
IDC_NO	⊘	This cursor can be used to indicate an unstable situation
IDC_SIZE		This cursor is not used anymore
IDC_SIZEALL	✥	The four arrow cursor pointing north, south, east, and west is highly used to indicate that an object is selected or that it is ready to be moved
IDC_SIZENESW	⤢	The northeast and southwest arrow cursor can be used when resizing an object on both the length and the height
IDC_SIZENS	↕	The north - south arrow pointing cursor can be used when shrinking or

		heightening an object
IDC_SIZENWSE	⤡	The northwest - southeast arrow pointing cursor can be used when resizing an object on both the length and the height
IDC_SIZEWE	↔	The west - east arrow pointing cursor can be used when narrowing or enlarging an object
IDC_UPARROW	↑	The vertical arrow cursor can be used to indicate the presence of the mouse or the caret
IDC_WAIT	⌛	The Hourglass cursor is usually used to indicate that a window or the application is not ready

To use one of these cursors, if you are creating an MFC application, you can call the CWinApp::LoadCursor() method to assign one of the above standard cursors to your application. This method comes in two versions as follows:

```
HCURSOR LoadCursor(LPCTSTR lpszResourceName) const;
HCURSOR LoadCursor(UINT nIDResource) const;
```

The cursor can be specified using its name, which would be a null-terminated string passed as lpszResourceName. If you had designed your cursor and gave it an ID, you can pass this identifier to the LoadCursor() method.

The LoadCursor() member function returns an HCURSOR value. You can assign it to the hCursor member variable of your WNDCLASS object. Here is an example:

```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;

	WndCls.style         = CS_VREDRAW | CS_HREDRAW;
	WndCls.lpfnWndProc   = AfxWndProc;
	WndCls.cbClsExtra    = 0;
	WndCls.cbWndExtra    = 0;
	WndCls.hInstance     = AfxGetInstanceHandle();
	WndCls.hIcon         = LoadIcon(NULL, IDI_WARNING));
	WndCls.hCursor       = LoadCursor(NULL, IDC_CROSS);
}
```

You can also call the CWinApp::LoadStandardCursor() method using the AfxGetApp() function. Its syntax is:

```
HCURSOR LoadStandardCursor(LPCTSTR lpszCursorName) const;
```

To paint the work area of the window, you must specify what color will be used to fill it. This color is created as an **HBRUSH** and assigned to the hbrBackground member variable of your WNDCLASS object. The color you are using must be a valid **HBRUSH** or you can cast a known color to **HBRUSH**. The Win32 library defines a series of colors known as stock objects. To use one of these colors, call the **GetStockObject()** function. For example, to paint the windows background in black, you can pass the **BLACK_BRUSH** constant to the GetStockObject() function, cast it to **HBRUSH** and assign the result to hbrBackground.

In addition to the stock objects, the Microsoft Windows operating system provides a series of colors for its own internal use. These are the colors used to paint the borders of frames, buttons, scroll bars, title bars, text, etc. The colors are named (you should be able to predict their appearance or role from their name):

COLOR_ACTIVEBORDER,COLOR_ACTIVECAPTION, COLOR_APPWORKSPACE, COLOR_BACKGROUND, COLOR_BTNFACE, COLOR_BTNSHADOW, COLOR_BTNTEXT, COLOR_CAPTIONTEXT, COLOR_GRAYTEXT, COLOR_HIGHLIGHT, COLOR_HIGHLIGHTTEXT, COLOR_INACTIVEBORDER, COLOR_INACTIVECAPTION, COLOR_MENU, COLOR_MENUTEXT, COLOR_SCROLLBAR, COLOR_WINDOW, COLOR_WINDOWFRAME, and COLOR_WINDOWTEXT.

To use one of these colors, cast it to **HBRUSH** and add 1 to its constant to paint the background of your window:

```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;
	const char *StrWndName = "Windows Fundamentals";

	WndCls.style         = CS_VREDRAW | CS_HREDRAW;
	WndCls.lpfnWndProc   = AfxWndProc;
	WndCls.cbClsExtra    = 0;
	WndCls.cbWndExtra    = 0;
	WndCls.hInstance     = AfxGetInstanceHandle();
	WndCls.hIcon         = AfxGetApp()->LoadStandardIcon(IDI_WARNING);
	WndCls.hCursor       = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
	WndCls.hbrBackground = (HBRUSH)(COLOR_ACTIVECAPTION+1);
	WndCls.hCursor       = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
}
```

To get the value of a system color, call the **GetSysColor()** function. Its syntax is:

```
DWORD GetSysColor(int nIndex);
```

The *nIndex* argument should be a valid name of one of the system color constants such as **COLOR_ACTIVECAPTION**. When this function has executed, it returns the **COLORREF** value of the *nIndex* color. If you provide a wrong or unrecognized value as the *nIndex* argument, this function returns 0, which is also a color and can therefore produce an unexpected result. If you want to consider only existing valid colors, call the **GetSysColorBrush()** function instead. Its syntax is:

```
HBRUSH GetSysColorBrush( int nIndex);
```

This function returns the color value of the system color that is passed as *nIndex*. If the value of *nIndex* is not valid, the function returns NULL, which is not 0 and therefore is not a color, producing a more predictable result.

Practical Learning: Building a Window Class

1. Change the contents of the file as follows:

```
#include <windows.h>
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
                         WPARAM wParam, LPARAM lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                 LPSTR lpCmdLine, int nCmdShow )
{
      WNDCLASSEX WndClsEx;

      WndClsEx.cbSize           = sizeof(WNDCLASSEX);
      WndClsEx.style            = CS_HREDRAW | CS_VREDRAW;
      WndClsEx.lpfnWndProc      = WndProc;
      WndClsEx.cbClsExtra       = 0;
      WndClsEx.cbWndExtra       = 0;
      WndClsEx.hInstance        = hInstance;
      WndClsEx.hIcon            = LoadIcon(NULL, IDI_APPLICATION);
      WndClsEx.hCursor          = LoadCursor(NULL, IDC_CROSS);
      WndClsEx.hbrBackground  = (HBRUSH)(COLOR_BACKGROUND + 1);
      WndClsEx.hIconSm          = LoadIcon(NULL, IDI_APPLICATION);

      return 0;
}
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
                         WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.1.7 Window Registration

After initializing the window class, you must make it available to the other controls that will be part of your application. This process is referred to as registration. If you are creating a Win32 application using the **WNDCLASS** structure, to register the window class, call the **RegisterClass().** If you created your window class using the **WNDCLASSEX** structure, call the **RegisterClassEx()** function. Their syntaxes are:

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);
ATOM RegisterClassEx(CONST WNDCLASSEX *lpwcx);
```

The function simply takes as argument a pointer to a **WNDCLASS** or **WNDCLASSEX**.

If you are working on an MFC application, to register your window class, call the **AfxRegisterWndClass()** function. Its syntax is:

```
LPCTSTR AFXAPI AfxRegisterWndClass(UINT nClassStyle, HCURSOR hCursor = 0,
                                   HBRUSH hbrBackground = 0, HICON hIcon = 0);
```

This function expects the window class, the cursor to use to indicate the position of the mouse, the color to paint the background, and an icon that represents the application.

These are the same values used to initialize the window class. Using these values as initialized above, you can register the window class as follows:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;

        WndCls.style          = CS_VREDRAW | CS_HREDRAW;
        WndCls.lpfnWndProc    = AfxWndProc;
        WndCls.cbClsExtra     = 0;
        WndCls.cbWndExtra     = 0;
        WndCls.hInstance      = AfxGetInstanceHandle();
        WndCls.hIcon          = AfxGetApp()->LoadStandardIcon(IDI_WARNING);
        WndCls.hCursor        = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
        WndCls.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
        WndCls.hCursor        = AfxGetApp()->LoadStandardCursor(IDC_CROSS);

        AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                  WndCls.hbrBackground, WndCls.hIcon);
}
```

Practical Learning: Registering a Window

1. Just above the return line of the WinMain() function, register the class using the RegisterClassEx() function:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow )
{
    WNDCLASSEX WndClsEx;

    . . .

    WndClsEx.hIconSm      = LoadIcon(NULL, IDI_APPLICATION);

    RegisterClassEx(&WndClsEx);

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2 Window Creation

10.2.1 The Main Window

The **WNDLCLASS** and the **WNDCLASSEX** classes are only used to initialize the application window class. To display a window, that is, to give the user an object to work with, you must create a window object. This window is the object the user uses to interact with the computer.

If you are creating a Win32 application, to create a window, you can call either the **CreateWindow()** or the **CreateWindowEx()** function. Their syntaxes are:

```
HWND CreateWindow(
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName,
  DWORD dwStyle,
  int x,
  int y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  LPVOID lpParam
);
```

```
HWND CreateWindowEx(
  DWORD dwExStyle,
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName,
  DWORD dwStyle,
  int x,
  int y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  LPVOID lpParam
);
```

You can simply call this function and specify its arguments after you have registered the window class. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                    LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASS      WndCls;

        . . .

        RegisterClass(&WndCls);

        CreateWindow(. . .);
}
```

If you are planning to use the window further in your application, you should retrieve the result of the CreateWindow() or the CreateWindowEx() function, which is a handle to the window that is being created. To do this, you can declare an HWND variable and initialize it with the create function. This can be done as follows:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                    LPSTR lpCmdLine, int nCmdShow)
{
        HWND   hWnd;
        WNDCLASS      WndCls;

        . . .

        RegisterClass(&WndCls);

        hWnd = CreateWindow(. . .);
}
```

If you are working on an MFC application, you can derive a class from CFrameWnd, as we have done so far. Each window that can display on the computer screen is based on a class equipped with a method called Create. The structure of this method is different from one type of window to another. Nevertheless, because all window objects are based on CWnd, the CWnd class provides the primary functionality used by all the other window controls.

The process of creating a frame window in an MFC application is done by using the class' constructor and calling its **Create()** method. As we have seen in the past, the main window object you must create is a frame because it gives presence to your application. The most basic frame is created using the **CFrameWnd** class. We have learned that this is done by deriving your own class from CFrameWnd. Therefore, in the constructor of your **CFrameWnd**-derived class, call the **Create()** method. As a reminder from Lesson 2, the syntax of the **CFrameWnd::Create()** method is as follows:

```
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 );
```

If the method succeeds in creating the window, it returns TRUE. If it fails, it returns FALSE.

Practical Learning: Initiating Window Creation

1. Declare a handle to a window object as **HWND** and initialize it with the **CreateWindowEx()** function:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow )
{
    HWND      hWnd;
    WNDCLASSEX WndClsEx;

    . . .

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx();

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.2 The Window Class Name

To create a window, you must provide its name as everything else in the computer has a name. There are two main types of class names you will use in your applications. If the window you are creating is a known type of window, its class is already registered with the operating system. In this case you can provided it. If you are creating your own fresh class, which means you are in charge of its characteristics (properties), then define a null-terminated string as the class' name.

For a Win32 application, the class name of your main window must be provided to the *lpszClassName* member variable of your **WNDCLASS** or **WNDCLASSEX** variable. You can provide the name to the variable or declare a global null-terminated string. The name must also be passed to the *lpClassName* argument of either the **CreateWindow()** or the **CreateWindowEx()** functions. Here is an example:

If you are creating an MFC application, the class name is passed as the first argument of the **CFrameWnd::Create()** method. You can use a null terminated string as done for the **CreateWindow()** or the **CreateWindowEx()** function. If you have initialized the window class and registered it using **AfxRegisterWndClass()**, you may remember that this function returns a null-terminated string. Therefore, you can pass its return value to the **Create()** method. This can be done as follows:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;

        . . .

        const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                                   WndCls.hbrBackground, WndCls.hIcon);

        Create(StrClass, ;
}
```

Once an application is created, although you can, you should refrain from ever changing the name of a class. It may take more than simply assigning a new value to the **AfxRegisterWndClass()** function.

Practical Learning: Naming a Window Class

1. To provide a name for the window being created, declare a null-terminated string variable. Initialize the lpszClassName member variable of your window application and pass it to the CreateWindowEx() function:

```
#include <windows.h>
//---------------------------------------------------------------------------
char StrClassName[] = "Win32Exercise";
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
                                         WPARAM wParam, LPARAM
lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow )
{
    HWND      hWnd;
    WNDCLASSEX WndClsEx;

    . . .
    WndClsEx.hbrBackground = (HBRUSH)(COLOR_ BACKGROUND + 1);
```

```
        WndClsEx.lpszClassName = StrClassName;
        WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        RegisterClassEx(&WndClsEx);

        hWnd = CreateWindowEx(0,
                              StrClassName, );

        return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.3 The Window Name

We saw in Lesson 2 that every window should have a name to easily identify it. For a main window, the name displays on the title bar of the frame.

The name is passed as the lpWindowName argument of either the **CreateWindow()** or the **CreateWindowEx()** functions. To do this, you can provide a null-terminated string to the argument or declare a global string. Here is an example:

```
const char *ClsName = "WndFund";
const char *WndName = "Application Name";

LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
        WNDCLASS     WndCls;

        . . .

        RegisterClass(&WndCls);

        CreateWindow(ClsName, WndName,
}
```

If you are creating an MFC application, to provide a name for the window, pass a null-terminated string as the second argument of the CFrameWnd::Create() method:

```
CMainFrame::CMainFrame()
{
        WNDCLASS WndCls;
        const char *StrWndName = "Application Name";

        . . .

        const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                                   WndCls.hbrBackground, WndCls.hIcon);

        Create(StrClass, StrWndName);
}
```

Practical Learning: Setting the Window Name

1. To provide a name for the window, declare and initialize a null-terminated string and pass its value as the lpWindowName argument of the CreateWindowEx() function:

```
#include <windows.h>
//---------------------------------------------------------------------------
char StrClassName[] = "Win32Exercise";
char StrWndName[]   = "Simple Win32 Application";
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
                                              WPARAM wParam, LPARAM
lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
               LPSTR lpCmdLine, int nCmdShow )
{
    HWND       hWnd;
    WNDCLASSEX WndClsEx;

    . . .

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx(0,
                          StrClassName,
                          StrWndName, );

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.4 Windows Styles

We had a formal introduction to windows styles in Lesson 2 and we reviewed all necessary styles to apply or not apply to a main window. Once again, a **WS_OVERLAPPEDWINDOW** has a caption that displays the window name (if any). It is also equipped with the system menu, a thick frame, a system Minimize button, a system Maximize button, and a system Close button.

For a Win32 application, you can apply the **WS_OVERLAPPEDWINDOW** style as follows:

```
CreateWindow(ClsName, WndName, WS_OVERLAPPEDWINDOW,
```

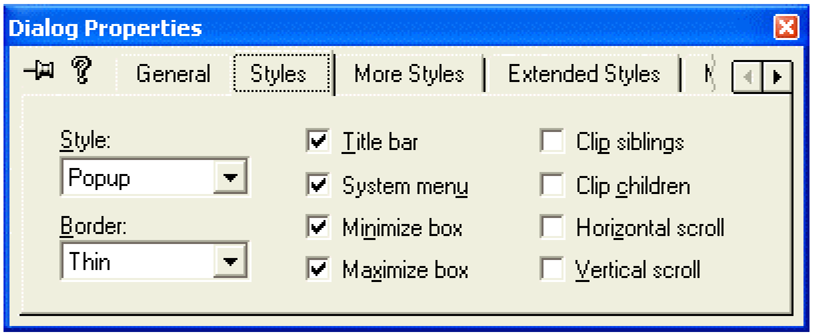
For an MFC application, this style can be added as follows:

```
CMainFrame::CMainFrame()
{
         // Declare a window class variable
         WNDCLASS WndCls;
         const char *StrWndName = "Windows Fundamentals";
```

```
        . . .

        const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                                        WndCls.hbrBackground, WndCls.hIcon);

        Create(StrClass, StrWndName, WS_OVERLAPPEDWINDOW);
}
```

Remember that, to apply a combination of styles, use the bitwise OR operator.

If you are designing a form or a dialog box, you can use the Properties window to visually select the styles you want to apply to the window:



Style: Popup

Border: Thin

☑ Title bar
☑ System menu
☑ Minimize box
☑ Maximize box
☐ Clip siblings
☐ Clip children
☐ Horizontal scroll
☐ Vertical scroll

Practical Learning: Creating an Overlapped Window

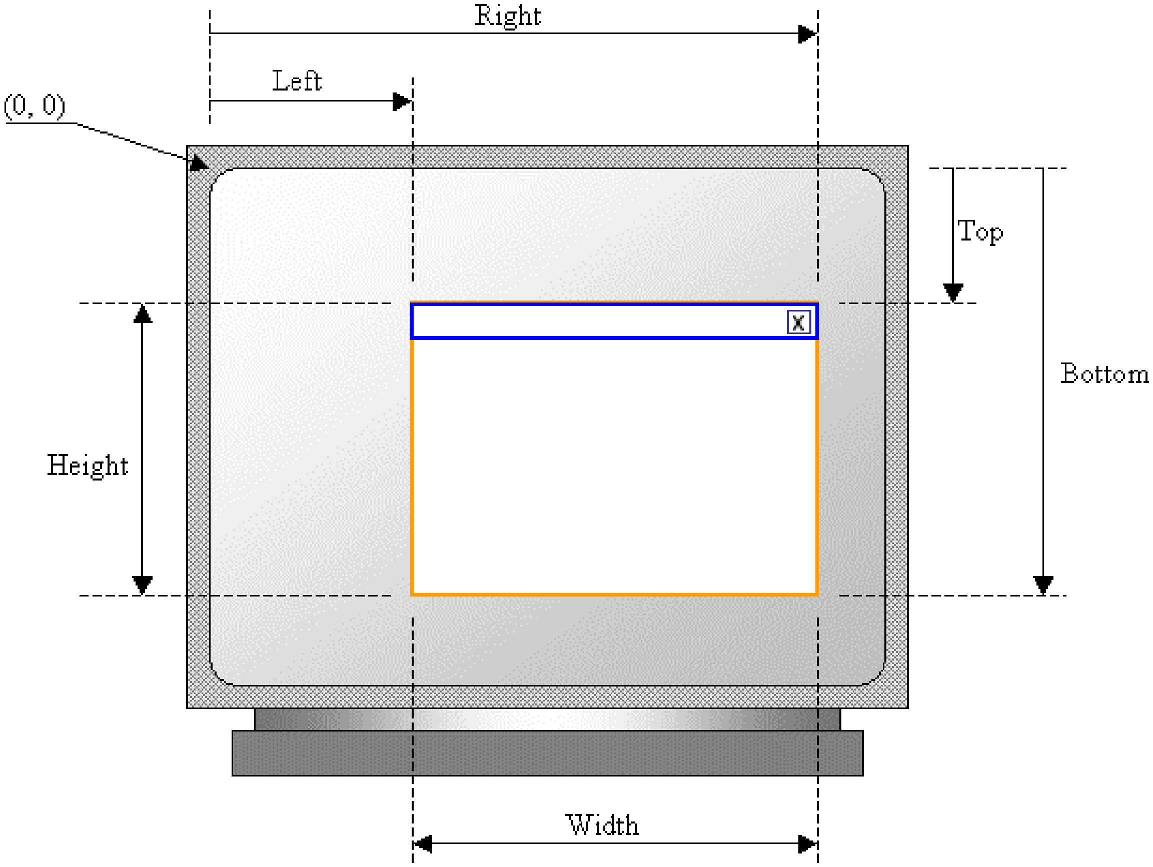
1. To create borders and a title bar for the window, apply the **WS_OVERLAPPEDWINDOW** style as follows:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow )
{
    . . .

    hWnd = CreateWindowEx(0,
                          StrClassName,
                          StrWndName,
                          WS_OVERLAPPEDWINDOW, );

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save all

10.2.5 Window's Location and Size

As we learned in Lesson 2, the location of a window is defined by the distance from the left border of the monitor to the window's left border and its distance from the top border of the monitor to its own top border. The size of a window is its width and its height. These can be illustrated for a main window frame as follows:



For a Win32 application, the original distance from the left border of the monitor is passed as the x argument to the **CreateWindow()** or the **CreateWindowEx()** function. The distance from top is specified using the y argument. The x and y arguments define the location of the window. The distance from the left border of the monitor to the right border of the window is specified as the nWidth argument. The distance from the top border of the monitor to the lower border of the window is specified with the nHeight value.

If you cannot make up your mind for these four values, you can use the **CW_USEDEFAULT** (when-Creating-the-Window-USE-the-DEFAULT-value) constant for either one or all four arguments. In such a case, the compiler would select a value for the argument.

For an MFC application, when calling the **CFrameWnd::Create()** method, the location and size of the frame window is specified as a RECT or CRect rectangle as we saw in Lesson 2. As done with the **CW_USEDEFAULT** constant, you can let the compiler decide on the rectangle by passing the rect argument as rectDefault. Here is an example:

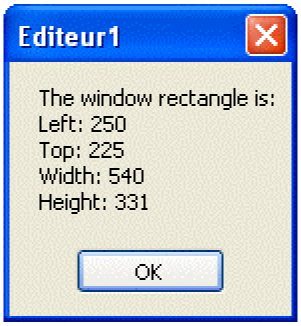
```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;
	const char *StrWndName = "Windows Fundamentals";

	. . .

	const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                                WndCls.hbrBackground, WndCls.hIcon);

          Create(StrClass, StrWndName, WS_OVERLAPPEDWINDOW, rectDefault);
}
```

At any time you can find out the location and size of a rectangle by calling the **CWnd::GetWindowRect()** method. Its syntax is:

```
void GetWindowRect(LPRECT lpRect) const;
```

To use this method, pass it a **RECT** or a **CRect** variable as *lpRect*. The method returns the rectangle properties of the window. Here is an example:

```
void CMainFrame::OnViewLocationandsize()
{
          // TODO: Add your command handler code here
          CRect Recto;
          GetWindowRect(&Recto);

          char Str[80];
          sprintf(Str, "The window rectangle is:\nLeft: %d\nTop:
                  %d\nWidth: %d\nHeight: %d",
                  Recto.left, Recto.top, Recto.Width(), Recto.Height());
          MessageBox(Str);
}
```



The window rectangle is:
Left: 250
Top: 225
Width: 540
Height: 331

If you created your application using AppWizard, it sets a default rectangle for the frame (actually the left and top values are randomly selected). Whether you created the frame using the **CFrameWnd::Create()** method or AppWizard, you can redefine its location and size from the PreCreateWindow() event. Because this event is called after CFrameWnd::Create but before the window is displayed, its values are applied to the window. The syntax of the **CWnd::PreCreateWindow()** event is (this event is inherited from CWnd but CFrameWnd, like any other class that needs it, overrides it and provides its own functionality as it relates to a window frame):

```
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

This event takes as argument a **CREATESTRUCT** object. This structure is defined as follows:

```
typedef struct tagCREATESTRUCT {
   LPVOID     lpCreateParams;
   HANDLE     hInstance;
   HMENU      hMenu;
   HWND       hwndParent;
```

```
  int       cy;
  int       cx;
  int       y;
  int       x;
  LONG      style;
  LPCSTR    lpszName;
  LPCSTR    lpszClass;
  DWORD     dwExStyle;
} CREATESTRUCT;
```

As you can see, the member variables of this structure are very similar to the arguments of the Win32's CreateWindow() and CreateWindowEx() functions. Its member variables correspond as follows:

CREATESTRUCT	CreateWindow and CreateWindowEx	Meaning
x	x	Distance from left border of monitor to left border of window frame
Y	y	Distance from top border of monitor to top border of window frame
cx	nWidth	Distance from left border of monitor to right border of window frame
cy	nHeight	Distance from top border of monitor to bottom border of window frame

Therefore, in your **CFrameWnd::PreCreateWindow()** event, assign the desired values for the location and size of the window. In the following example, the original dimensions of the window are set to 450x325 (width x height):

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreateWindow(cs) )
		return FALSE;
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs
	cs.cx = 450;
	cs.cy = 325;

	return TRUE;
}
```

Of course, you can also use the **PreCreateWindow()** event to customize the appearance of the window frame.

Once a regular window, that is, an overlapped window, has been created, it displays a title bar, its system buttons, and borders. As stated already, if you created an application using AppWizard, the window may appear in a random location, which you can control. If you want the window to be positioned in the center of the screen, call the **CWnd::CenterWindow()** method. Its syntax is:

```
void CenterWindow(CWnd* pAlternateOwner = NULL);
```

By default, this window positioned the caller (the window that called it) in the middle-center of the main window, also called its owner. For a main frame of a window, the owner would be the monitor screen. As long as the window is owned by another, the compiler can find out and position it accordingly. For this reason, you can omit the pAlternateOwner argument.

You should call the CenterWindow() method in the event that creates the window. For a frame this would be the OnCreate event. Here is an example:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;
        // . . .

        CenterWindow();

        return 0;
}
```

If you want the caller to be centrally positioned relative to another window, obtain a handle to that window and pass it as the pAlternateOwner argument.

Once a window is displaying on the screen, the user can change its location by dragging its title bar. This would change its **CREATESTRUCT**::x or its rect::left argument. If you do not want the user to change the location of a window, one thing you can do is to prevent the mouse from capturing, that is, taking ownership, of the title bar. This can be done calling the Win32's **ReleaseCapture()** function. Its syntax is:

```
BOOL ReleaseCapture(VOID);
```

When this function is called, the event in which it is accessed prevents the mouse from capturing the object on which the mouse is positioned. Nevertheless, if the function succeeds, it returns TRUE. If for some reason it fails, it returns FALSE. Because a **WM_MOVE** message is sent when a window is moved, you can use it to call this function. Here is an example:

```
void CMainFrame::OnMove(int x, int y)
{
        CFrameWnd::OnMove(x, y);

        ReleaseCapture();
        // TODO: Add your message handler code here
}
```

To change the dimensions of a window, the user can click and drag one of its borders. Imagine you do not want the user to change the size of the window. If you remove the system buttons and/or the system menu(

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        if( !CFrameWnd::PreCreateWindow(cs) )
                return FALSE;
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs
        cs.style &= ~WS_SYSMENU;

        return TRUE;
}
```

), the user is still able to resize it. One solution you can use is to call the **ReleaseCapture()** function on the **WM_SIZE** event. That way, every time the user grabs a border and starts dragging, the mouse would lose control of the border:

```
void CMainFrame::OnSize(UINT nType, int cx, int cy)
{
	CFrameWnd::OnSize(nType, cx, cy);

	ReleaseCapture();
	// TODO: Add your message handler code here
}
```

Another way the user change the size of a window consists of minimizing it. As you surely know already, to minimize the window, the user clicks the system Minimize button 🗕. If at two time you want to find out whether a window is minimized, you can call the **CWnd::IsIconic()** method. Its syntax is:

```
BOOL IsIconic( ) const;
```

This method returns TRUE if the window is minimized. Otherwise it returns FALSE.

If you do not want the user to be able to minimize a wndow, you can omit the **WS_MINIMIZEBOX** style when creating the window.

If you want the user to be able to maximize the window, add the **WS_ MAXIMIZEBOX** style. This can be done as follows:

```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;
	const char *StrWndName = "Windows Fundamentals";

	. . .

	const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
				WndCls.hbrBackground, WndCls.hIcon);

	Create(StrClass, StrWndName,
		WS_OVERLAPPED | WS_CAPTION |
		WS_SYSMENU | WS_THICKFRAME |
		WS_MAXIMIZEBOX, rectDefault);
}
```

7.

At any time, to find out if the window is maximized, you can call the **CWnd::IsZoomed()** method. Its syntax is:

```
BOOL IsZoomed( ) const;
```

This method returns TRUE if the window is maximized. Otherwise, it returns FALSE.

If you created the application using AppWizard, all system buttons are added to the frame. If you want to remove a style, use the **CFrameWnd::PreCreateWindow()** event. In the following example, the system Minimize button is removed on the frame:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        if( !CFrameWnd::PreCreateWindow(cs) )
                return FALSE;
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs

        cs.style &= ~WS_MINIMIZEBOX;

        return TRUE;
}
```

In the same way, you can remove the system maximize button.

If you allow the system buttons, the user can minimize and maximize the window at will. To do this, besides using the system buttons, the user can also double-click the title bar of the window or click its Maximize menu item either from the window's system icon or after right-clicking its button on the Taskbar. When the window is maximized, the frame is resized to occupy the whole monitor area, starting from the top-left corner as the origin.

When the user decides to maximize a window, the frame sends a **WM_GETMINMAXINFO** message which appears like "The user wants to maximize me. What should I do?". This message produces the **OnGetMinMaxInfo()** event. If the application does not give any specific instruction, the window gets maximized. If you want to do something before the window is formally maximized, you can intercept **OnGetMinMaxInfo()** event. Remember that, if you want to prevent the window from being minimized or maximized, you should remove the Minimize (**WS_MINIMIZEBOX**) or the Maximize (**WS_MAXIMIZEBOX**) buttons from its style.

If you want to control the minimum and/or maximum size that a window can have when the user wants to maximize it, set the desired size in the **OnGetMinMaxInfo()** event. It syntax is:

```
afx_msg void OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI);
```

The information pertinent to a window's maximized size, including the maximum size it can have, is stored in a structure called MINMAXINFO. The MINMAXINFO structure is defined as follows:

```
typedef struct tagMINMAXINFO {
   POINT ptReserved;
   POINT ptMaxSize;
   POINT ptMaxPosition;
   POINT ptMinTrackSize;
   POINT ptMaxTrackSize;
} MINMAXINFO;
```

A variable of **MINMAXINFO** type is passed to the **OnGetMinMaxInfo()** event.

The *ptReserved* member variable is reserved and never used.

The ptMaxSize value is the maximum width and the maximum height. To specify this value, you can call the x member variable of the point value of ptMaxSize and assign the desired value. Here is an example:

```
void CMainFrame::OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI)
{
	// TODO: Add your message handler code here and/or call default
	lpMMI->ptMaxSize.x = 450;
	lpMMI->ptMaxSize.y = 325;

	CFrameWnd::OnGetMinMaxInfo(lpMMI);
}
```

The *ptMaxPosition* member variable is the location of the window when it gets maximized. Normally, when the window gets maximized, it starts occupying the monitor screen from the top-left corner. The ptMaxPosition allows you to change that default behavior. The ptMaxPosition.x is the distance from the left border of the monitor to the left border of the maximized window. The ptMaxPosition.y is the distance from the top border of the monitor to the top border of the maximized window.

The *ptMinTrackSize* member variable is the minimum size the window must assume when it gets maximized. Normally, when a window frame gets maximized, it occupies the whole screen area. This member variable can be used to control that size. The ptMinTrackSize.x is the minimum width. The *ptMinTrackSize*.y is the minimum height.

The *ptMaxTrackSize* member variable is the maximum size the window must have when it is maximized. The *ptMaxTrackSize*.x is the minimum width. The *ptMaxTrackSize*.y is the minimum height.

Practical Learning: Setting the Window Location and Dimensions

1. Apply the default location and dimensions to the window with the CW_USEDEFAULT constant as follows:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow )
{
      . . .

      hWnd = CreateWindowEx(0,
                                   StrClassName,
                                   StrWndName,
                                   WS_OVERLAPPEDWINDOW,
                                   CW_USEDEFAULT,
                                   CW_USEDEFAULT,
                                   CW_USEDEFAULT,
                                   CW_USEDEFAULT, );

      return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.6 Window's Parenting

If the window you are creating has a parent, obtain its handle and pass it as the *hWndParent* argument of the CreateWindow() or the CreateWindowEx() functions for a

Win32 application. If you are creating the main window of your application, it does not have a parent. In this case, pass the hWndParent as NULL.

If you are creating an MFC application, pass the handle of the parent as the pParentWnd argument of the **Create()** method. For a main frame, the kind we have created so far, specify this argument as NULL:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;
        const char *StrWndName = "Windows Fundamentals";

        . . .

        Create(StrClass, StrWndName, WS_OVERLAPPEDWINDOW, rectDefault, NULL);
}
```

Practical Learning: Parenting the Application

1. Since the current window does not have a parent or owner, pass the NULL constant as the parent:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow )
{
    . . .

    hWnd = CreateWindowEx(0,
                          StrClassName,
                          StrWndName,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL, );

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.7 The Window's Menu

If you want the window to display a menu, first create or design the resource menu. Visual C++ provides a convenient way to design a basic menu. To do this, display the Add Resource dialog box and select Menu. Then add the desired items.

After creating the menu, assign its name to the *lpszMenuName* name to your **WNDCLASS** variable for a Win32 variable. If you had created the menu and gave it an identifier, use the **MAKEINTRRESOURCE** macro to convert it. If the window you are creating is a child window that will need to display its particular menu when that child window is displaying, create its menu and pass its handle as the hMenu argument of the

CreateWindow() or **CreateWindowEx()** functions. Otherwise, pass this argument as NULL.

If you are creating an MFC application and want the main window to display a menu, design the menu as you see fit and create its resource. Then pass the menu name as the lpszMenuName argument of the Create() method. If you want to use the identifier of the menu, pass its ID to the **MAKEINTRESOURCE** macro. This can be done as follows:

```
CMainFrame::CMainFrame()
{
	// Declare a window class variable
	WNDCLASS WndCls;
	const char *StrWndName = "Windows Fundamentals";

	WndCls.style         = CS_VREDRAW | CS_HREDRAW;
	WndCls.lpfnWndProc   = AfxWndProc;
	WndCls.cbClsExtra    = 0;
	WndCls.cbWndExtra    = 0;
	WndCls.hInstance     = AfxGetInstanceHandle();
	WndCls.hIcon         = AfxGetApp()->LoadStandardIcon(IDI_WARNING);
	WndCls.hCursor       = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
	WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
	WndCls.hCursor       = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
	WndCls.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);

	const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
				WndCls.hbrBackground, WndCls.hIcon);

	Create(StrClass, StrWndName, WS_OVERLAPPEDWINDOW, rectDefault,
		NULL, MAKEINTRESOURCE(IDR_MENU1));
}
```

To make sure that the window was created, you can check its return value.

If the **CreateWindow()** or the **CreateWindowEx()** functions of a Win32 application succeeds in creating the window, it returns a handle to the window that was created. You can check this validity using an if conditional statement. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                   LPSTR lpCmdLine, int nCmdShow)
{
   HWND          hWnd;
   WNDCLASS      WndCls;

   . . .

   RegisterClass(&WndCls);

   hWnd = CreateWindow(ClsName,
                 WndName,
                 WS_OVERLAPPEDWINDOW,
                 CW_USEDEFAULT,
                 CW_USEDEFAULT,
                 CW_USEDEFAULT,
                 CW_USEDEFAULT,
                 NULL,
                 NULL,
                 hInstance,
                    NULL);

        if( !hWnd ) // If the window was not created,
                return 0; // stop the application
}
```

For an MFC application, a **Create()** method returns TRUE if the window was created. Otherwise, it returns FALSE. Therefore, you can use an if conditional statement to find out whether the window was successfully created or not. Here is an example:

```
CMainFrame::CMainFrame()
{
        // Declare a window class variable
        WNDCLASS WndCls;
        const char *StrWndName = "Windows Fundamentals";

        WndCls.style         = CS_VREDRAW | CS_HREDRAW;
        WndCls.lpfnWndProc   = AfxWndProc;
        WndCls.cbClsExtra    = 0;
        WndCls.cbWndExtra    = 0;
        WndCls.hInstance     = AfxGetInstanceHandle();
        WndCls.hIcon         = AfxGetApp()->LoadStandardIcon(IDI_WARNING);
        WndCls.hCursor       = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
        WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
        WndCls.hCursor       = AfxGetApp()->LoadStandardCursor(IDC_CROSS);
        WndCls.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);

        const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                                    WndCls.hbrBackground, WndCls.hIcon);

        if( !Create(StrClass, StrWndName, WS_OVERLAPPEDWINDOW, rectDefault,
          NULL, MAKEINTRESOURCE(IDR_MENU1)) )
                  return;
}
```

Practical Learning: Finalizing Window Creation

1. To finalize the creation of the window, change the **WinMain()** function as follows:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow )
{
    HWND       hWnd;
    WNDCLASSEX WndClsEx;

    WndClsEx.cbSize           = sizeof(WNDCLASSEX);
    WndClsEx.style            = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc      = WndProc;
    WndClsEx.cbClsExtra       = 0;
    WndClsEx.cbWndExtra       = 0;
    WndClsEx.hInstance        = hInstance;
    WndClsEx.hIcon            = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor          = LoadCursor(NULL, IDC_CROSS);
    WndClsEx.hbrBackground    = (HBRUSH)( COLOR_BACKGROUND + 1);
    WndClsEx.lpszMenuName     = NULL;
    WndClsEx.lpszClassName = StrClassName;
    WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx(0,
                          StrClassName,
                          StrWndName,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInstance,
                          NULL);
    if( !hWnd )
        return 0;

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.8 Window Display

Once a window has been created and if this was done successfully, you can display it to the user. This is done by calling the ShowWindow() function. The Win32 version of this function has the following syntax:

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

The *hWnd* argument is a handle to the window that you want to display. It could be the window returned by the **CreateWindow()** or the **CreateWindowEx()** function.

The *nCmdShow* specifies how the window must be displayed. It uses one of the *nCmdShow* values above.

To show its presence on the screen, the window must be painted. This can be done by calling the **UpdateWindow()** function. Its syntax is:

```
BOOL UpdateWindow(HWND hWnd);
```

This function simply wants to know what window needs to be painted. This window is specified by its handle.

Practical Learning: Displaying the Window

1. To show the window, call the **ShowWindow()** and the **UpdateWindow()** functions as follows:

```
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow )
{
    HWND      hWnd;
    WNDCLASSEX WndClsEx;

    . . .

    if( !hWnd )
        return 0;

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return 0;
}
//---------------------------------------------------------------------------
```

2. Save All

10.2.9 Considering Window's Messages

Once a window has been created, the user can use it. This is done by the user clicking things with the mouse or pressing keys on the keyboard. A message that a window sends is received by the application. This application must analyze, translate, and decode the message to know what object sent the message and what the message consists of. To do this, the application uses the GetMessage() function. Its syntax is:

```
BOOL GetMessage(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT
wMsgFilterMax);
```

The *lpMsg* argument is a pointer to the **MSG** structure.

The hWnd argument identifies which window sent the message. If you want the messages of all windows to be processed, pass this argument as NULL.

The wMsgFilterMin and the wMsgFilterMax arguments specify the message values that will be treated. If you want all messages to be considered, pass each of them as 0.

The **MSG** structure is defined as follows:

```
typedef struct tagMSG {
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG, *PMSG;
```

Once a message has been sent, the application analyzes it using the **TranslateMessage()** function. Its syntax is:

```
BOOL TranslateMessage(CONST MSG *lpMsg);
```

This function takes as argument the MSG object that was passed to the **GetMessage()** function and analyzes it. If this function successfully translates the message, it returns TRUE. If it cannot identify and translate the message, it returns FALSE.

Once a message has been decoded, the application must send it to the window procedure. This is done using the **DispatchMessage()** function. Its syntax is:

```
LRESULT DispatchMessage(CONST MSG *lpMsg);
```

This function also takes as argument the **MSG** object that was passed to **GetMessage()** and analyzed by **TranslateMessage().** This DispatchMessage() function sends the lpMsg message to the window procedure. The window procedure processes it and sends back the result, which becomes the return value of this function. Normally, when the window procedure receives the message, it establishes a relationship with the control that sent the message and starts treating it. By the time the window procedure finishes with the message, the issue is resolved (or aborted). This means that, by the time the window procedure returns its result, the message is not an issue anymore. For this reason you will usually, if ever, not need to retrieve the result of the **DispatchMessage()** function.

This translating and dispatching of messages is an on-going process that goes on as long as your application is running and as long as somebody is using it. For this reason, the application uses a while loop to continuously check new messages. This behavior can be implemented as follows:

```
while( GetMessage(&Msg, NULL, 0, 0) )
{
	TranslateMessage(&Msg);
	DispatchMessage(&Msg);
}
```

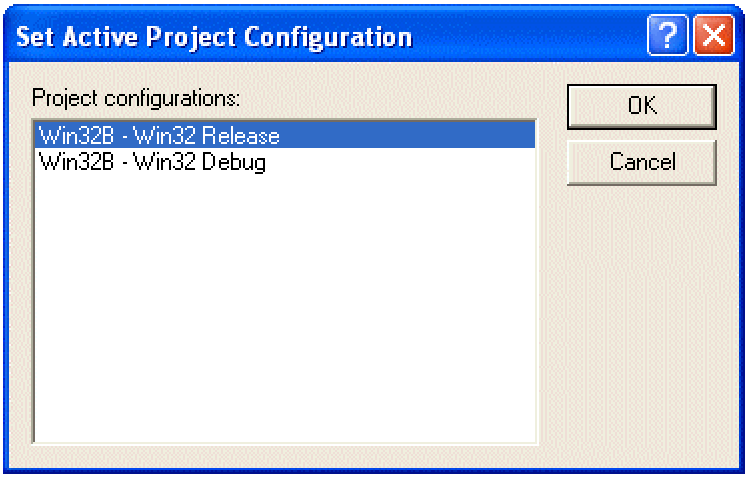
If the **WinMain()** function successfully creates the application and the window, it returns the *wParam* value of the **MSG**used on the application.

Practical Learning: Completing the Application

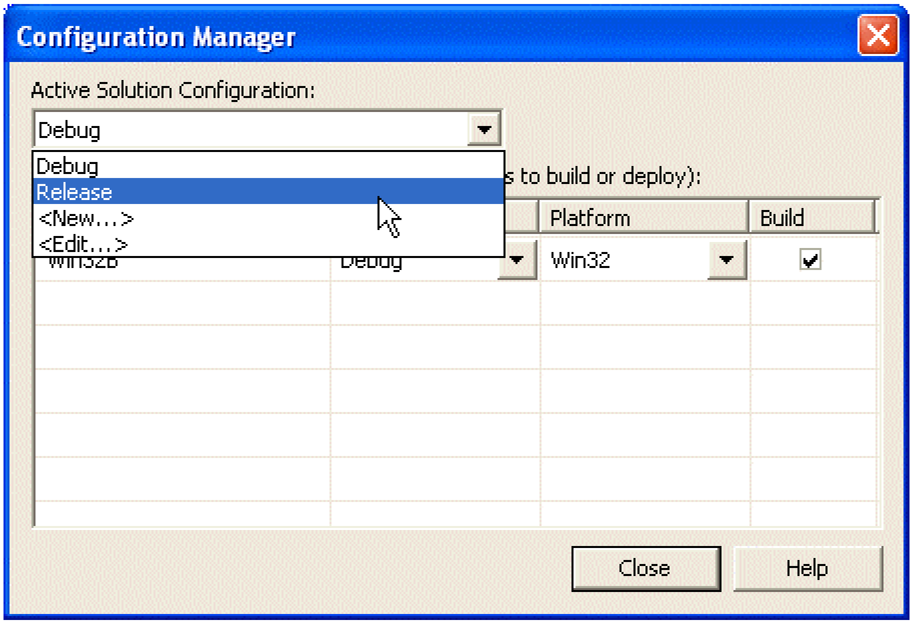
1. To finalize the application, complete it as follows:

```
#include <windows.h>
//---------------------------------------------------------------------------
char StrClassName[] = "Win32Exercise";
char StrWndName[]   = "Simple Win32 Application";
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
                         WPARAM wParam, LPARAM lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow )
{
    MSG         Msg;
    HWND        hWnd;
    WNDCLASSEX WndClsEx;

    WndClsEx.cbSize        = sizeof(WNDCLASSEX);
    WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc   = WndProc;
    WndClsEx.cbClsExtra    = 0;
    WndClsEx.cbWndExtra    = 0;
    WndClsEx.hInstance     = hInstance;
    WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor       = LoadCursor(NULL, IDC_CROSS);
    WndClsEx.hbrBackground = (HBRUSH)( COLOR_BACKGROUND + 1);
    WndClsEx.lpszMenuName  = NULL;
    WndClsEx.lpszClassName = StrClassName;
    WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx(0,
                          StrClassName,
                          StrWndName,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInstance,
                          NULL);
    if( !hWnd )
        return 0;

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    while( GetMessage(&Msg, NULL, 0, 0) )
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }

    return 0;
}
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
   switch(Msg)
   {
   case WM_DESTROY:
      PostQuitMessage(WM_QUIT);
      break;
   default:
      return DefWindowProc(hWnd, Msg, wParam, lParam);
   }
   return 0;
}
//---------------------------------------------------------------------------
```

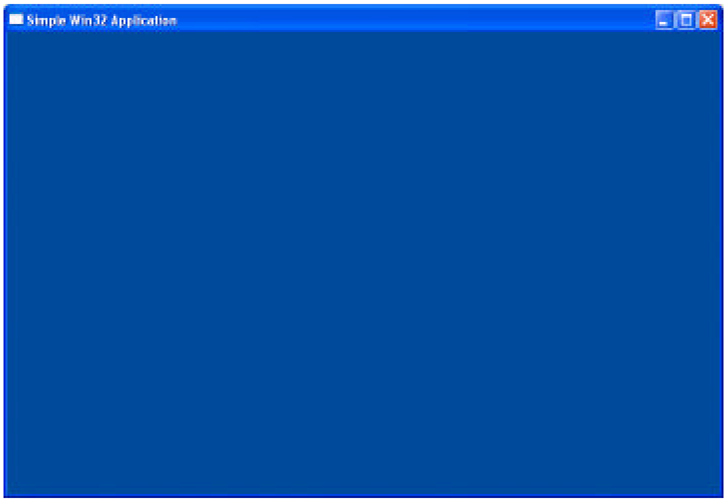
2. To test the application, if you are using MSVC 6, on the main menu, click Build -> Set Active Configuration and, on the Set Active Project Configuration dialog box, click Win32B Win32 Release:



If you are using MSVC 7, on the main menu, click Build -> Configuration Manager. The Configuration Manager dialog box, click the arrow of the Active Solution Configuration combo box and select Release



3. Click OK or Close
4. Execute the application



5. Return the MSVC and save everything

10.3 The Mini Frame Window

10.3.1 Introduction

A mini frame is a window that is mostly used as a floating object that accompany another window the main object of an application. It appears as a normal frame window we have seen so far with borders and a client area but a mini frame window is used a tool instead of as a main window. For this functionality, it does not have the System minimize and maximize buttons. As a tool window, it appears with a short title bar and is equipped with a system close button.

10.3.2 Creation of a Miniframe Window

The miniframe window is based on the **CMiniFrameWnd** class. To create a miniframe window, first declare a variable or a pointer to **CMiniFrameWnd**. You can define its class using the **AfxRegisterWndClass()** function. You can then pass the return value of that function to its Create() method. The syntax of this method is:

```
virtual BOOL Create(
   LPCTSTR lpClassName,
   LPCTSTR lpWindowName,
   DWORD dwStyle,
   const RECT& rect,
   CWnd* pParentWnd = NULL,
   UINT nID = 0
);
```

The lpClassName parameter should be a value returned from the **AfxRegisterWndClass()** function.

The lpWindowName is the caption displayed in the title bar of the window.

The dwStyle parameter is the style to apply to the window. Normally, it would use the regular styles of a normal window but many styles are invalid. For example, whether you add the Minimize and the Maximize buttons or not, they cannot be displayed. You should use only the following combination: **WS_POPUP | WS_CAPTION | WS_SYSMENU**. Besides the regular window styles, you should combine them with one or more of the following special styles:

✍✍ **MFS_MOVEFRAME**: With this style, if the user clicks and drags one (any) edge of the frame, the frame would start moving with the same reaction as if the user were dragging the title bar. If you apply this style, the user cannot resize the frame even if another style would allow it

✍✍ **MFS_4THICKFRAME**: This style prevents the user from resizing the mini frame

✍✍ **MFS_SYNCACTIVE**: This style makes sure that the mini frame is activated when its parent window is activated

✍✍ **MFS_THICKFRAME**: This creates a thick frame and allows the user to resize it if necessary, provided the **MFS_MOVEFRAME** is not used

✍✍ **MFS_BLOCKSYSMENU**: This disables access to the system menu

The rect parameter specifies the location and dimensions of the frame window.

The pParentWnd argument is the CWnd parent of the window. This argument is not required.

The nID argument is the identifier of the mini frame window. It is not required.

Besides the **Create()** method, the **CMiniFrameWnd** class also provides the **CreateEx()** member function to create a miniframe window.

Here is an example:

```
void CMiniFrame2Dlg::OnBnClickedMiniframeBtn()
{
	// TODO: Add your control notification handler code here
	CMiniFrameWnd *MFW = new CMiniFrameWnd;
	CString StrClassName = AfxRegisterWndClass(
				CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS,
				LoadCursor(NULL, IDC_ARROW),
				(HBRUSH)GetStockObject(COLOR_BTNFACE+1),
				LoadIcon(NULL, IDI_APPLICATION));

	MFW->CreateEx(0,
			StrClassName,
			"Small Application",
			WS_POPUP | WS_CAPTION | WS_SYSMENU |
			MFS_BLOCKSYSMENU,
			CRect(100, 100, 350, 420));

	MFW->ShowWindow(SW_SHOW);
}
```

Chapter 11: Introduction to Windows Controls

✍ Controls Fundamentals

✍ Parent Controls

✍ Windows Controls

☞ Controls Styles and Common Properties

☞ Extended Styles

11.1 Controls Fundamentals

11.1.1 Introduction

A Windows control, in this book called a control, is an object that displays or is part of an application and allows the user to interact with the computer. There are various types of controls as we will see in this book:

?? A text-based control is an object whose main function is to display to, or request text from, the user

?? A list-based control displays a list of items

?? A progress-based control is used to show the progress of an action

?? a static control can be used to show colors, a picture or something that does not regularly fit in the above categories

To make your application as efficient as possible, you will add controls as you judge them necessary. Some and most controls are already available so you can simply customize their behavior and appearance. Sometimes you will need a control for a specific task or for a better look. If such a control is not available, you may have to create your own.

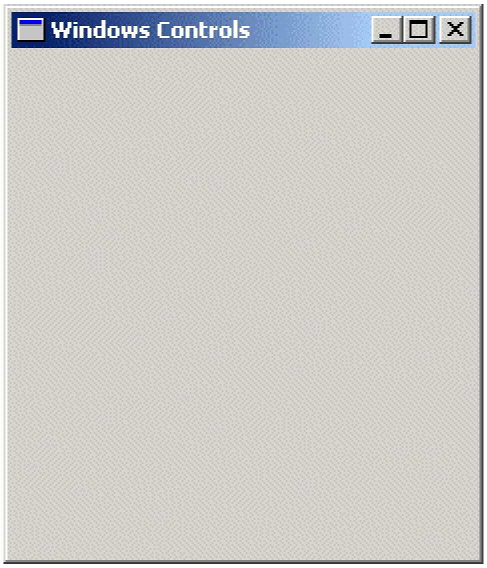
There are two main ways a control is made part of your application. At design time, which is referred to the time you are visually creating an application, you will select controls and place them on a host. Another technique, referred to as run time, consists of creating control programmatically. In this case you must write all the code that specifies the control's appearance and behavior.

Practical Learning: Introducing Controls

1. Start Visual Studio or Visual C++ and create a new Empty Win32 Project named **Win32C**
2. Create a C++ Source file named **Exercise** and type the following in it:

```
#include <windows.h>
//---------------------------------------------------------------------------
char StrClassName[] = "WndCtrls";
char StrWndName[]   = "Windows Controls";
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg,
                         WPARAM wParam, LPARAM lParam);
//---------------------------------------------------------------------------
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow )
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;

    WndClsEx.cbSize        = sizeof(WNDCLASSEX);
    WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc   = WndProc;
    WndClsEx.cbClsExtra          = 0;
    WndClsEx.cbWndExtra    = 0;
    WndClsEx.hInstance     = hInstance;
    WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
    WndClsEx.lpszMenuName  = NULL;
    WndClsEx.lpszClassName = StrClassName;
    WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx(0, StrClassName, StrWndName,
                          WS_OVERLAPPEDWINDOW,
                           340, 200,
                           240, 320,
                           NULL, NULL, hInstance, NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    while( GetMessage(&Msg, NULL, 0, 0) )
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }

    return 0;
}
//---------------------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
   switch(Msg)
   {
   case WM_DESTROY:
      PostQuitMessage(WM_QUIT);
      break;
   default:
      return DefWindowProc(hWnd, Msg, wParam, lParam);
   }
   return 0;
}
//---------------------------------------------------------------------------
```

3. Test the application



4. Close it and return to MSVC

11.1.2 The Parent-Child Window Relationship

There are two types of windows or controls you will deal with in your applications. The type referred to here is defined by the relationship a window has with regards to other windows that are part of an application:

?? **Parent**: a window is referred to as a parent when there are, or there can be, other windows that depend on it. For example, if you look at the toolbar of Visual C++, it is equipped with the buttons. The toolbar is a parent to the buttons. When a parent is created, it "gives life" to other windows that can depend on it. The most regular parents you will use are forms and dialog boxes.

?? **Child**: A window is referred to as child when its existence and especially its visibility depend on another window called its parent. When a parent is created, made active, or made visible, it gives existence and visibility to its children. When a parent gets hidden, it also hides its children. If a parent moves, it moves with its children. The children keep their positions and dimensions inside the parent. When a parent is destroyed, it also destroys its children (sometimes it does not happen so smoothly; a parent may make a child unavailable but the memory space the child was occupying after the parent has been destroyed may still be use, sometimes filled with garbage, but such memory may not be available to other applications until you explicitly recover it).

Child controls depend on a parent because the parent "carries", "holds", or hosts them. All of the Windows controls you will use in your applications are child controls. A child window can also be a parent of another control. For example, the Standard toolbar of Visual Studio is the parent of the buttons on it. If you close or hide the toolbar, its children disappear. At the same time, the toolbar is a child of the application's frame. If you close the application, the toolbar disappears, along with its own children. In this example, the toolbar is a child of the frame but is a parent to its buttons.

It is important to understand that this discussion refers to parents and children as windows, not as classes:

?? The **CButton** class is based on **CWnd**. Therefore, **CWnd** is the *parent class* of **CButton**

?? When a button is placed on a form, the form, which is based on **CView** (indirectly) is the window parent of the button and not its class parent

If you are creating an independent window, as we have done with the frames so far, you can simply call its **Create()** method and define its characteristics. Such a class or window is a prime candidate for parenthood.

11.2 Parent Controls

11.2.1 Definition

An object is referred to as a parent or container when its main job is to host other controls. This is done by adding controls to it. Besides all of the characteristics of **WNDCLASS**, the **WNDCLASSEX** structure requires that you specify the size of the structure and it allows you to set a small icon for your application.

Besides the **CreateWindow()** function, the Win32 library provides the **CreateWindowEx()** function you can use to create a window. Once again, the **CreateWindowEx()** function has the followsing syntax:

```
HWND CreateWindowEx(
  DWORD dwExStyle,
  LPCTSTR lpClassName,
  LPCTSTR lpWindowName,
  DWORD dwStyle,
  int x,
  int y,
  int nWidth,
  int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
  LPVOID lpParam
);
```

11.2.2 Parent Windows Styles

The role of a parent object is very important to the other controls it is hosting. Based on this, the parent window must possess some valuable characteristics. For example, you must provide a way to close the parent window. If necessary, you can also allow moving the parent. This is usually done by the user dragging the title bar of a frame or a dialog box. If the parent is a toolbar, you can make it dockable, in which case it can be moved and positioned to various parts of the frame window. You can also allow the user to move a parent such as a dialog box by dragging its body. We will review the characteristics of parent window when studying dialog boxes

11.3 Windows Controls

11.3.1 Introduction

To create a control, you can use either the **CreateWindow()** or the **CreateWindowEx()** Win32 functions. When doing this, you must specify the control's name. To help with this, the Win32 library provides already defined names of classes you can use to create a control.

We already saw that, when creating a window, you need to declare its handle and initialize it with the **CreateWindow()** or the **CreateWindowEx()** functions. If the window is created successfully, the function returns a window handle. If the control created was a parent, its handle can be used as the *hWndParent* argument. This would be done as follows:

```
HWND hWndParent;

// Create the parent window
hWndParent = CreateWindowEx(0, ClassName, StrWndName,
                            WS_OVERLAPPEDWINDOW,
                            0, 100, 140, 320,
                            NULL, NULL, hInstance, NULL);

// Create a window
CreateWindowEx(0, WndClassName, CaptionOrText,
                            ChildStyle, Left, Top, Width, Height,
                            hWndParent, NULL, hInstance, NULL);
```

The reason you declare a handle to the parent window is to allow you to refer to it later on. In the same way, if you need to refer to any control that is part of your application, you should define its own handle:

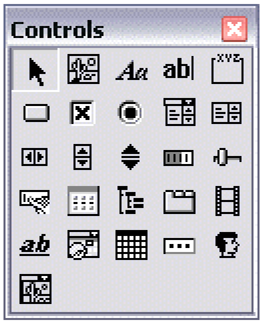
```
HWND hWndParent, hWndChild;

// Create the parent window
hWndParent = CreateWindowEx(0, ClassName, StrWndName,
                            WS_OVERLAPPEDWINDOW,
                            0, 100, 140, 320,
                            NULL, NULL, hInstance, NULL);

// Create a window
hWndChild = CreateWindowEx(0, WndClassName, CaptionOrText,
                            ChildStyle, Left, Top, Width, Height,
                            hWndParent, NULL, hInstance, NULL);
```

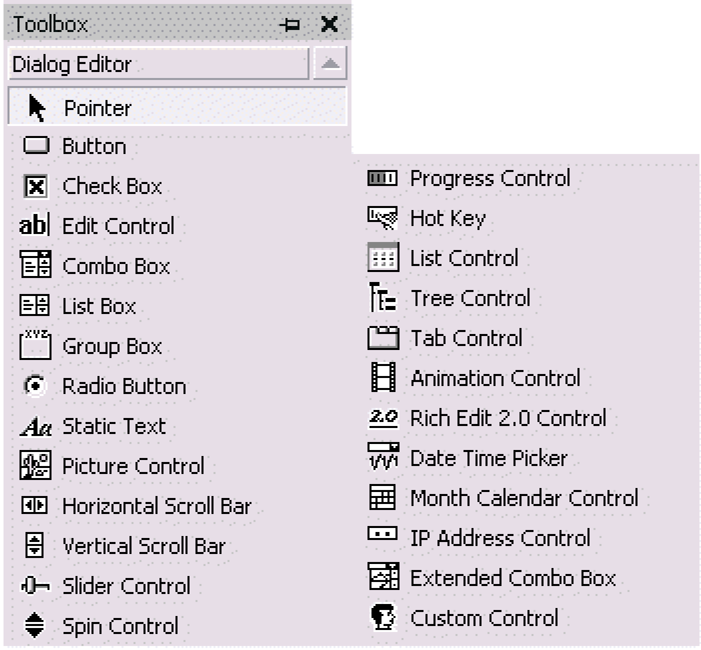
11.3.2 Control Creation Options

To allow the user to interact with the computer, you equip your application with the necessary controls. The controls are positioned on the client area of the parent window. The easiest technique used to add a control to an application consists of picking it up from the Controls toolbox and positioning it on a parent window. To help with this, Visual C++ provides a series or ready-made objects on the Controls toolbox:

Visual C++ 6 Controls

Visual C++ 7 Controls





The second option used to add a control to an application consists of programmatically creating it. To do this, you can call either the **CreateWindow()** or the **CreateWindowEx()** functions.

All visible controls of the MFC library are based on the **CWnd** class. This class is equipped with all the primary functionality that a regular window object needs. As the parent of all visible window classes, it provides the other controls with properties and methods to implement their appearance and behavior. To use the **CWnd** class you have various options:

?? You can directly derive a class from **CWnd**

?? You can use one of the **CWnd**-derived classes (**CView**, **CFrameWnd**, etc)

?? You can derive a class from one of the **CWnd**-derived classes; for example you can create a class based on **CStatic**

?? You can directly declare a **CWnd** variable in your application and initialize it with the class of your choice

To create a control using a known class, you must declare a variable of the class on which the control is based. Every control is based on a specific class, and that class is directly or indirectly derived from **CWnd**. An example of such a declaration would be:

```
void CParentLoader::TodaysMainEvent()
{
        CStatic Label;
}
```

You can also declare the variable as a pointer. In this case, make sure that you initialize the class using the **new**operator. An example would:

```
void CParentLoader::TodaysMainEvent()
{
        CStatic *Label = new CStatic;
}
```

Based on the rules of inheritance and polymorphism, you can also declare the pointer as **CWnd** but initialize it with the desired class. Here is an example:

```
void CParentLoader::TodaysMainEvent()
{
	CWnd *Panel = new CStatic;
}
```

After declaring the variable, you can call the **CWnd::Create()** method to initialize the control. The syntax of this method is:

```
virtual BOOL Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
                    DWORD dwStyle, const RECT& rect, CWnd* pParentWnd,
                    UINT nID, CCreateContext* pContext = NULL);
```

To create a fancier control, you can use the **CWnd::CreateEx()** method.

The availability of a child window to other parts of a program is defined by the scope in which the control is created. If you create a control in an event or a method of a parent class, it can be accessed only inside of that event. Trying to access it outside would produce an error. If you want to access a child or dependent window from more than one event or method, you must create it globally. This is usually done by declaring the variable or pointer in the class of the parent.

After declaring the variable, to initialize it, call its **Create()** method. The syntax of this method depends on the class you are using to create the window. After initializing the window, it may become alive, depending on the style or properties you assigned to it.

The other technique you can use is to derive a class from the control whose behavior and appearance can serve as a foundation for your control. For example, you can derive a control as follows:

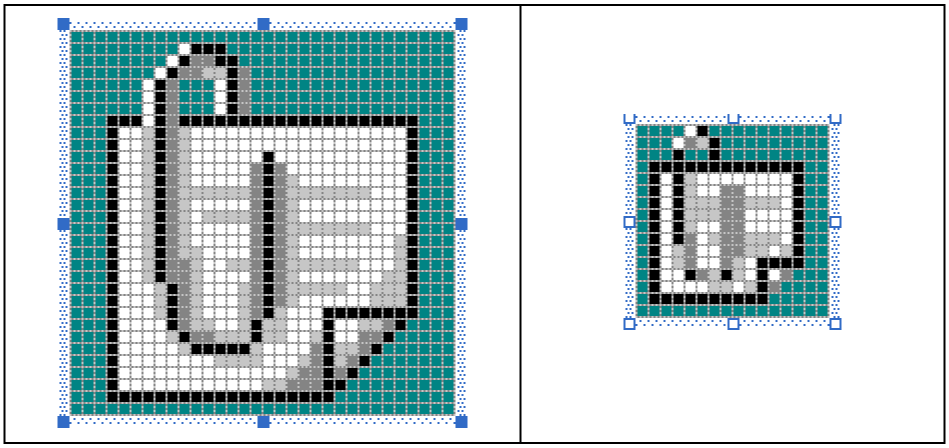
```
class CColoredLabel : public CStatic
{
public:
	CColoredLabel();
	virtual ~CColoredLabel();

	DECLARE_MESSAGE_MAP()
};
```

Such a class as the above CColoredLabel gives you access to the **public** and **protected** properties and methods of the base class. To use the new class in your application, declare a variable from it and call the **Create()** method.

Practical Learning: Introducing Controls

1. To create a new application, on the main menu, click File -> New ->Project…
2. Click MFC Application. Set the application Name as **Controls** and click OK
3. Set the **Application Type** to **Dialog Based** and click Finish
4. On the dialog, click the TODO line and press Delete on the keyboard

5. Change the design of the IDR_MAINFRAME icon as follows:



6. Display the dialog box

11.3.3 The Control's Class Name

To create a control, you must specify its name. To do this, you have various options. The Win32 library provides already defined names of classes you can use to create a control. These classes are:

Class Name	Description
STATIC	A static control can be used to display text, a drawing, or a picture
EDIT	As it name suggests, an edit control is used to display text to the user, to request text from the user, or both
RichEdit	Like an edit box, a rich edit control displays text, requests it, or does both. Besides all the capabilities of the edit control, this control can display formatted text with characters or words that use different colors or weight. The paragraphs can also be individually aligned. The RichEdit class name is used to create a rich edit control using the features of Release 1.0 of its class
RICHEDIT_CLASS	This control is used for the same reasons as the RichEdit class name except that it provides a few more text and paragraph formatting features based on Release 2.0
LISTBOX	A list box is a control that displays items, such as text, arranged so each item, displays on its own line
COMBOBOX	A combo box is a combination of an edit control and a list box. It holds a list of items so the current selection displays in the edit box part of the control
SCROLLBAR	A scroll bar is a rectangular object equipped with a bar terminated by an arrow at each end. It is used to navigate left and right or up and down on a document
BUTTON	A button is an object that the user clicks to initiate an action
MDICLIENT	This class name is used to create a child window frame for an MDI application

To use one of these classes, you have various options. At design time, on the Controls toolbox, you can click the Custom Control button and click the dialog box or form.

After placing the control on the host. This would display the Properties window that allows you to visually set the characteristics of the control.

In Visual C++ .Net, the Properties window displays, by default, to the lower right side of the IDE.

If you are creating or designing a control using the Custom button, on the Properties window and in the Class Name box, you can type the desired name as above for the control.

If you are programmatically creating the control, pass its name as string to the *lpszClassName* argument of the **Create()** or the **CreateEx()** methods. Here is an example that would start an edit box:

```
void CSecondDlg::OnThirdControl()
{
	// TODO: Add your control notification handler code here
	CWnd *Memo = new CWnd;

	Memo->Create("BUTTON', );
}
```

Another option you have is to either declare a **WNDCLASS** variable and initialize it, or call the **AfxRegisterWndClass()** function and pass it the desired values. As we saw in the previous lesson, this function returns a string. You can pass that string to the **CWnd::Create()** method as the class name. Here is an example:

```
void Whatever()
{
	// TODO: Add your control notification handler code here
	CWnd *First = new CWnd;
	CString StrClsName = AfxRegisterWndClass(
				CS_VREDRAW | CS_HREDRAW,
				LoadCursor(NULL, IDC_CROSS),
				(HBRUSH)GetStockObject(BLACK_BRUSH),
				LoadIcon(NULL, IDI_WARNING));
				First->Create(StrClsName, );
}
```

The other option you have is to specify the class name as **NULL**. In this case, the compiler would use a default name:

```
void Whatever ()
{
	// TODO: Add your control notification handler code here
	CWnd *Second = new CWnd;

	Second->Create(NULL, );
}
```

Practical Learning: Creating a Control

1. On the Controls toolbox, click the Custom Control button and click on the top left section of the dialog box (you do not need to be precise, anywhere you place it will be fine)

2. On the dialog box, click the custom control to select it
3. On the Properties window, click the Class box, type **Static** and press Enter
4. Test the application and return to MSVC

11.3.4 The Control's Window Name

The window name is a default string that displays on the control when the control comes up. In the previous lessons, we saw that this string displays as caption on the title bar of a window frame or a dialog box. This property is different for various controls.

To set the default text of a control, you can either do this when creating the control or change its text at any time. Again, this can depend on the control. To set the default text when programmatically creating the control, pass a string value as the *lpszWindowName* argument of the **Create()** or **CreateEx()** method. Here is an example:

```
void CSecondDlg::OnThirdControl()
{
	// TODO: Add your control notification handler code here
	CWnd *Memo = new CWnd;

	Memo->Create("EDIT", "Voice Recorder", );
}
```

Many (even most) controls do not use a window name. Therefore, you can pass it as NULL.

Practical Learning: Setting a Caption

1. On the dialog box, click the custom button. Then, on the Properties window, delete the content of the Caption field. Then, in the Class field, type **ListBox**
2. Test the application and return to MSVC

11.4 Controls Styles and Common Properties

11.4.1 Childhood

A style is a characteristic that defines the appearance, and can set the behavior, of a control. The styles are varied from one control to another although they share some of the characteristics common to most Windows controls.

All of the controls you will create need to be hosted by another control. During design, once you position a control on a dialog box or a form, it automatically gets the status of child. If you are programmatically creating a control, to specify that it is a child, add the **WS_CHILD** to the *dwStyle* argument. Here are examples:

```
void CSecondDlg::OnFirstControl()
{
	// TODO: Add your control notification handler code here
	CWnd *First = new CWnd;
```

```
        CString StrClsName = AfxRegisterWndClass(
                                CS_VREDRAW | CS_HREDRAW,
                                LoadCursor(NULL, IDC_CROSS),
                                (HBRUSH)GetStockObject(BLACK_BRUSH),
                                LoadIcon(NULL, IDI_WARNING));

        First->Create(StrClsName, NULL, WS_CHILD, );
}

void CSecondDlg::OnSecondControl()
{
        // TODO: Add your control notification handler code here
        CWnd *Second = new CWnd;

        Second->Create(NULL, NULL, WS_CHILD, );
}

void CSecondDlg::OnThirdCont rol()
{
        // TODO: Add your control notification handler code here
        CWnd *Memo = new CWnd;

        Memo->Create("EDIT", "Voice Recorder", WS_CHILD, );
}

void CSecondDlg::OnFourthControl()
{
        // TODO: Add your control notification handler code here
        CStatic *Label = new CStatic;

        Label->Create("United Nations", WS_CHILD, );
}
```

The **WS_CHILD** style is defined in the Win32 library as:

```
#define WS_CHILD          0x40000000L
```

11.4.2 Visibility

If you want to display a control to the user when the parent window comes up, at design time, set the **Visible** property to True. If you are programmatically creating the control, add the **WS_VISIBLE** style. Here is an example:

```
void CSecondDlg::OnSecondControl()
{
        // TODO: Add your control notification handler code here
        CWnd *Second = new CWnd;

        Second->Create(NULL, NULL, WS_CHILD | WS_VISIBLE, );
}
```

The visibility style is defined as:

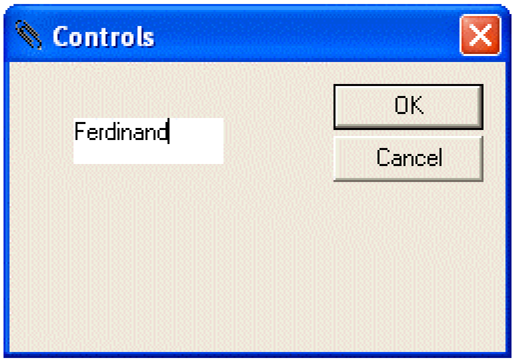
```
#define WS_VISIBLE          0x10000000L
```

The child style combined with the visibility style produces:

```
  4000 0000
| 1000 0000
-----------------
= 5000 0000
```

Practical Learning: Displaying a Control

1. After making sure the custom control is selected on the dialog box, on the Properties window, set the **Class** name to **Edit**
2. To apply childhood and visibility to a control, , in the **Style** field, type **0x50000000** and press Enter
3. Test the application. In the edit box, type a name:



4. Close it and return to MSVC

11.4.3 Availability

By default, after a control has been created, it is available to the user. If it is a non-static control, the user can possibly select or change its value. If you do not want the control to be immediately usable but do not want to hide it, you can disable it. To do this, if you are designing the control, you can set its **Disable** property to True or checked. If you are programmatically creating the control, add the **WS_DISABLED** style as follows:

```
void CSecondDlg::OnSecondControl()
{
	// TODO: Add your control notification handler code here
	CWnd *Second = new CWnd;

	Second->Create(NULL, NULL, WS_CHILD | WS_VISIBLE | WS_DISABLED, );
}
```

The dis abled style is defined as follows:

```
#define WS_DISABLED         0x08000000L
```

The above combination would produce:

```
  4000 0000
| 1000 0000
| 0800 0000
---------------
```

= 5800 0000

```
BOOL EnableWindow(BOOL bEnable = TRUE);
```

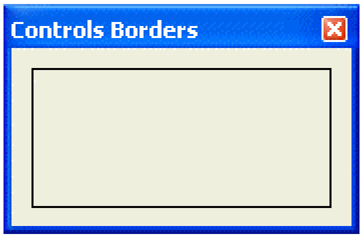
Practical Learning: Disabling a Control

1. To disable the custom control, change its style to a value of **0x58000000**
2. Test the application and trying typing in the edit box again
3. Close it and return to MSVC

11.4.4 Borders

One of the visual features you can give a control is to draw its borders. As it happens, most Microsoft Windows controls have a 3-D look by default. Some other controls must explicitly be given a border. To add a border to a control, at design time, set its **Border** property to True. If you are programmatically creating the control, add the **WS_BORDER** style to it:

```
void CSecondDlg::OnSecondControl()
{
	// TODO: Add your control notification handler code here
	CWnd *Second = new CWnd;

	Second->Create(NULL, NULL, WS_CHILD | WS_VISIBLE | WS_BORDER, );
}
```

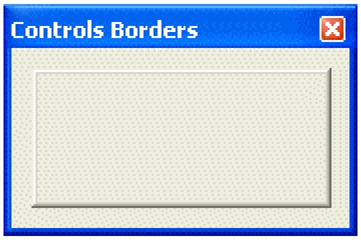


The border style is defined as:

```
#define WS_BORDER          0x00800000L
```

To raised the borders of such a control, add the **WS_THICKFRAME** to its styles:

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CtrlBorders->Create(NULL, NULL,
	WS_CHILD | WS_VISIBLE | WS_THICKFRAME, );

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

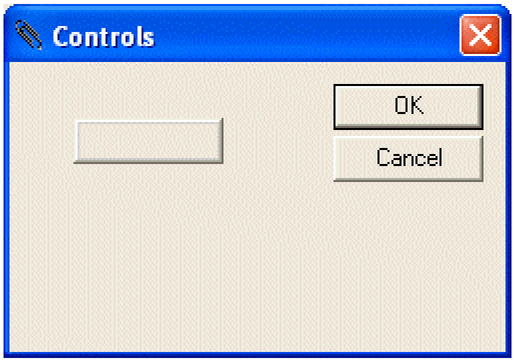


The thick frame style is defined as

```
#define WS_THICKFRAME       0x00040000L
```

Practical Learning: Adding Borders to a Control

1. Change the **Class** name to **static**
2. To add a simple border to the custom control, combine the child, the visible and the thick frame styles to give a value of **0x50040000** to its **Style** field
3. Test the application



4. Close it and return to MSVC

11.4.5 Tab Sequence

When using the controls of a dialog, the user may choose to press Tab to navigate from one control to another. This process follows a sequence of controls that can receive input in an incremental order. To include a control in this sequence, at design time, set its **Tab Stop** property to True.

If you are creating your control with code, to make sure that it can fit in the tab sequence, add the **WS_TABTOP** style. Here is an example:

```
void CSecondDlg::OnThirdControl()
{
        // TODO: Add your control notification handler code here

        Memo->Create("EDIT", "Voice Recorder",
           WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, );

        CWnd *CanTab = new CWnd;

        CanTab->Create("BUTTON", "&Apply",
                              WS_CHILD |  WS_VISIBLE | WS_BORDER | WS_TABSTOP, );
}
```

The tab stop style is defined as:

```
#define WS_TABSTOP        0x00010000L
```

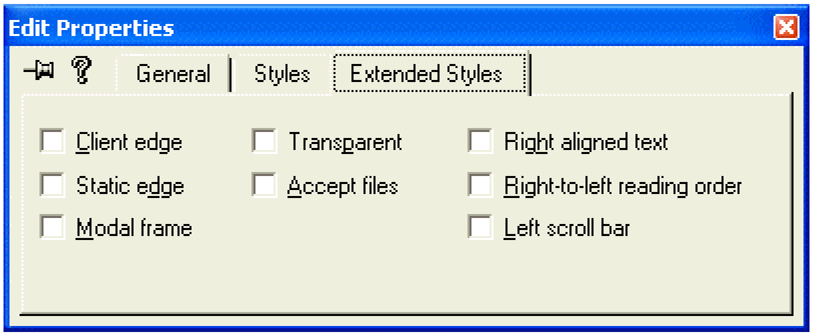
11.5 Extended Styles

11.5.1 Introduction

Besides the above regular styles and properties used on controls, if you want to add a more features to a window, you can create it using the **CWnd::CreateEx()** method. It comes in two versions as follows:

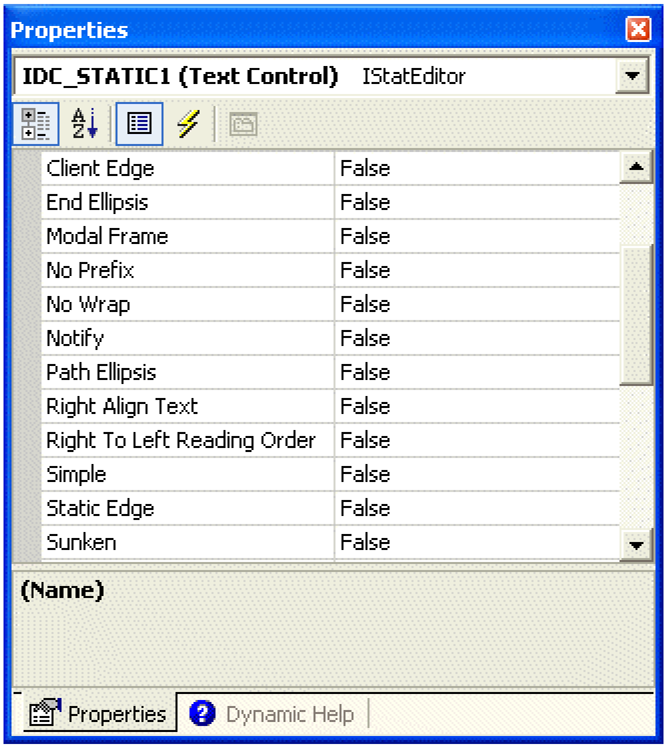
```
BOOL CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
          DWORD dwStyle, int  x, int y, int nWidth, int nHeight,
          HWND hwndParent, HMENU nIDorHMenu, LPVOID lpParam = NULL );

BOOL CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
          DWORD dwStyle, const RECT& rect, CWnd* pParentWnd,
          UINT nID, LPVOID lpParam = NULL);
```

In Visual C++ 6, some of the extended styles are on the Styles tab of the Properties window and some others are in the **Extended Styles** tab:



□ Client edge
□ Static edge
□ Modal frame
□ Transparent
□ Accept files
□ Right aligned text
□ Right-to-left reading order
□ Left scroll bar

In Visual C++ 7, all styles are listed in the Properties window's vertical list:



IDC_STATIC1 (Text Control) IStatEditor

Client Edge	False
End Ellipsis	False
Modal Frame	False
No Prefix	False
No Wrap	False
Notify	False
Path Ellipsis	False
Right Align Text	False
Right To Left Reading Order	False
Simple	False
Static Edge	False
Sunken	False

(Name)

11.5.2 Left Text Alignment

Text-based controls (such as the static label, the edit box, or the rich edit control) align their text to the left by default. This means that when the control displays, its text starts on the left side of its area (for US English and other Latin-based versions of Microsoft Windows). To align text to the left on a control that allows it, at design time, select the **Left** value in the **Align Text** combo box.

The default text alignment of a text-based control is to the left. This is because the **WS_EX_LEFT** extended style is applied to it. If you want to reinforce this, you can add that style as the *dwExStyle* argument of the **CreateEx()** method. This can be done as follows:

```
void CSecondDlg::OnThirdControl()
{
	// TODO: Add your control notification handler code here

	Memo->CreateEx(WS_EX_LEFT, "EDIT", "Voice Recorder",
	   WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, );
}
```

The extended left text alignment is defined as:

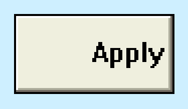
```
#define WS_EX_LEFT              0x00000000L
```

11.5.3 Right Text Alignment

Many text-based controls, including the button, allow you to set their text close to the right border of their confined rectangle. To do this at design time, select the **Right** value in the **Align Text** combo box.

If you are programmatically creating the control, to align its text to the right, set or add the **WS_EX_RIGHT** extended style. Here is an example:

```
void CSecondDlg::OnThirdControl()
{
	// TODO: Add your control notification handler code here
	CWnd *CanTab = new CWnd;

	CanTab->CreateEx(WS_EX_RIGHT, "BUTTON", "&Apply",
	     WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, );
}
```

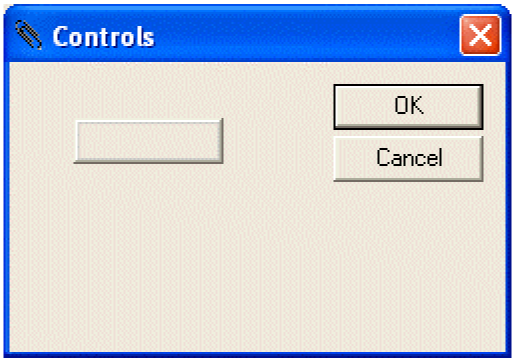


The extended left text alignment is defined as:

```
#define WS_EX_RIGHT            0x00001000L
```

Practical Learning: Right-Aligning Text

1. Set the Caption of the Custom control to Leo and change its ExStyle value to 0x00001000
2. Test the application



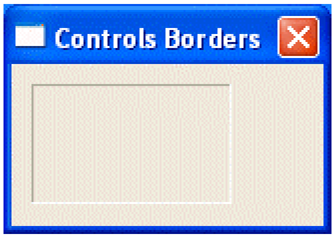
3. Return to MSVC

11.5.4 Extended Borders

Besides the border features of the dwStyle argument of the Create() member function, the CreateEx() method provides other border drawings to apply to a control.

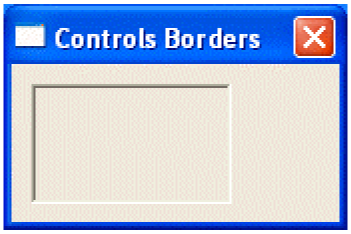
Static Borders: To give a 3-D appearance to a control, especially one that does not receive input from the user (such as a static control), you can apply a static edge to it. To do this, at design time, set the Static Edge property to True. At run time, you can set or add the WS_EX_STATICEDGE extended style. Here is an example:

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CWnd* Panel = new CStatic;

	Panel->CreateEx(WS_EX_STATICEDGE, "STATIC", NULL,
		WS_CHILD | WS_VISIBLE, );

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



Sunken Borders: You can further sink the borders of a control, give them an advanced 3-D appearance. To apply this effect at design time, set the Client Edge property to True. If you are programmatically creating a control, add the **WS_EX_CLIENTEDGE** extended style. Here is an example:

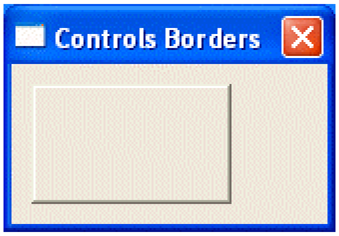
```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CWnd* Panel = new CStatic;

	Panel->CreateEx(WS_EX_CLIENTEDGE, "STATIC", NULL,
		WS_CHILD | WS_VISIBLE, );

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



Raised Borders: A control appears raised when it borders come out of the client area. This creates the effect of light left and top borders while the right and bottom borders appear darker. To apply this property, at design time, set the Modal Frame property to True or, at run time, add the **WS_EX_DLGMODALFRAME** extended style. Here is an example:

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
```

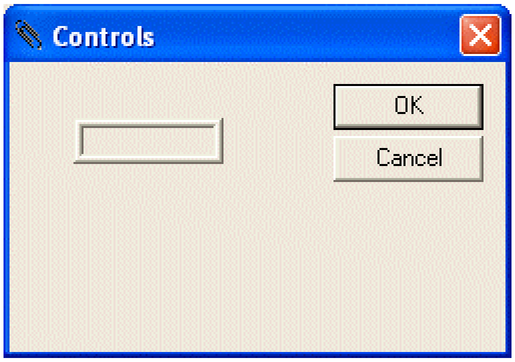
```
            // TODO: Add extra initialization here
            CWnd* Panel = new CStatic;

            Panel->CreateEx(WS_EX_DLGMODALFRAME, "STATIC", NULL,
                        WS_CHILD | WS_VISIBLE, );

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



To raise only the borders of a control without including the body of the control, you can combine **WS_EX_CLIENTEDGE** and the **WS_EX_DLGMODALFRAME** extended styles. Here is an example:

Practical Learning: Extending a Control's Borders

1. To create a fancy static control, delete its Caption field and change its ExStyle value to 0x201
2. Test the application

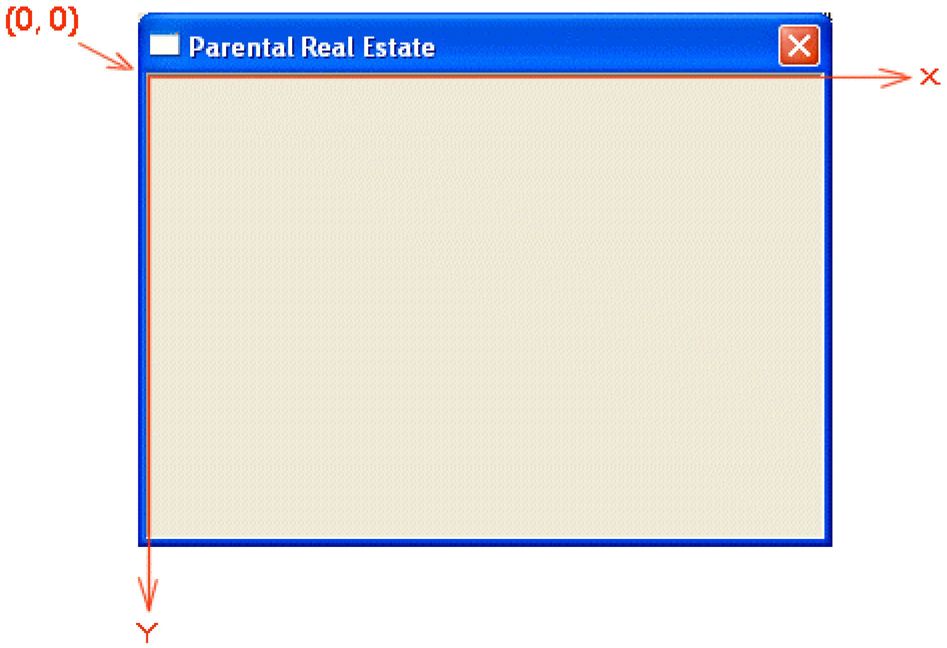


3. Return to MSVC

11.5.5 Controls Location and Dimensions

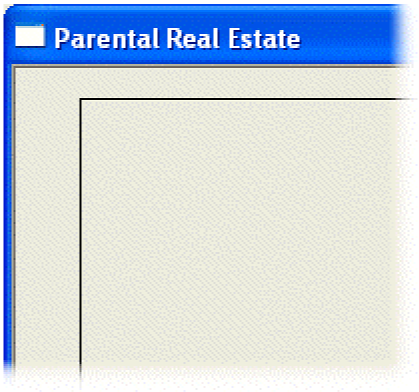
One of the main roles of a parent window is its ability to host other controls. Such controls are called its children because the parent carries them when it is moved. The parent also controls the children's visibility and their availability. These controls are also called its clients because they request the parental service from it.

The controls are confined to the area of the body offered by the parent window. After visually adding it to a host, the control assumes a location and takes some dimensions in a rectangular body of its parent. The origin of this rectangular area is on the upper-left corner of the parent window. The horizontal measurements move from the origin to the right. The vertical measurements move from the origin to the bottom:



If you are programmatically creating the control, to set its location, if you are using the **CreateWindow()**, the **CreateWindowEx()** functions, or the first version of the **CreateEx()** method, specify the value for the distance from the left border of the body of the parent to the left border of the control, and pass the desired value for the upper distance from the top border of the body of the parent to the top border of the control. Here is an example:

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CWnd* Panel = new CStatic;

	Panel->CreateEx(NULL, "STATIC", NULL,
	     WS_CHILD | WS_VISIBLE | WS_BORDER,
	     32, 15, );

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

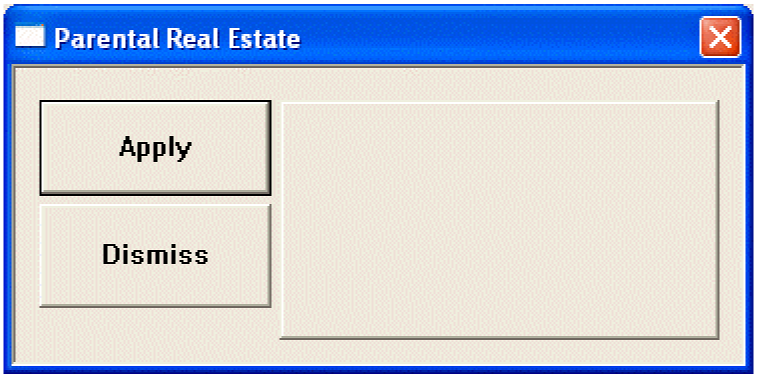


If you specify negative values for the left and top distances, either the left or the top borders, respectively, will be hidden.

To set the dimensions of the control, if you are using the **CreateWindow()**, the **CreateWindowEx()** functions, or the first version of the **CreateEx()** member function, specify the *nWidth* for the width and the *nHeight* for the height of the control. If you specify measures that are higher than the width of the body of the parent - x or the height of the of the parent - y, the right border or the bottom border, respectively, of the control will be hidden.

To specify the location and the dimensions of the control at the same time, pass a **RECT** or a **CRect** variable as the *rect* argument of the **Create()** or the second version of the **CreateEx()** methods. Here are examples:

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CWnd* btnApply = new CWnd;
	RECT Recto = {12, 16, 128, 64};
	btnApply->Create("BUTTON", "&Apply", WS_CHILD | WS_VISIBLE,
		Recto, );

	CWnd *btnDismiss = new CWnd;
	btnDismiss->Create("BUTTON", "&Dismiss", WS_CHILD | WS_VISIBLE,
		 CRect(12, 68, 128, 120), );

	CWnd* Panel = new CStatic;
	Panel->CreateEx(WS_EX_DLGMODALFRAME, "STATIC", NULL,
		WS_CHILD | WS_VISIBLE | WS_BORDER,
		132, 16, 220, 120, );

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



11.5.6 Parenthood

After specifying that the control you are creating is a child, which is done by placing the control on a form or a dialog box or by using or adding the **WS_CHILD** style, you must specify what window is the parent of your control. The parenthood of a control is automatically set when the control is placed on a host: the host becomes the parent and the parent is responsible for destroying the children when it is closed.

If you are programmatically creating the control, you can specify its parent by passing a window handle as the *pParentWnd* argument of the the **CreateWindow()**, the **CreateWindowEx()** functions, the **Create()**, or the second version of the **CreateEx()** member functions. If you are creating the control in a member function or an event of the parent window that is a descendent of **CWnd**, you can pass this argument as the **this** pointer. Here is an example:

```
void CSecondDlg::OnFirstControl()
{
	// TODO: Add your control notification handler code here
	CWnd *First = new CWnd;
	CString StrClsName = AfxRegisterWndClass(CS_VREDRAW | CS_HREDRAW,
				LoadCursor(NULL, IDC_CROSS),
				(HBRUSH)GetStockObject(BLACK_BRUSH)
				LoadIcon(NULL, IDI_WARNING));

	First->Create(StrClsName, NULL,WS_CHILD | WS_VISIBLE | WS_BORDER,
		CRect(20, 20, 120, 60), this, );
}
```

Specifying the parent as the **this** pointer indicates that when the parent is destroyed, it will also make sure the child control is destroyed. If you want the application to be the parent and owner of the control, use the first version of the **CreateEx()** method and pass the handle of your application as the *hwndParent* argument.

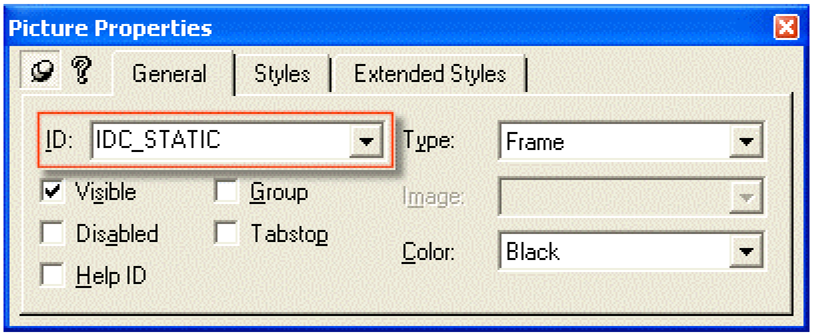
11.5.7 Control Identification

After adding a control to your application, you must identify it. An identifier is not a string, it is a constant integer that is used to identify the control. Therefore, it is not the name of the control.

If you have just added the control, Visual C++ would assign a default identifier. If the control is static, it would have the **IDC_STATIC** identifier. Every other control gets an identifier that mimics its class name followed by a number. An example would be

IDC_EDIT1. If you add another control of the same kind, it would receive an identifier with an incremental number.

To have a better idea of each control, you should change its ID to make it more intuitive. To do this, if you are visually adding the control, first display the Properties window:



ID: IDC_STATIC
Type: Frame
☑ Visible
□ Group
Image:
□ Disabled
□ Tabstop
Color: Black
□ Help ID

The identifier can be a word or a hexadecimal number. Here are the rules you must follow when assigning identifiers:

If you decide to use a word	If you decide to use a hexadecimal number
?? The identifier must start with an underscore or a letter ?? The identifier must be in one word ?? After the first character, the identifier can have letters, underscores, and digits in any combination ?? The identifier must not have non-alphanumeric characters ?? The identifier cannot have space(s)	?? The identifier must start with a digit or one of the following letters: a, b, c, d, e, f, A, B, C, D, E, or F ?? The identifier can have only digits and the above letters except this: if the identifier starts with 0, the second character can be x or X followed by the above letters and digits so that, when converted to decimal, it must have a value between -32768 and 65535

Here are suggestions you should follow:

?? The identifier should be in all uppercase

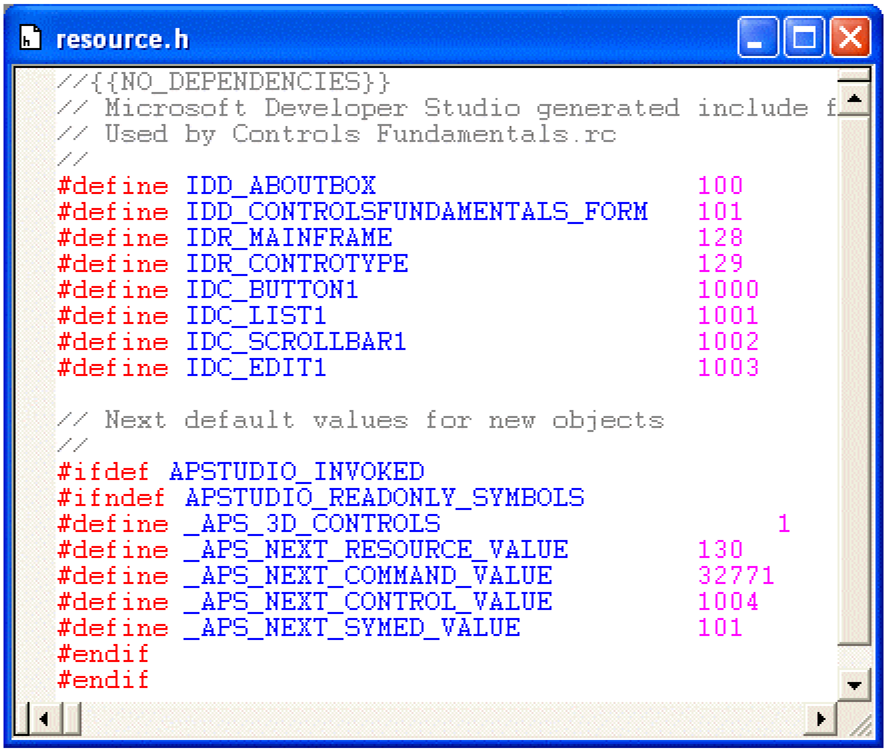
?? The identifier should have a maximum of 30 characters

?? If the object is a form or dialog box, its identifier should start with IDD_ followed by a valid name. Examples: IDD_EMPLOYEES or IDD_EMPL_RECORS

?? If you are creating a control, the identifier should start with IDC_ followed by a name. Examples are: IDC_ADDRESS or IDC_FIRST_NAME

?? If you are creating a static control but plan to use it in your code, (you must) change its identifier to something more meaningful. If the static control will hold text and you plan to change that text, you can identify it with IDC_LABEL. For a static control used to display a picture of employees, you can identify it as IDC_EMPL_PICTURE

The identifiers of the controls used in your application must be listed in a header file. This file is usually called Resource.h



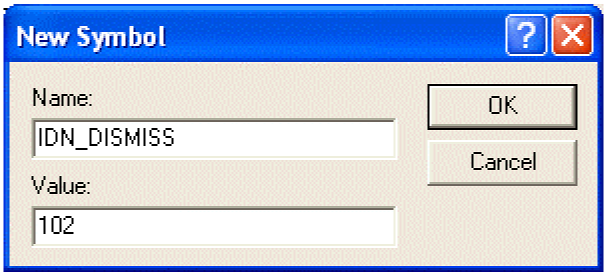
```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include f
// Used by Controls Fundamentals.rc
//
#define IDD_ABOUTBOX                       100
#define IDD_CONTROLSFUNDAMENTALS_FORM      101
#define IDR_MAINFRAME                      128
#define IDR_CONTROTYPE                     129
#define IDC_BUTTON1                        1000
#define IDC_LIST1                          1001
#define IDC_SCROLLBAR1                     1002
#define IDC_EDIT1                          1003

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_3D_CONTROLS                        1
#define _APS_NEXT_RESOURCE_VALUE           130
#define _APS_NEXT_COMMAND_VALUE            32771
#define _APS_NEXT_CONTROL_VALUE            1004
#define _APS_NEXT_SYMED_VALUE              101
#endif
#endif
```

If you visually add controls to your application, their identifiers are automatically added to this file. Even if you change the identifier of the control, Visual C++ updates this file. You can also manually add identifiers to this file, provided you know why you are doing that.

If you are programmatically creating a control using the **CreateWindow()** or the **CreateWindowEx()** functions, you do not need to specify the identifier. If you are programmatically creating a control, you can locally set an identifier by specifying a (randomly selected) decimal or hexadecimal number as the *nID* argument of the **Create()** or the second version of the **CreateEx()** methods. Here are examples:

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CWnd* btnApply = new CWnd;
	RECT Recto = {12, 16, 128, 64};
	btnApply->Create("BUTTON", "&Apply", WS_CHILD | WS_VISIBLE,
	         Recto, this, 0x10);

	CWnd *btnDismiss = new CWnd;
	btnDismiss->Create("BUTTON", "&Dismiss", WS_CHILD | WS_VISIBLE,
	           CRect(12, 68, 128, 120), this, 258);

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

Alternatively, you can first create an identifier in the String Table. We saw how to do this in Lesson 3. You can also create an identifier using the Resource Symbols dialog box. Then use that identifier for your control. Here is an example:



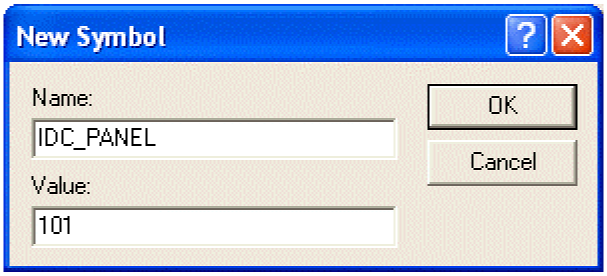
Name: IDN_DISMISS

Value: 102

```
BOOL CBordersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CWnd *btnDismiss = new CWnd;
	btnDismiss->Create("BUTTON", "&Dismiss", WS_CHILD | WS_VISIBLE,
		CRect(12, 68, 128, 120), this, IDN_DISMISS);

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

Practical Learning: Completing the Control

1. To create an identifier, in the Resource View, right-click the name of the appication, (MSVC 6) Controls resources or (MSVC 7) Controls, and click Resource Symbols…
2. In the Resource Symbols dialog box, click the New… button
3. Set the Name to IDC_PANEL



Name: IDC_PANEL

Value: 101

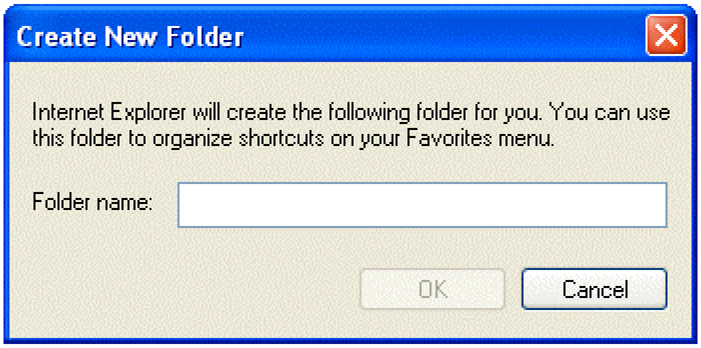
4. Click OK and click Close
5. On the dialog box, click the Custom control to select it. On the Properties window, click the arrow of the ID combo box and select IDC_PANEL
6. Test the application and return to MSVC

Chapter 12: Dialog-Based Windows

12.1 Dialog Boxes

12.1.1 Overview

A dialog box is a rectangular window whose main role is to host or hold other Windows controls. For this reason, a dialog box is referred to as a container. It is the primary interface of user interaction with the computer. By itself, a dialog box means nothing. The controls it hosts accomplish the role of dialog between the user and the machine. Here is an example of a dialog box:



Create New Folder

Internet Explorer will create the following folder for you. You can use this folder to organize shortcuts on your Favorites menu.

A dialog box has the following characteristics:

?? It is equipped with the system Close button ☒. As the only system button, this button allows the user to dismiss the dialog and ignore whatever the user would have done on the dialog box.

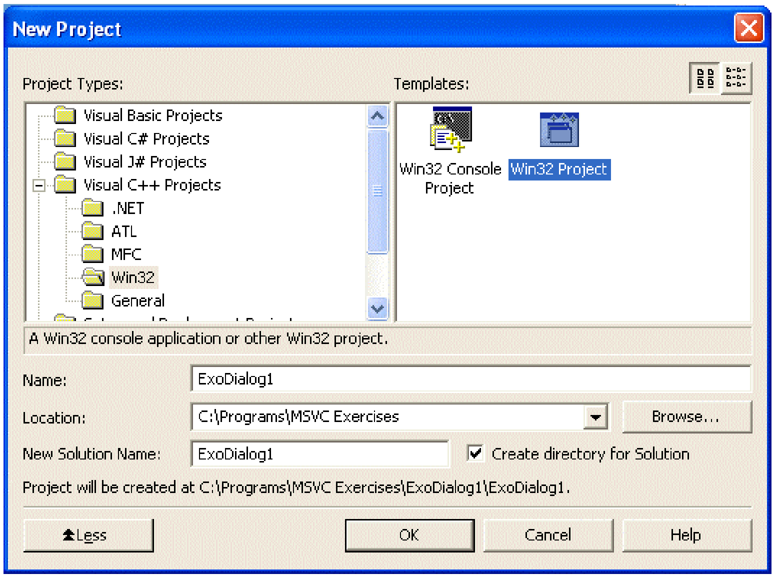
?? It cannot be minimized, maximized, or restored. A dialog box does not have any other system button but Close.

?? It is usually modal, in which case the user is not allowed to continue any other operation until the dialog box is dismissed.

?? It provides a way for the user to close or dismiss the dialog. Most dialog boxes have the OK and the Cancel buttons, although this depends on the application developer. When the dialog has the OK and the Cancel buttons, the OK button is configured to behave as if the user had pressed Enter. In that case, whatever the user had done would be acknowledged and transferred to the hosting dialog box, window, or application. Pressing Esc applies the same behavior as if the user had clicked Cancel.

Practical Learning: Creating an Application

1. Start Microsoft Visual C++ or Microsoft Visual Studio
2. Create a new Win32 Application or Win32 Project named **ExoDialog1**



A Win32 console application or other Win32 project.

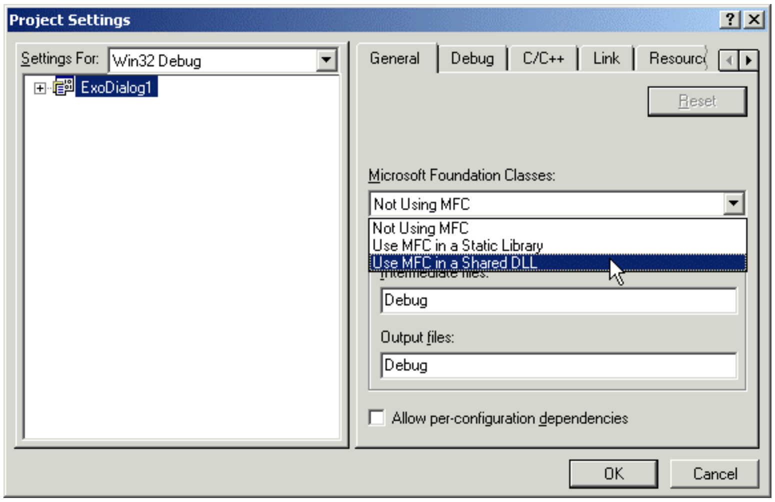
Name: ExoDialog1

Location: C:\Programs\MSVC Exercises

New Solution Name: ExoDialog1 ☑ Create directory for Solution

Project will be created at C:\Programs\MSVC Exercises\ExoDialog1\ExoDialog1.

3. Make sure you create the Windows Application in An Empty Project and click OK or Finish
4. If you are using MSVC 6, on the main menu, click Project -> Settings…
   If you are using MSVC 7, in the Solution Explorer, right-click ExoDialog1 and click Properties...
5. In the Property Pages, specify that you want to **Use MFC in a Shared DLL**



Settings For: Win32 Debug

ExoDialog1

Microsoft Foundation Classes: Not Using MFC → Use MFC in a Shared DLL

Intermediate files: Debug

Output files: Debug

□ Allow per-configuration dependencies

6. Click OK
7. To create the application class of the project, on the main menu, click File -> New or Project -> Add New Item...
8. In the Files property page or int the Templates section, click C++ (Source) File. Set the File Name to **Exercise** and click OK or Open.
9. To create the application, change the file as follows:

```
#include <afxwin.h>

class CExerciseApp : public CWinApp
{
public:
    BOOL InitInstance();
};

BOOL CExerciseApp::InitInstance()
{
    return TRUE;
}

CExerciseApp theApp;
```

10. Save All

12.1.2 Dialog Box Creation

To create a dialog box in MSVC, from the Add Resources dialog box, simply select the Dialog node and click New. Then you can manipulate its characteristics using the Properties window.

In reality, unlike most or all other controls, a dialog box requires some preparation before actually programmatically creating it. While most other controls can be easily added to a host, a dialog box must already “exist” before it is asked to display. Based on this, a dialog box can first be manually created as a text file. This is called a resource file. In the file, you create a section for the dialog box. This section itslef is divided in lines each serving a specific purpose.

Like most other controls, a dialog box must be identified. The identifer of a dialog box usually starts with IDD_ wher the second D stands for dialog. An example of a dialog identifer would be IDD_SCHOOL_SURVEY. Therefore, the section of the dialog box in the file can start with:

```
IDD_SCHOOL_SURVEY
```

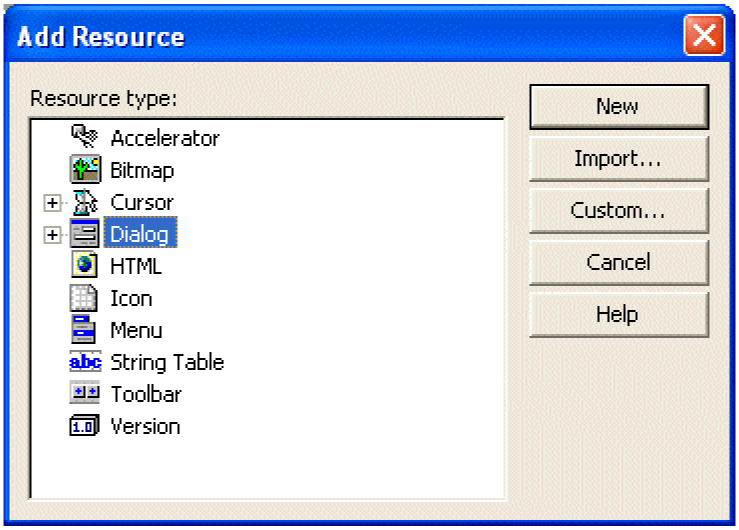
At design time, the identifier of the dialog box is specified using the ID combo box.

When having our introduction to controls, we saw that the Win32 library provides a series of ready-made names of classes to create controls. In the same way, a dialog box has two types of classes used to create a dialog box. The DIALOG statement was primarily used to create a dialog box. It has been updated and replaced by the DIALOGEX name. Therefore, to indicate in the file that the section you are creating is for a dialog box, type DIALOGEX. Here is an example:

```
IDD_SCHOOL_SURVEY DIALOGEX
```

Practical Learning: Creating a Dialog Box

1. To create a dialog box, on the main menu, click Insert -> Resource… or Project -> Add Resource...



2. In the Add Resource dialog box, click Dialog and click New
3. Click OK on the dialog box and press Delete twice to dismiss the OK and the Cancel buttons
   If you are using MSVC 6 and the Properties window is not displaying, right-click the dialog box and click Properties
4. In the Properties window, select IDD_DIALOG1. Type **IDD_EXERCISE_DLG** and press Enter

12.1.3 Dialog Box Location

Like the other controls and as we have reviewed their primary characteristics so far, a dialog box must be "physically" located on an application. Because a dialog box is usually created as a parent to other controls, its location depends on its relationship to its parent window or to the desktop. The location of a dialog box is defined by x for the distance from the left border of the monitor to the left border of the dialog box and by y for the distance from the top border of the monitor to the top border of the dialog box:

IDD_SCHOOL_SURVEY DIALOGEX *x*, *y*,

At design time, the x value is set using the **X Pos** field of the Properties window. The y value is set using the **Y Pos** value in the Properties window.

If you specify these two dimensions as 0, the left and top borders of the dialog box would be set so the object appears in the center-middle of the screen:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0,
```

If you set the value of *x* or **X Pos** property to a number less than or equal to 0, the left border of the dialog box would be aligned with the left border of the monitor. If you set the value of *x* or of the **X Pos** property higher than 0, the value would be used as the distance, in Dialog Box Unit (DLU), from the left border of the screen to the left border of the dialog box. This scenario also applies for the *y* value or the **Y Pos** property.

12.1.4 Dialog Box Dimensions

The dimensions of a dialog box are its width and its height. At design time, to set the desired width, position the mouse to its right border until the mouse pointer appears as a horizontal double arrow ↔ Then click and drag left or right.

To set the desired height, position the mouse to the bottom border until the mouse cursor appears as a vertical double arrow ↕. Then click and drag up or down.

To set both the width and the height in one operation, position the mouse in the bottom-right corner until the mouse pointer becomes a North-West – South-East diagonal double arrow ⤡. Then click and drag up, down, left, or right as necessary.

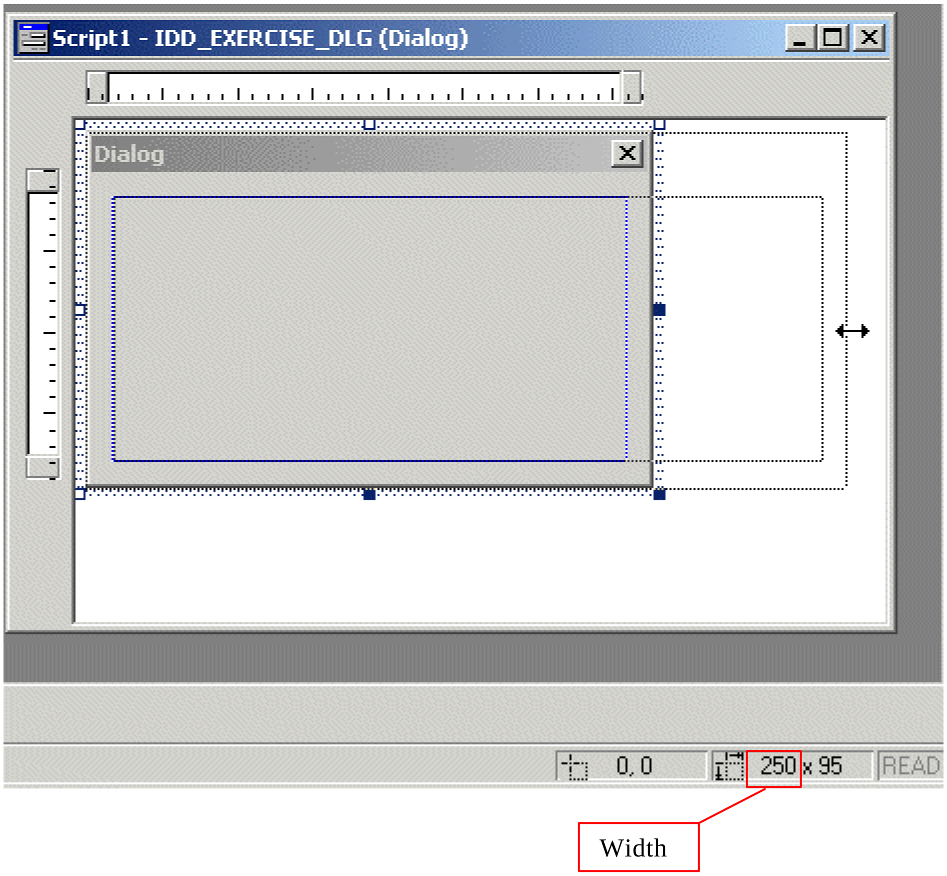
While dragging in any of these operations, you can refer to the right section of the status bar to get the current width and height. In the resource file, the width and the height of the dialog box follow the location values. Here is an example:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
```

There are other items that can be entered on this first line but the identifier, the **DIALOGEX** statement, the location, and the dimensions are required.

Practical Learning: Setting Dialog Box Dimensions

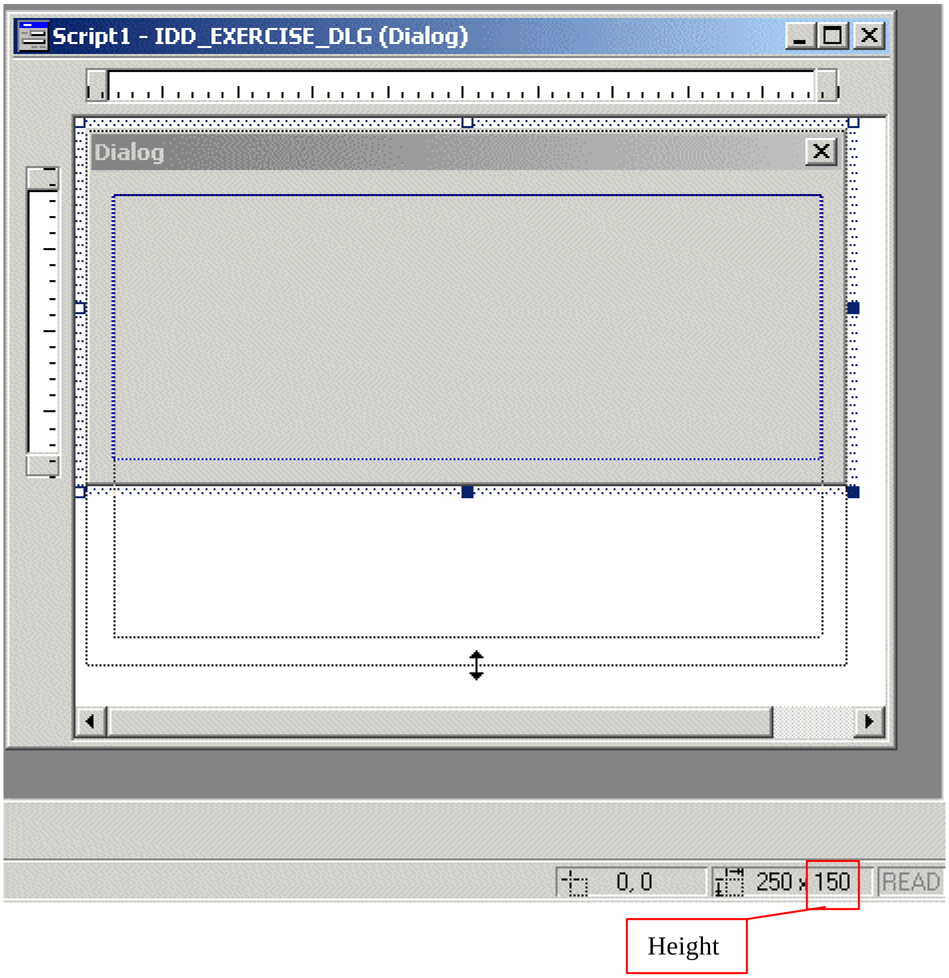
1. If you are using MSVC 6, close the Properties window
   Position the mouse to the right border of the dialog box with a horizontal mouse cursor ↔
2. Click and hold the mouse. Then drag in the right direction. Meanwhile, observe the change of the width on the Status Bar



3. When the width value gets to 250, release the mouse

4. Position the mouse to the right border of the dialog box with a vertical mouse cursor ↕

5. Click and hold the mouse. Then drag in the right direction. Meanwhile, observe the change of the height on the Status Bar



6. When the height values gets to 150, release the mouse
7. If you are using MSVC 6, display the Properties window

12.1.5 Windows Styles for a Dialog Box

As the most regular used parent of Windows controls, a dialog box must have some characteristics that would allow it to be as efficient as possible. The characteristics of a dialog box, as done with the controls, are referred to as its styles. In the resource file, to specify these aspects, start a line with the **STYLE** keyword:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE
```

WS_CAPTION: Because a dialog box serves only to hold other objects, it should indicate what it is used for. The top side of a dialog box is made of a horizontal section that we call the title bar. To have a title bar on a dialog box, it must be created with the **WS_CAPTION** style:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION
```

The main area of this bar contains a word or a group of words forming a sentence called a caption. There are various ways you can change it. At design time, on the Properties window, change the value of the **Caption** field. If you are creating a resource file, start a new line with the **CAPTION** keyword and provide the word or group of words as a null-terminated string. Here is an example:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION
CAPTION "School Survey – For Teachers Only"
```

WS_SYSMENU: By convention, a dialog box is equipped with only the system **Close** button ☒ on its title bar. This is made possible by adding the **WS_SYSMENU** style:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU
CAPTION "School Survey – For Teachers Only"
```

WS_MINIMIZEBOX: The **Minimize** and **Maximize** buttons are omitted. If you want to add the **Minimize** button ▁, at design time, set the **Minimize Box** property to True or checked. This is equivalent to adding the **WS_MINIMIZEBOX** style to the class.

WS_MINIMIZE: If you create a dialog box and equip it with the Minimize button so the user can shrink it to the Taskbar, if you want the dialog box to be minimized when it comes, you can open the resource file and add the **WS_MINIMIZE** style. This would be done as follows:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MINIMIZE
CAPTION "School Survey – For Teachers Only"
```

Remember that a dialog box can be minimized only if its system menu indicates so.

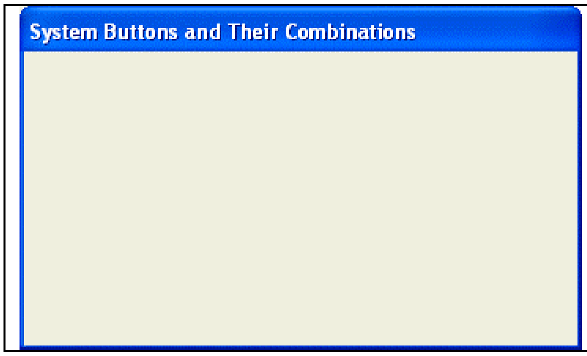
WS_MAXIMIZEBOX: If you want the **Maximize** button □, set the **Maximize Box** property to **True** or checked. This is equivalent to adding the **WS_MAXIMIZEBOX** style.

WS_MAXIMIZE: If you create a dialog box equipped with the Maximize button, if you want the dialog box to be automatically maximized when it displays, you can add the **WS_MAXIMIZE** style to the resource file. This would be done as follows:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MINIMIZE |
                                          WS_MINIMIZEBOX | WS_MINIMIZE
CAPTION "School Survey – For Teachers Only"
```

Remember that a window box can be maximized only if it includes the **WS_MAXIMIZEBOX** style.

Setting one of these two system button properties to True or checked or adding either **WS_MAXIMIZEBOX** or **WS_MINIMIZEBOX** and not the other would disable the other system button. Therefore, here are the combinations you can get:

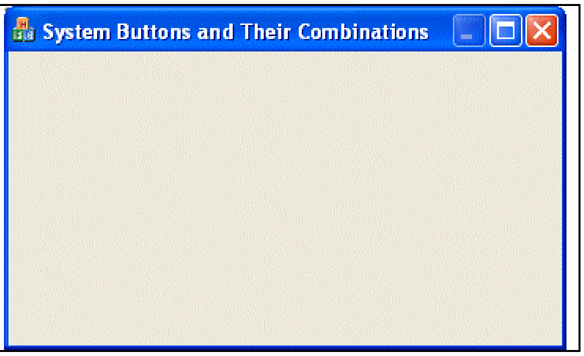


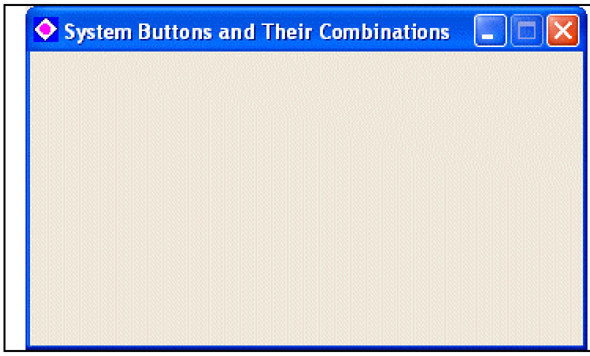
System Menu = False

There is no system button

System Menu = True
Minimize Box = False
Maximize Box = True

The Minimize button is disabled



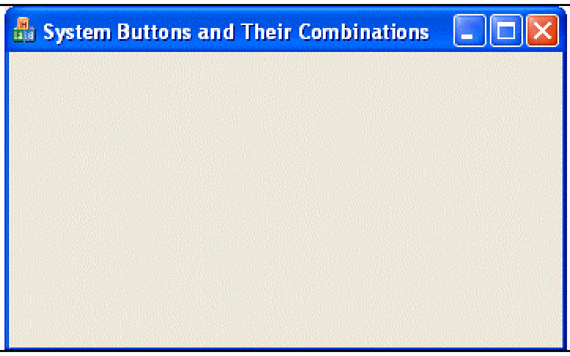


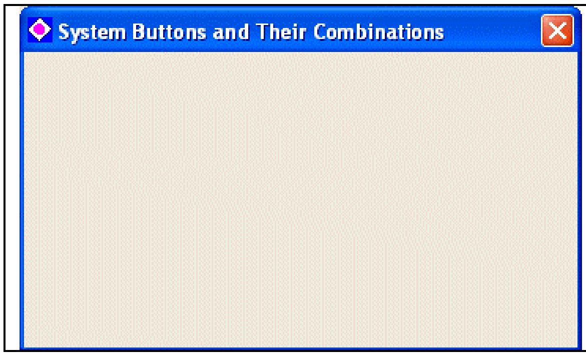
System Menu = True
Minimize Box = True
Maximize Box = False

The **Maximize** button is disabled

System Menu = True
Minimize Box = True
Maximize Box = True

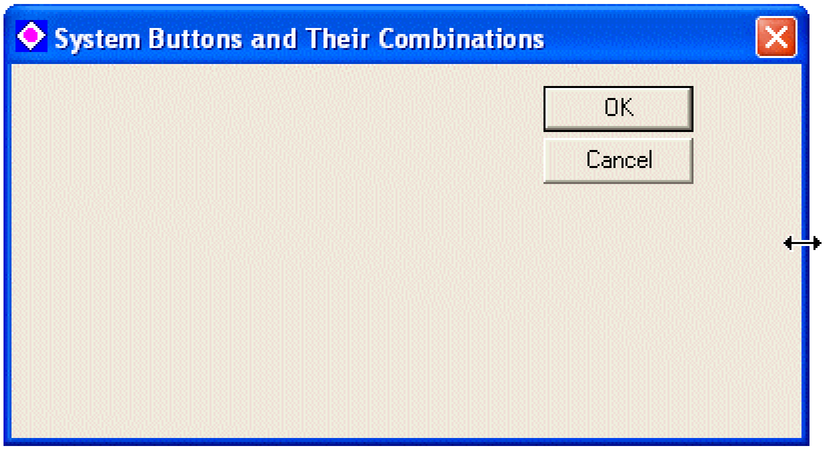
Both the **Minimize** and **Maximize** buttons are enabled



	**System Menu = True** **Minimize Box = False** **Maximize Box = False** Only the **Close** button is available

BS_BORDER: The borders of a control allow you and the user to "physically" locate it on the screen. This can be done by adding the **WS_BORDER** to a control. For a dialog box, the type of border used on the object is controlled by the Border combo box

WS_THICKFRAME: Based on the Microsoft Windows standards and suggestions, a dialog box usually uses constant dimensions. This means that the user should not change its width and/or height. If you want to give the user the ability to resize a dialog box, change its Border value to **Resizing**. With this type of object, the user can position the mouse on of its borders or corners, click, hold down the mouse and drag in the desired direction to change its dimensions:



In the resource file, this can also be done by creating the dialog box with the **WS_THICKFRAME** style:

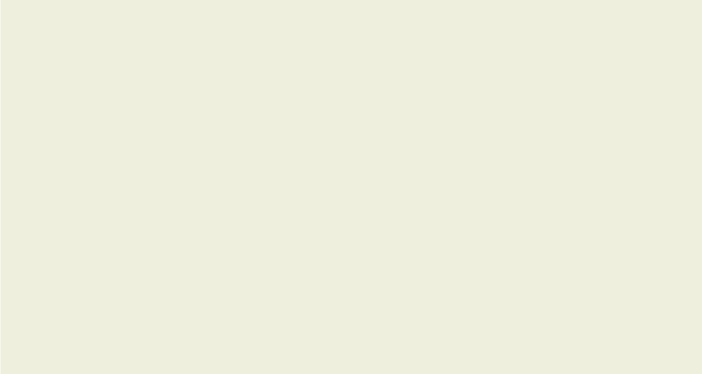
```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | WS_THICKFRAME
CAPTION "School Survey – For Teachers Only"
```

A **Thin** value gives a thin border to the dialog box.

WS_POPUP: A window is referred to as popup if it can be displayed on the screen. That is, if it can be "physically" located. To create a popup dialog box, at design time, select **Popup** from the **Style** combo box. In a resource file, this can be done by adding the **WS_POPUP** style:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP
CAPTION "School Survey – For Teachers Only"
```

If you do not want borders at all on the dialog box, set its **Border** value to **None**. This also removes the title bar and thus the system buttons:



This is equivalent to using only the **WS_POPUP** style:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_POPUP
CAPTION “School Survey – For Teachers Only”
```

WS_CHILD: A window is referred to as child if its appearance is dependent of the appearance of another window. All of the Windows controls that we will be placing on our dialog boxes are child controls. As we will see when studying property sheets and wizards, a dialog box that is designed to be “embedded” in a property sheet or a wizard must be created as a child. To specify that a dialog box is a child of another window, at design time, select the Child value on the Style combo box. This characteristic can be applied by adding the **WS_CHILD** style.

WS_OVERLAPPED: A window is called overlapped if it has the following characteristics:

?? It has a title bar equipped with a caption and at least one of the system buttons (usually at least the system Close button)

?? It has borders

To get an overlapped dialog box, at design time, open the **Style** comb box and select the **Overlapped** value. This is equivalent to the **WS_OVERLAPPED** style.

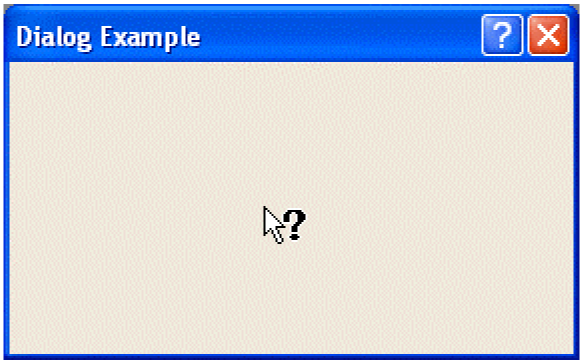
WS_OVERLAPPEDWINDOW: To create a dialog box equipped with a title bar, a system menu, the ability to popup, and a border, instead of combining the WS_CAPTION, the WS_SYSMENU, the WS_POPUP, and the WS_BORDER styles, use the **WS_OVERLAPPEDWINDOW** value, which combines them all.

Practical Learning: Changing a Dialog Style

1. In the properties window, delete the value in the Caption field and replace it with **Aerobics Exercises** and press Enter
2. Click the arrow of the **Style** combo box and select **Overlapped**

12.1.6 Dialog Box Styles

DS_CONTEXTHELP: We saw already how to manage the title bar and its system buttons. If you are creating a strict dialog box, one that is equipped with only the system close button ☒, if you intend to provide help on the dialog box, you can add a context-sensitive help button to the left of the close button. To do this, set its Context Help property to True or create it with the **DS_CONTEXTHELP** style:



DS_MODALFRAME: By design, a dialog box has thick borders and cannot be resized from its borders or corners, even if the system buttons are available. This characteristic of dialog boxes is controlled by the **Border** property. The default border of a dialog box is **Dialog Frame**. This is the same as applying the **DS_MODALFRAME** dialog style:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME
CAPTION “School Survey – For Teachers Only”
```

DS-3DLOOK: In previous versions of MS Windows, namely Windows NT 3.51, to get a 3-dimensional appearance, you needed to apply the **DS-3DLOOK** style. At this time, this style is added by default.

DS_ABSALIGN: Earlier, when reviewing the techniques of setting the location of a dialog box, we mentioned that, if the *x* and the *y* values, or the **X Pos** and the **Y Pos** properties, are set to 0, the dialog box is positioned in the middle of the screen. This is because, like all other controls that can act as children of a parent window, the location of a dialog box is set according to its parent. That is, the location of a dialog box is based on its ability to be a client (also called child) of another window (called a parent). This system is referred to as ***Client Coordinate***. If you are creating an application based on a dialog box, that is, an application whose main or even only container is the dialog box, such a dialog box does not have a parent. In this case, it does not have a referential window to base its location on. Consequently, it gets positioned to the middle of the screen. If you want the location of the dialog box to be based on the monitor screen, set its **Absolute Align** property to True. In a resource file, this would be done by adding the **DS_ABSALIGN** style. This system is referred as the ***Screen Coordinate***.

DS_CENTER: Based on the coordinate system you decide to use, screen or client, when the dialog box of a dialog-based application comes up, if you want the dialog to be centered on the monitor screen, even if you are using the screen coordinate system with the **DS_ABSALIGN** style, set its **Center** property to True or checked. This is also done by adding the **DS_CENTER** style. If the dialog box is participating in an application that is view-based, if you set its **Center** property to True, it would be centered with regard to the window that called it.

DS_CENTERMOUSE: We have described how to control the location of the dialog box when it comes up, depending on the coordinate system you are using, and the **DS_CENTER** style. Alternatively, if you want the dialog box to be displayed based on the location of the user's mouse cursor, set its Center Mouse property to True or checked. In this case, when the dialog box comes up, it would find the mouse pointer and position itself so the mouse cursor would be in the middle-center of the dialog box. This ability is also applied by adding the **DS_CENTERMOUSE** style.

Font: To display its controls, the dialog box uses a predefined font known as **MS Shell Dlg**. To change it at design time, if you are using MSVC 6, click the Font button in the General property page of the Properties window. In MSVC 7, you can click the ellipsis button of the Font (Size) field. The font characteristics are managed through the **DS_SHELLFONT**, the **DS_SETFONT**, and the **DS_FIXEDSYS** styles. The section of the resource file that specifies the general font starts with the FONT keyword:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME
CAPTION "School Survey – For Teachers Only"
FONT
```

The FONT keyword is followed by the characteristics of the font using the following syntax:

FONT *size*, *name*, *weight*, *italic*, *charset*

Here is an example:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME
CAPTION "School Survey – For Teachers Only"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
```

If you want to manage the fonts, weights, and colors of the controls on the dialog box of an MFC application, you must do this programmatically.

Practical Learning: Using Dialog Box Properties

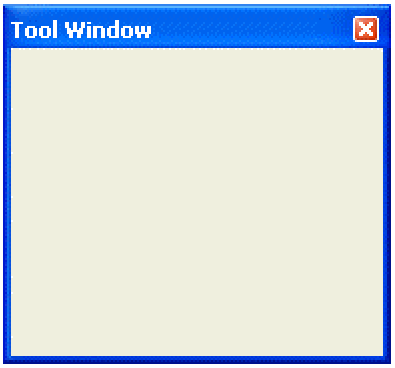
1. In the properties window, click the arrow of the **Border** combo box and select **Resizing**
2. Click the **Center** check box or set it to True

12.1.7 Extended Windows Styles for a Dialog Box

Windows extended styles are used to improve the appearance or the role of a dialog box on an application. To use these styles in the resource file, start a new line with the EXSTYLE keyword:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME
EXSTYLE
CAPTION "School Survey – For Teachers Only"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
```

WS_EX_TOOLWINDOW: A tool window is a parent object mostly used to float, like a toolbar, on a frame of another window. It has a short title bar that displays its caption. It can neither be minimized nor maximized but it can be equipped with a circumstantial system menu:



To create a tool window, when designing the dialog box, check the Tool Window check box or set it to True. This can also be applied by adding the **WS_EX_TOOLWINDOW** extended style:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME
EXSTYLE WS_EX_TOOLWINDOW
CAPTION “School Survey – For Teachers Only”
FONT 8, “MS Shell Dlg”, 400, 0, 0x1
```

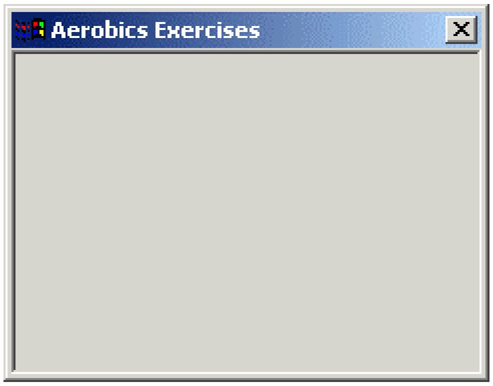
If you want the user to be able to resize the tool window by dragging its borders or corners, at design time, set its Border value to Resizing or create it with the WS_THICKFRAME style. If you do not want the user to be able to resize the tool window, at design time, set its Border to either Thin or Dialog Frame. In the resource file, you can also do this by creating it with the **WS_CAPTION** and **WS_SYSMENU** style:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                        LPSTR lpCmdLine, int nCmdShow)
{
    HWND          hWnd;
    MSG           Msg;
    WNDCLASSEX    WndClsEx;

    . . .

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx(WS_EX_TOOLWINDOW,
                          ClsName,
                          WndName,
                          WS_CAPTION | WS_SYSMENU,
                          . . .,
                          );
}
```

WS_EX_APPWINDOW: As mentioned already, a window without the system Minimize button cannot be minimized. This, of course, is the same for a tool window. Also, by default, a tool window does not display a button on the taskbar. If you are creating a window that will depend on a main window such as a frame, you should keep that default. If for any reason you want to display a button on the Taskbar for the tool window, add the **WS_EX_APPWINDOW** extended style.

WS_EX_TOPMOST: If you want your window to stay on top of another or other windows, you have two choices. If the window you are creating, such as a dialog box, is being added to a frame-based or a form-based application and you want that window to always be on top of its parent window, add the **WS_VISIBLE** style to it and display it using **ShowWindow(SW_SHOW)**. If you want the window to always be on top of all other windows regardless of their applications, add the **WS_EX_TOPMOST** extended style to it.

WS_EX_CLIENTEDGE: A window appears sunken when its body is sunk with regards to its borders. To create a dialog box with a sunken body, at design time, check its **Client Edge** check box or set it to **True**:



This is equivalent to applying the **WS_EX_CLIENTEDGE** style

WS_EX_WINDOWEDGE: A window can have an inside body that appears to be raised with regards to its borders. Such a window can be created by adding the **WS_EX_WINDOWEDGE** style.

WS_EX_STATICEDGE: To create a window with a 3-dimensional thin border, at design time, check its Static Edge check box or set it to **True**. This is the same as adding the **WS_EX_STATICEDGE**.

WS_EX_PALETTEWINDOW: To create a dialog box that has a raised border and stays on top of others, which is equivalent to combining the **WS_EX_WINDOWEDGE** and the **WS_EX_TOPMOST** styles, in the resource file, add the **WS_EX_PALETTEWINDOW** value in the **EXSTYLE** line

WS_EX_OVERLAPPEDWINDOW: A window is referred to as overlapped when it presents a combination of a sunk and a raised edge. Such a window can be created by either combining the **WS_EX_CLIENTEDGE** and the **WS_EX_WINDOWEDGE** styles or by using the **WS_EX_OVERLAPPEDWINDOW**.

WS_EX_ACCEPTFILES: If you are creating a dialog box that will be accepting files dragged from other window, at design time, check the Accept Files check box or set it to **True**. You can also do this in the resource file by adding the **WS_EX_ACCEPTFILES** extended style.

Practical Learning: Creating a Top-Most Window

1. To create the dialog box with a thin border, check its **Static Edge** check box or set it to True
2. To indicate that this dialog can accept dragged files, check its **Accept Files** check box or set it to True

12.1.8 Creating the Dialog Resource File

As mentioned already, a dialog box means nothing except for the controls it is created to host. The controls are listed in a section that starts with the **BEGIN** and ends with the **END** keywords. Here is an example:

```
IDD_SCHOOL_SURVEY DIALOGEX 0, 0, 340, 268
STYLE WS_CAPTION | WS_SYSMENU | WS_POPUP | DS_MODALFRAME
EXSTYLE WS_EX_TOOLWINDOW
CAPTION "School Survey – For Teachers Only"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
END
```

If you design a dialog box and the first object of your application, you must save it before using it. Even if you had decided to manually create the dialog box as a text file, you must save it before using it. In both cases, whether saving the dialog box designed or the text file, saving it results in creating a resource file. This file should have the .rc extension.

If your resource file is using some identifiers, which is the case for most or all controls you will use in your application, you must list them in a header file. This file is traditionally called Resource.h or resource.h. Each identifier must be listed using the following syntax:

#define *Identifier Constant*

When you save the resource file, Visual C++ automatically creates the resource header file and names it resource.h

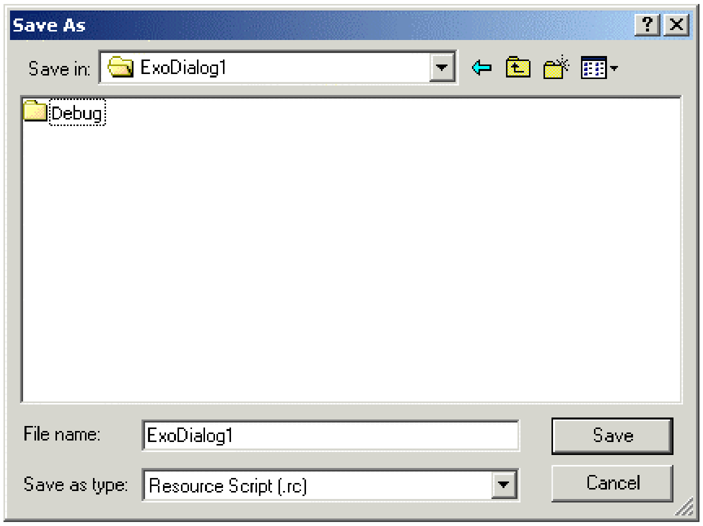
After saving the resource file, it is still considered external to the application. If you are using MSVC 6, you must add it to your project. After it has been added to your project, you can continue working on the dialog box, adding, deleting, or manipulating the controls. After doing that, if you save the resource, Visual C++ automatically makes the appropriate changes in the resource and the resource header files.

Practical Learning: Saving the Resource File

1. If you are using MSVC 7, on the Standard toolbar, click the Save All button
   If you are using MSVC 6:
   a. To save the dialog box, click the system Close button of the window that holds the dialog box.
   b. This will display a window with a tree view that starts with Script1. Close its window also.

c. You will receive a message box asking you whether you want to save the script. Click Yes

d. Locate the ExoDialog1 folder that was used to create the current project. Display it in the Save In combo box.
Change the name in the File Name with ExoDialog1



e. Click Save

2. To add the resource to your application, on the main menu, click Project -> Add To Project -> Files…
3. Click ExoDialog1.rc and click OK
4. To verify that the dialog box has been added, in the Workspace, click the **ResourceView** tab and expand the ExoDialog1 Resources node. Then expand the Dialog folder and double-click IDD_EXERCISE_DLG
5. If you are using MSVC 7, on the Standard toolbar, the Save All. Then

12.1.9 Creating a Class for the Dialog

After creating the resource for the dialog box, you must create a class that will be used to handle its assignment. To do this, you must derive a class from CDialog. In the header file of the dialog's class, define an enumerator whose only member is called IDD and initialize it with the identifier of the dialog box. Because the identifier is listed in the resource header file, you must include this resource header in the file in which you are using the dialog's identifier.

Practical Learning: Create the Dialog's Class

1. To create a class for the dialog box, open the Exercise.cpp created earlier and add the following:

```
#include <afxwin.h>
#include <afxdlgs.h>
#include "resource.h"

class CExerciseApp : public CWinApp
{
public:
    BOOL InitInstance();
};

class CExerciseDlg : public CDialog
{
public:
    enum { IDD = IDD_EXERCISE_DLG };
};

BOOL CExerciseApp::InitInstance()
{
    return TRUE;
}

CExerciseApp theApp;
```

2. Save All

12.1.10 Dialog Box Methods

A dialog box is based on the **CDialog** class. As seen above, when creating your dialog box, you can derive a class from **CDialog**. The **CDialog** class itself provides three constructors as follows:

```
CDialog();
CDialog(UINT nIDTemplate, CWnd* pParentWnd = NULL);
CDialog(LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL);
```

The default constructor, **CDialog()**, can be used to declare a variable whose behavior is not yet known or, for one reason or another, cannot yet be defined. When creating your class, you should also declare at least a default constructor.

The identifier of the dialog box, such as IDD_DIALOG1, can be used as the first argument, *nIDTemplate*, of a **CDialog()** constructor to create a dialog box from an existing resource.

If you are using a Win32 template to create your dialog box, pass the name of this template as a string to a **CDialog()** constructor, *lpszTemplateName*.

When implementing your default constructor, initialize the parent **CDialog** constructor with the IDD enumerator declared in your class. If your dialog box has an owner, specify it as the pParentWnd argument. If you set it to **NULL**, the application will be used as the dialog's parent.

If you dynamically create objects for your application using your dialog class, it is a good idea to also declare and define a destructor. This would be used to destroy such dynamic objects.

Most other methods of a dialog box depend on circumstances we have not yet reviewed

Practical Learning: Creating a Dialog Box

1. Declare a default constructor and a destructor for your dialog class and implement them as follows:

```
class CExerciseDlg : public CDialog
```

```
{
public:
    enum { IDD = IDD_EXERCISE_DLG };

    CExerciseDlg();
    ~CExerciseDlg();
};

CExerciseDlg::CExerciseDlg()
    : CDialog(CExerciseDlg::IDD)
{
}

CExerciseDlg::~CExerciseDlg()
{
}
```

2. Before using the new class, declare a variable of it as follows:

```
BOOL CExerciseApp::InitInstance()
{
    CExerciseDlg Dlg;

    m_pMainWnd = &Dlg;

    return TRUE;
}
```

3. Save All

12.2 Modal Dialog Boxes

12.2.1 Dialog-Based Applications

There are two types of dialog boxes: modal and modeless. A Modal dialog box is one that the user must first close in order to have access to any other framed window or dialog box of the same application.

One of the scenarios in which you use a dialog box is to create an application that is centered around a dialog box. In this case, if either there is no other window in your application or all the other windows depend on this central dialog box, it must be created as modal. Such an application is referred to as dialog-based

There are two main techniques you can use to create a dialog-based application: from scratch or using one of Visual C++ wizards. After creating a dialog resource and deriving a class from CDialog, you can declare a variable of your dialog class. To display your dialog box as modal, you can call the **CDialog::DoModal()** method in your **CWinApp::InitInstance()** method. Its syntax is:

```
virtual int DoModal();
```

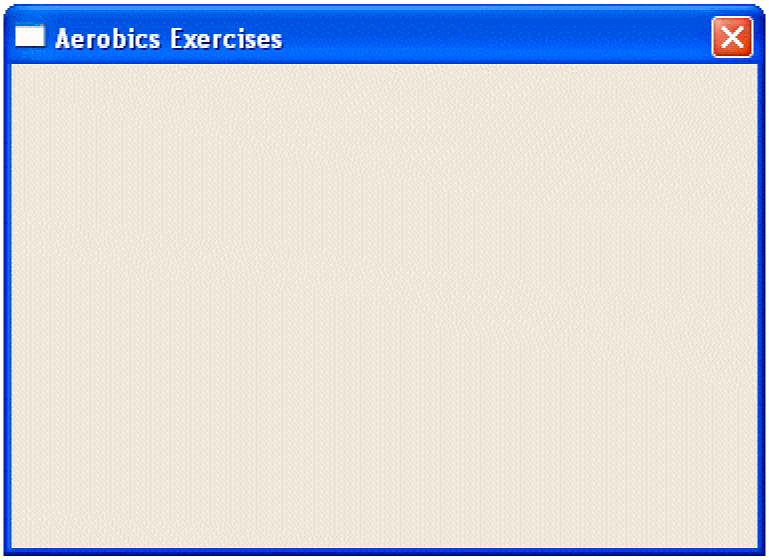
This method by itself does nothing more than displaying a dialog box as modal. We will learn that you can use this method to find out how the user had closed such a dialog box.

Practical learning: Displaying a Modal Dialog Box

1. To display the dialog box as modal, in the **InitInstance()** event of your **CWinApp** derived class, call the **DoModal()** method using your dialog variable:

```
#include <afxwin.h>
#include <afxdlgs.h>
#include "resource.h"

class CExerciseApp : public CWinApp
{
public:
	BOOL InitInstance();
};

class CExerciseDlg : public CDialog
{
public:
	enum { IDD = IDD_EXERCISE_DLG };

	CExerciseDlg();
	~CExerciseDlg();
};

CExerciseDlg::CExerciseDlg()
	: CDialog(CExerciseDlg::IDD)
{
}

CExerciseDlg::~CExerciseDlg()
{
}

BOOL CExerciseApp::InitInstance()
{
	CExerciseDlg Dlg;

	m_pMainWnd = &Dlg;
	Dlg.DoModal();

	return TRUE;
}

CExerciseApp theApp;
```

2. Test the application



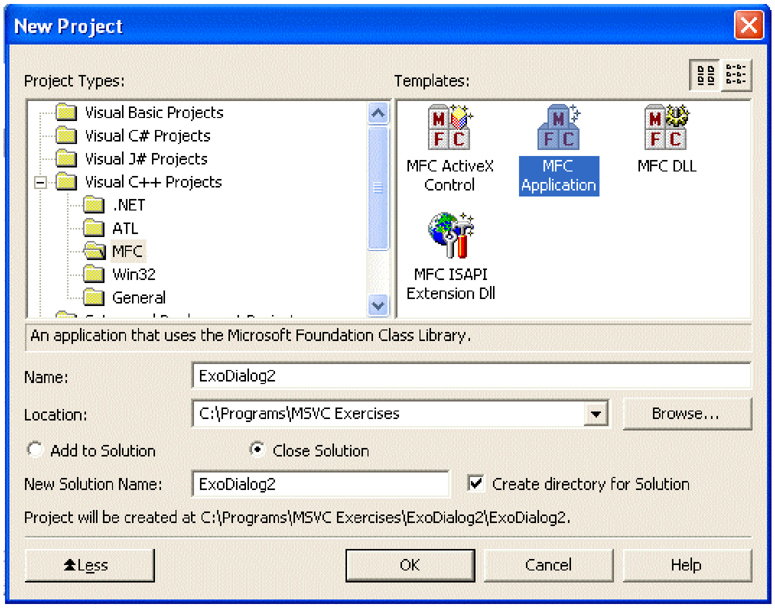
3. Close it and return to MSVC

12.2.2 The MFC Wizard for a Dialog-Based Application

Microsoft Visual C++ provides an easier way to create an application that is mainly based on a dialog box. To use this technique, start a new project and specify that you want to create an MFC Application. In the MFC Application Wizard, set the Application Type to Dialog Based

Practical Learning: Using the Wizard to create a Dialog-Based Application

1. On the main menu, click File -> New -> Project...
2. In the New Project dialog box, in the Templates list, click **MFC Application**
3. Set the Project Name to **ExoDialog2**



An application that uses the Microsoft Foundation Class Library.

Name: ExoDialog2

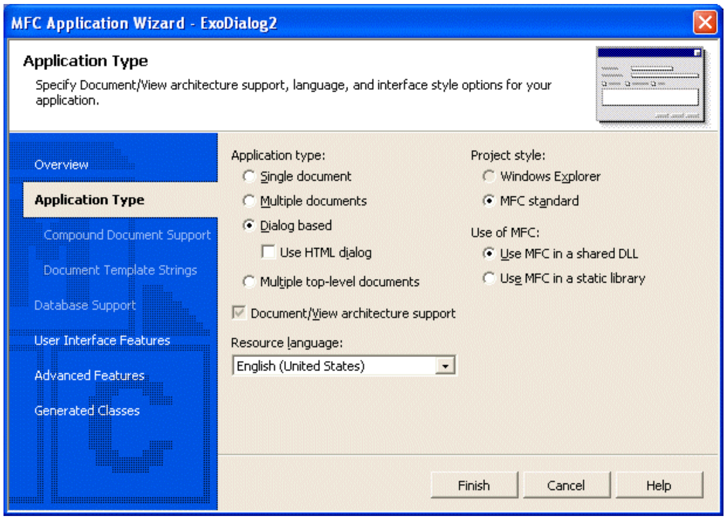
Location: C:\Programs\MSVC Exercises

○ Add to Solution ● Close Solution

New Solution Name: ExoDialog2 ☑ Create directory for Solution

Project will be created at C:\Programs\MSVC Exercises\ExoDialog2\ExoDialog2.

4. Click OK
5. In the MFC Application Wizard dialog box, click Application Type and, on the right side, click the **Dialog Based** radio button



MFC Application Wizard - ExoDialog2

Application Type

Specify Document/View architecture support, language, and interface style options for your application.

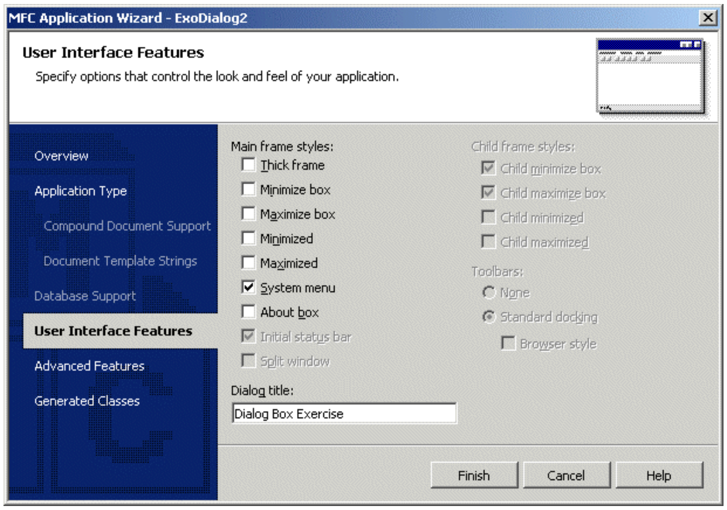
Application type:
○ Single document
○ Multiple documents
● Dialog based
□ Use HTML dialog
○ Multiple top-level documents
☑ Document/View architecture support

Resource language: English (United States)

Project style:
○ Windows Explorer
● MFC standard

Use of MFC:
● Use MFC in a shared DLL
○ Use MFC in a static library

6. Click **User Interface Features** and click **About Box** to remove its check box
7. Under **Dialog Title**, select the text and replace it with **Dialog Box Exercise**



MFC Application Wizard - ExoDialog2

User Interface Features

Specify options that control the look and feel of your application.

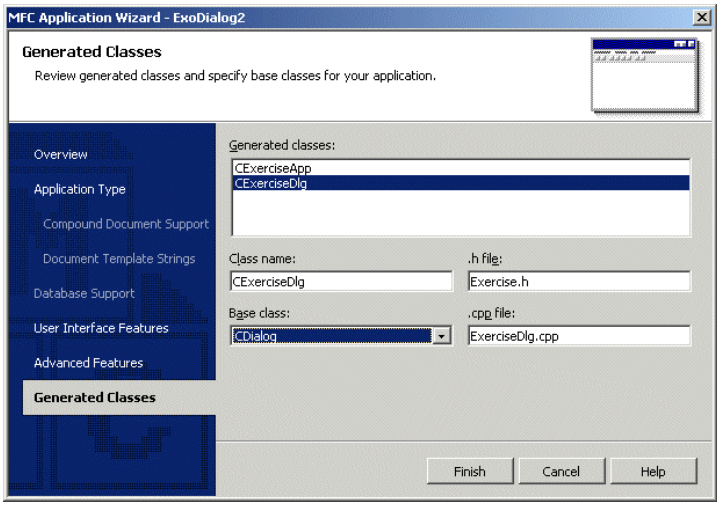
Main frame styles:
□ Thick frame
□ Minimize box
□ Maximize box
□ Minimized
□ Maximized
☑ System menu
□ About box
☑ Initial status bar
□ Split window

Dialog title: Dialog Box Exercise

Child frame styles:
☑ Child minimize box
☑ Child maximize box
□ Child minimized
□ Child maximized

Toolbars:
○ None
● Standard docking
□ Browser style

8. Click Advance Features to see its content
9. Click Generated Classes
10. In the Generated Classes list, click CExoDialog2App and, in the Class Name, replace the text with **CExerciseApp**
11. In the Generated Classes list, click CExoDialog2Dlg
    In the Class Name, replace the name with **CExerciseDlg**
    In the .h file edit box, replace the name of the file with **ExerciseDlg.h**
    In the .cpp file edit box, replace the name of the file with **ExerciseDlg.cpp**
    Make sure the Base Class is set to **CDialog**



MFC Application Wizard - ExoDialog2

Generated Classes

Review generated classes and specify base classes for your application.

Generated classes: CExerciseApp, CExerciseDlg

Class name: CExerciseDlg

.h file: Exercise.h

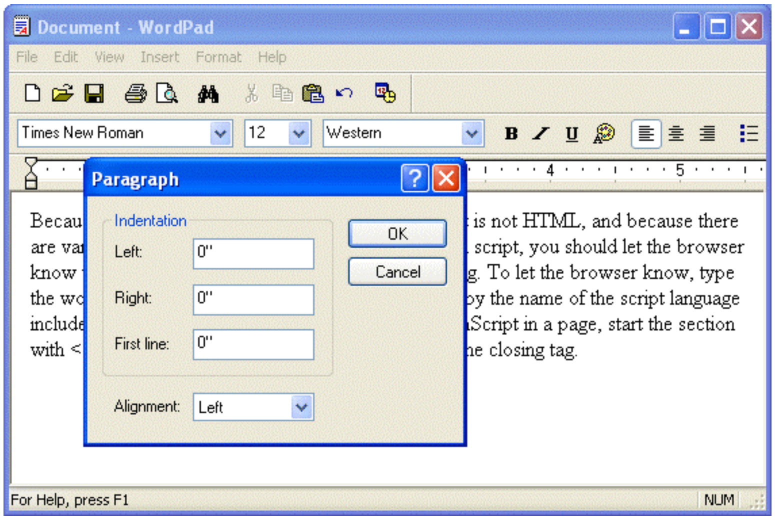
Base class: CDialog

.cpp file: ExerciseDlg.cpp

12. Click Finish
13. Test the application and return to MSVC

12.2.3 A Modal Dialog Box in an Application

Some applications require various dialog boxes to complete their functionality. When in case, you may need to call one dialog box from another and display it as modal. Here is an example:



Document - WordPad

Paragraph

Indentation

Left: 0"

Right: 0"

First line: 0"

Alignment: Left

File
Edit
View
Insert
Format
Help
Times New Roman
12
Western
OK
Cancel
Becau
are va
know
the wo
include
with <
is not HTML, and because there
script, you should let the browser
g. To let the browser know, type
by the name of the script language
Script in a page, start the section
ne closing tag.
For Help, press F1
NUM

The Paragraph dialog box of WordPad is a modal dialog box: when it is displaying, the user cannot use any other part of WordPad unless he or she closes this object first

Since a dialog box is created using a class, the first thing you must do is to include the header file of the **CDialog** object whose box you want to call.

To call one dialog box from another window, select the event (or message) from where you would make the call. Declare a variable of the other class and use the **CDialog::DoModal()** method to display the other object as a modal dialog box.

Calling a dialog box from the menu of a frame-based application is done the same way as calling it from a dialog box

Practical Learning: Adding a Dialog Box to an Existing Application

1. Start a new MFC Application project named **SDIAndDlg**
2. Create it as a Single Document and click Finish
3. To add a dialog box, on the main menu, click Project -> Add Resource…
4. In the Add Resource dialog box, double-click Dialog
5. To create a class for the new dialog box, right-click it and click Add Class…
6. In the MFC Class Wizard, specify the Class Name as **CExerciseDlg**
7. In the Base Class combo box, select CDialog and click Finish
8. Display the menu by double-clicking IDR_MAINFRAME under the Menu folder of the Resource View tab
9. Click View and click the first empty item under it. Type – and press Enter to add a separator.
10. In the menu item under the new separator, type **&Exercise…** and press Enter
11. Right-click the new Exercise menu item and click Add Event Handler…
12. In the Message Type, access the COMMAND item. In the Class List, click CMainFrame. Accept the Function Handler Name then click Finish Add And Edit
13. Change the file as follows:

```
#include "stdafx.h"
#include "SDIAndDlg.h"

#include "MainFrm.h"
#include "ExerciseDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

. . .

void CMainFrame::OnViewExercise()
{
	// TODO: Add your command handler code here
	CExerciseDlg Dlg;

	Dlg.DoModal();
}
```

14. Execute the application. On its main menu, click View -> Exercise…
15. After using the dialog box, close it and close the application to return to MSVC

12.3 Property Sheets and Wizards

12.3.1 Introduction to Property Pages

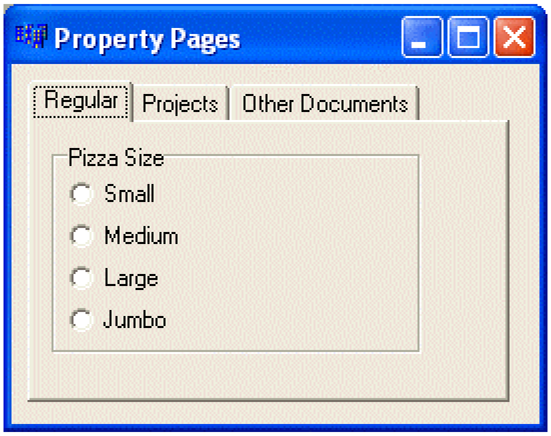
In some applications, you may want to add many but necessary and useful controls to a dialog box. You can solve this problem in three main ways:

?? You may have to tremendously increase the width and the height of your dialog box. Although this solution works sometimes, it may have the disadvantage of producing a highly crowded dialog box

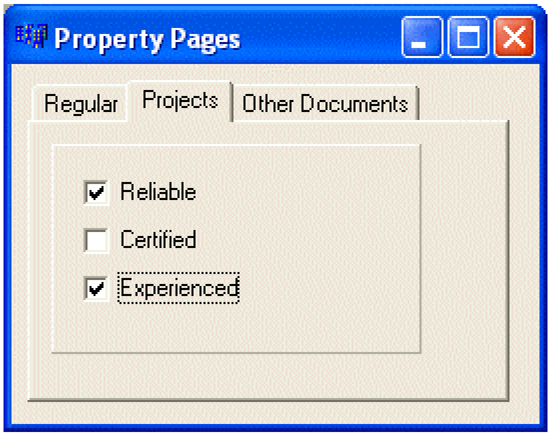
?? You can hide some controls and display them only when needed, such as in response to a user clicking a button. The concept of this type of application involves an unpredictable and non-practical design. It also eventually requires a lot of coding

?? The third solution involves the user of property pages

A property page is a dialog box, that can be positioned in front of, or behind of, another. This has the advantage of providing various dialog boxes that are "physically" grouped as one entity. Each part is represented by a tab. The tabs are usually positioned in the top section and each is used to identify a particular page:



To use a property page, the user clicks one. The page clicked besomes positioned in front of the other(s). The user can click another tab to select a different page:



12.3.2 Creating Property Pages

A property page is designed from a dialog box and it must have the following characteristics:

?? Style: Child or **WS_CHILD**

?? Border: Thin or **WS_POPUP**

?? Title Bar: True or **WS_CAPTION**

?? System Menu: False or no **WS_SYSMENU**

?? Visible: False or no **WS_VIS IBLE**

?? Disabled: True or **WS_DISABLED**

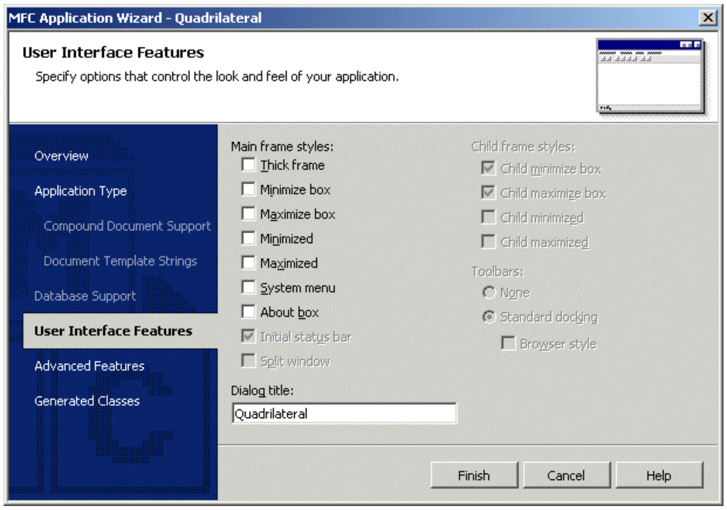
You can create each property page like that and with the same size. If you create dialog boxes that have different sizes, the dimensions of the taller and/or wider will be applied to all the other property pages when they come up.

Visual C++ makes it easy to create resources for property pages by displaying the Add Resource dialog box, expanding the Dialog node and selecting one of the IDD_PROPPAGE_X items.

After adding the resource for the property page, as done for a dialog box, you can add a class for it. Unlike the regular dialog box, a property page is based on the **CPropertyPage** class which itself is based on **CDialog**. The **CPropertyPage** class is declared in the **afxdlgs.h** header file.

Practical Learning: Creating Property Pages

1. Using either MFC AppWizard (exe) or MFC Application, start a New Project named Geometry
2. Create it as Dialog-Based with no About Box and set the Dialog Title to Quadrilateral



MFC Application Wizard - Quadrilateral

User Interface Features

Specify options that control the look and feel of your application.

Main frame styles:

☐ Thick frame
☐ Minimize box
☐ Maximize box
☐ Minimized
☐ Maximized
☐ System menu
☐ About box
☑ Initial status bar
☐ Split window

Dialog title: Quadrilateral

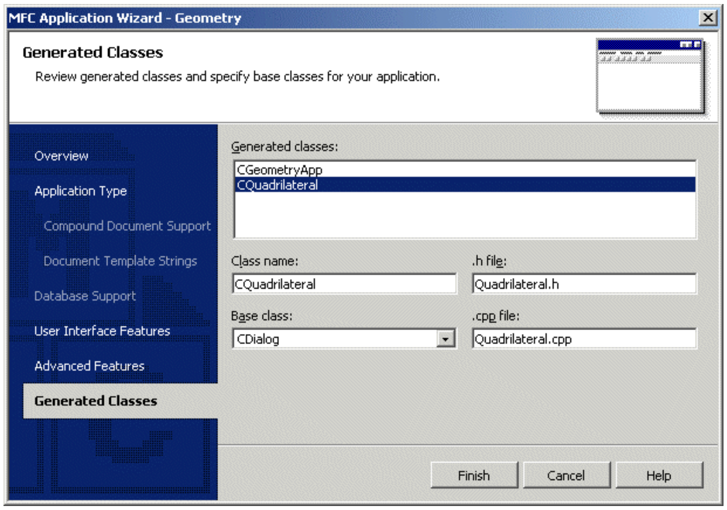
Child frame styles:

☑ Child minimize box
☑ Child maximize box
☐ Child minimized
☐ Child maximized

Toolbars:

○ None
● Standard docking
☐ Browser style

3. Change the Class Name of the dialog to **CQuadrilateral**
4. Change the name of the header of the dialog to **Quadrilateral.h** and the name of the source file to **Quadrilateral.cpp**



Generated Classes
Review generated classes and specify base classes for your application.

Generated classes:
CGeometryApp
CQuadrilateral

Class name: CQuadrilateral
.h file: Quadrilateral.h
Base class: CDialog
.cpp file: Quadrilateral.cpp

5. Click Finish

6. On the dialog, click TODO and press Delete three times to delete the TODO line, the OK and the Cancel buttons

7. While the dialog box is displaying, access its properties.
   Change its **ID** to **IDD_QUADRILATERAL**
   If necessary, change its **Caption** to **Quadrilateral**
   Change its **Style** value to **Child**
   Change its **Border** to **Thin**
   Remove the check mark of the **System Menu** check box or set it to False
   Set its **Visible** property to False or remove its check mark
   If necessary, remove the check mark of the **Disabled** check box or set it to False

8. Save All

9. From the Controls toolbox, click the Edit Box button ab| and click anywhere in the dialog box

10. Open the header file of the dialog
    Change its base class to **CPropertyPage**
    Change the value of the IDD to **IDD_QUADRILATERAL**

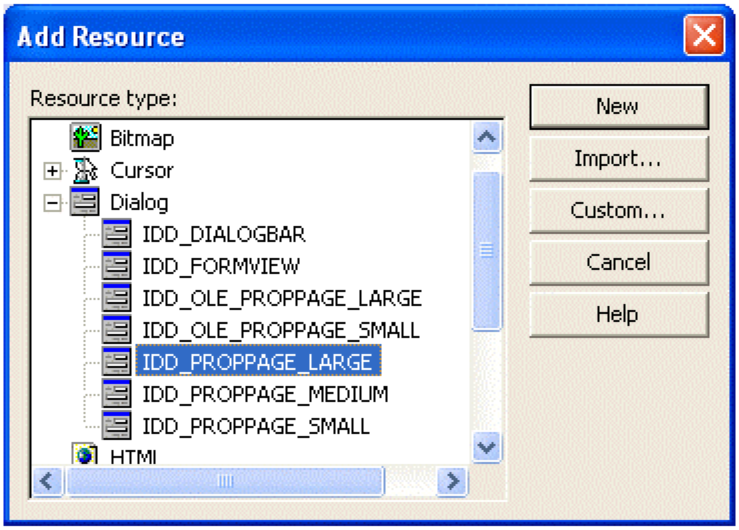
```
class CQuadrilateral : public CPropertyPage
{
// Construction
public:
	CQuadrilateral(CWnd* pParent = NULL);	// standard constructor

// Dialog Data
	//{{AFX_DATA(CQuadrilateral)
	enum { IDD = IDD_QUADRILATERAL };
		// NOTE: the ClassWizard will add data members here
	//}}AFX_DATA
```

11. Change the source file of the CQuadrilateral class as follows:

```
// Quadrilateral.cpp : implementation file
//

#include "stdafx.h"
#include "Geometry.h"
#include "Quadrilateral.h"
```

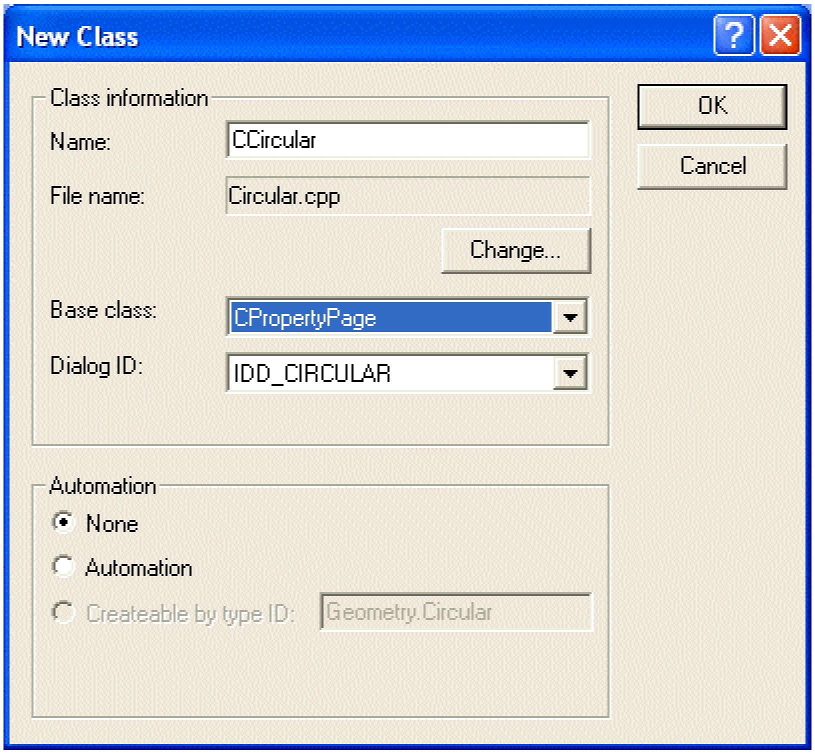
```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CQuadrilateral dialog

CQuadrilateral::CQuadrilateral(CWnd* pParent /*=NULL*/)
    : CPropertyPage(CQuadrilateral::IDD)
{
    //{{AFX_DATA_INIT(CQuadrilateral)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CQuadrilateral::DoDataExchange(CDataExchange* pDX)
{
    CPropertyPage::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CQuadrilateral)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CQuadrilateral, CPropertyPage)
    //{{AFX_MSG_MAP(CQuadrilateral)
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CQuadrilateral message handlers

BOOL CQuadrilateral::OnInitDialog()
{
    CPropertyPage::OnInitDialog();

    SetIcon(m_hIcon, TRUE);                 // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // TODO: Add extra initialization here

    return TRUE;  // return TRUE  unless you set the focus to a control
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CQuadrilateral::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting
        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CPropertyPage::OnPaint();
    }
}

HCURSOR CQuadrilateral::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}
```

12. Save All
13. On the main menu, click either Insert -> Resource… (MSVC 6) or Project -> Add Resource… (MSVC 7)
14. In the Add Resource dialog box, click the + button of Dialog to expand it
15. Click IDD_PROPPAGE_LARGE



16. Click New
17. Delete the TODO line
18. Change the **ID** of the new dialog to **IDD_CIRCULAR** and its **Caption** to **Circular**
19. On the Controls toolbox, click the Slider button and click anywhere on the dialog box
20. If you are using MSVC 6, right-click the Circular dialog box and click ClassWizard. A message box will display. Read it. Select **Create A New Class** and click OK.

Type **CCircular** and, in the **Base Class** combo box, select **CPropertyPage**



New Class

Class information

Name: CCircular

File name: Circular.cpp

Base class: CPropertyPage

Dialog ID: IDD_CIRCULAR

Automation

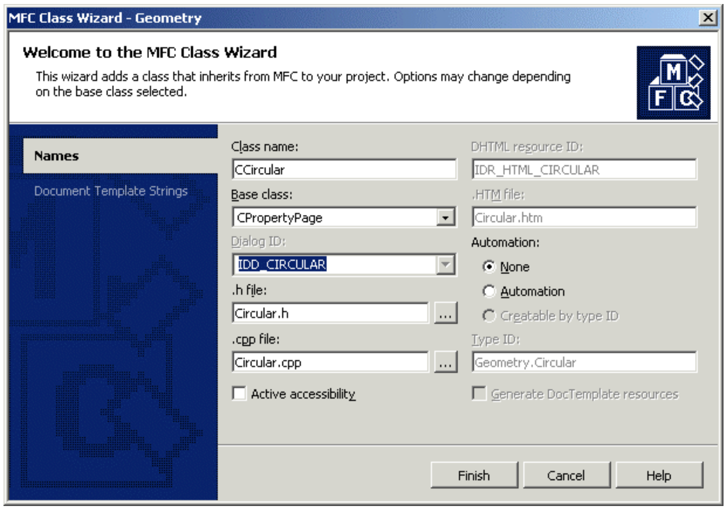
● None

○ Automation

○ Createable by type ID: Geometry.Circular

Click OK twice

If you are using MSVC 7, right-click the dialog box and click Add Class… Type the **Class Name** as **CCircular** and, in the **Base Class** combo box, select **CPropertyPage**



MFC Class Wizard - Geometry

Welcome to the MFC Class Wizard

This wizard adds a class that inherits from MFC to your project. Options may change depending on the base class selected.

Class name: CCircular

Base class: CPropertyPage

Dialog ID: IDD_CIRCULAR

.h file: Circular.h

.cpp file: Circular.cpp

□ Active accessibility

DHTML resource ID: IDR_HTML_CIRCULAR

.HTM file: Circular.htm

Automation:

● None

○ Automation

○ Creatable by type ID

Type ID: Geometry.Circular

□ Generate DocTemplate resources

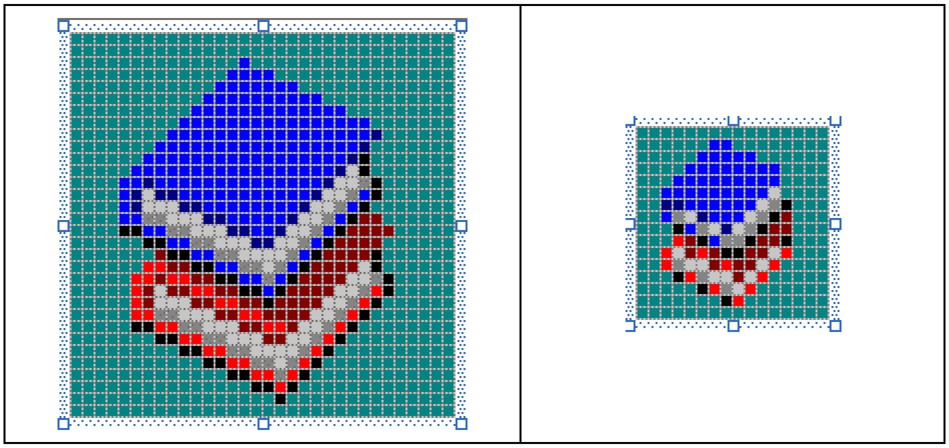
Click Finish

21. Add another IDD_PROPPAGE_LARGE property page. Delete its TODO line. Change its ID to **IDD_G3D** and its Caption to **3-Dimensions**

22. On the Controls toolbox, click the Check Box button ☒ and click anywhere on the dialog

23. Create or Add A New Class for the IDD_G3D dialog box
    Name it **CGeome3D**
    Base it on the **CPropertyPage** class

24. Change the design of the IDR_MAINFRAME icon as follows:



25. Save All

12.3.3 Property Sheets

To implement its functionality, the property pages are put together and kept as an entity by an object called a property sheet. The property sheet acts as their parent. Like other controls, we will see that the property pages must be added to a property sheet.

There is no resource to create for a property sheet. A property sheet is based on the **CPropertySheet** class, which is not based on **CDialog** but it provides the same functionality as dialog. Therefore, a property sheet is sometimes called, or referred to as, a dialog box. Instead, **CPropertySheet** is based on the **CWnd** class. Therefore, to implement your property page(s), you can simply declare a **CPropertySheet** variable and use it to display the application or you can programmatically derive a class from **CPropertySheet**. The **CPropertySheet** class is declared in the **afxdlgs.h** header file.

If you decide to directly use the **CPropertySheet** class, declare a variable for it where the application will need to be displayed or instantiated, which could be in the **CWinApp::InitInstance()** event. To display the property sheet, call its **DoModal()** method. This could be done as follows:

```
BOOL CMyApp::InitInstance()
{
        CPropertySheet MySheet;

        MySheet.DoModal();
}
```

To specify your property pages as belonging to the property page, declare a variable for each one of them. Then, call the **CPropertySheet::AddPage()** method to add each property page. The syntax of this method is:

```
void AddPage(CPropertyPage *pPage);
```

This method takes the variable of each page and adds it as part of the property sheet. Here is an example:

```
BOOL CmyApp::InitInstance()
{
	CPropertySheet MySheet;

	CFirstPage     First;
	CSecondPage    Second;

	MySheet.AddPage(&First);
	MySheet.AddPage(&Second);

	MySheet.DoModal();
}
```

If you want to have better access to the property sheet as a class, you should derive your own class from **CPropertySheet**. You will have the ability to use any or a combination of these constructors:

```
CPropertySheet();
CPropertySheet(UINT nIDCaption, CWnd *pParentWnd=NULL, UINT iSelectPage=0);
CPropertySheet(LPCTSTR pszCaption, CWnd *pParentWnd=NULL, UINT iSelectPage=0);
```

The default constructor is used in the same circumstance as the **CPropertySheet** variable declared above. Its allows you to create a **CPropertySheet** instance and change its characteristics later. Both the second and the third constructors allow you to specify a caption for the property. If you want to use the first, create a string with an identifier in a String Table and use that ID as the argument. Otherwise, when declaring a variable using the second constructor, you can directly provide a null-terminated string as argument.

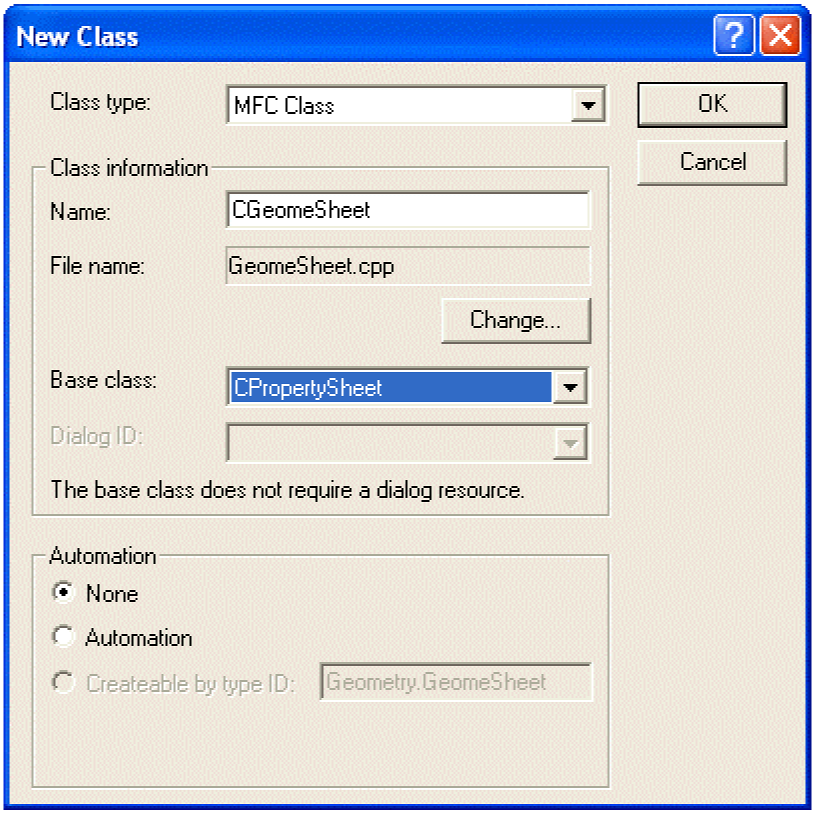
If you want to specify a title for the property sheet, you can call the **CPropertySheet::SetTitle()** method. Its syntax is:

```
void SetTitle(LPCTSTR lpszText, UINT nStyle = 0);
```

The first argument is a null terminated string that will be the new title. If you want the caption to display the string starting with “Properties for”, pass a second argument as **PSH_PROPTITLE**.

Practical Learning: Creating a Property Sheet

1. In the Class View, right-click Geometry and click either New Class or Add -> Add Class…
2. If you are using MSVC 6, set the Class Type to MFC Class
   If you are using MSVC 7, click MFC Class and click Open
   Set the name of the class to **CGeomeSheet**
3. In the Base Class combo box, select **CPropertySheet**



Class type: MFC Class

Class information

Name: CGeomeSheet

File name: GeomeSheet.cpp

Base class: CPropertySheet

Dialog ID:

The base class does not require a dialog resource.

Automation

● None

○ Automation

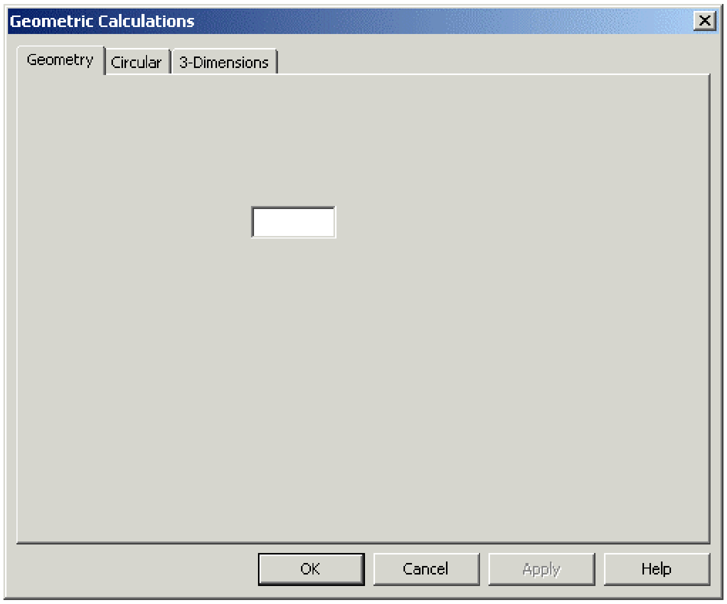
○ Createable by type ID: Geometry.GeomeSheet

4. Click OK or Finish
5. Change the source code of the CGeometryApp class as follows:

```
// Geometry.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "Geometry.h"
#include "Quadrilateral.h"
#include "GeomeSheet.h"
#include "Circular.h"
#include "Geome3D.h"

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

/////////////////////////////////////////////////////////////////////////////
// CGeometryApp

BEGIN_MESSAGE_MAP(CGeometryApp, CWinApp)
	//{{AFX_MSG_MAP(CGeometryApp)
	//}}AFX_MSG
	ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CGeometryApp construction
```

```
CGeometryApp::CGeometryApp()
{
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CGeometryApp object

CGeometryApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CGeometryApp initialization

BOOL CGeometryApp::InitInstance()
{
	AfxEnableControlContainer();

	// Standard initialization

#ifdef _AFXDLL
	Enable3dControls();			// Call this when using MFC in a 
shared DLL
#else
	Enable3dControlsStatic();	// Call this when linking to MFC statically
#endif

	CGeomeSheet GeoSheet("Geometric Calculations");

	CQuadrilateral	Quad;
	CCircular		Circ;
	CGeome3D		G3D;

	GeoSheet.AddPage(&Quad);
	GeoSheet.AddPage(&Circ);
	GeoSheet.AddPage(&G3D);

	m_pMainWnd = &GeoSheet;
	int nResponse = GeoSheet.DoModal();

	if (nResponse == IDOK)
	{
	}
	else if (nResponse == IDCANCEL)
	{
	}

	// Since the dialog has been closed, return FALSE so that we exit the
	//  application, rather than start the application's message pump.
	return FALSE;
}
```

6. Test the application



7. Return to MSVC and close the project (MSVC 6: File -> Close Workspace; MSVC 7: File -> Close Solution)

12.3.4 Wizard Pages

A wizard, like a property sheet, is a series of dialog boxes that are considered as one entity, tremendously saving the available space. When put together, the dialog boxes are referred to as wizard pages. Like the property pages, the wizard pages can help the programmer add more Windows controls than a single dialog box with the same dimension would allocate. While the property pages are positioned one in front of the others in a z-axis, the wizard pages are positioned so that, when one displays, the others are completely hidden. While a property page can be accessed by the user clicking its tab to bring it to the front and send the others to the back, a wizard is equipped with buttons such as Back or Next.

A wizard is created using the exact same approach as the property sheet. Each involved dialog box is created with the same properties:

?? **Style: Child**

?? **Border: Thin**

?? **Title Bar**: True

?? **System Menu**: False

?? **Visible**: False

?? **Disabled**: True

Each page is based on the **CPropertyPage** class. To display the wizard, use the **CPropertySheet** class in the exact same way as seen for the property pages above. The only difference is that, to make this a wizard, you must call the **CPropertySheet::SetWizardMode()** method before calling **DoModal().** The syntax of the **SetWizardMode()** member function is:

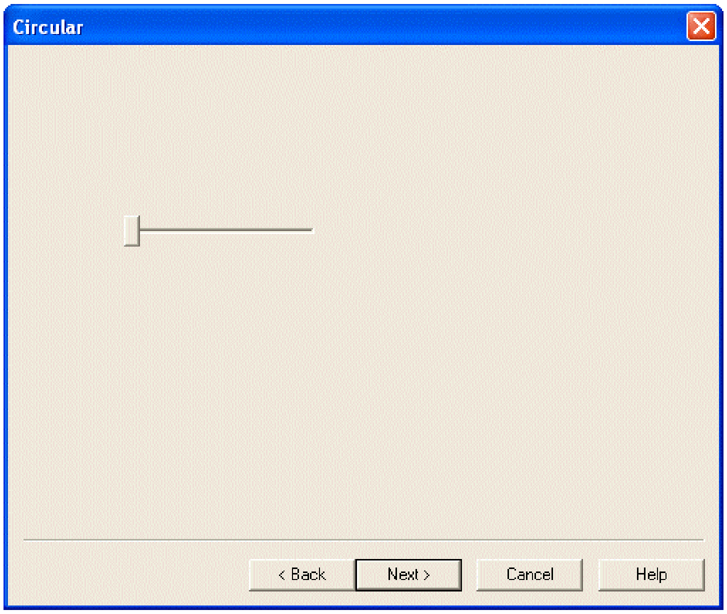
```
void SetWizardMode();
```

Practical Learning: Creating a Wizard

1. Open Windows Explorer or My Computer and display the contents of the parent folder that holds the previous exercise
2. In the right frame, right-click **Geometry** and click **Copy**
3. Right-click an unoccupied area in the same right frame and click Paste
4. Rename Copy of Geometry to **WizardPages**
5. Open the **WizardPages** project or solution
6. Access the CGeometry::InitInstance() event and change it as follows:

```
BOOL CGeometryApp::InitInstance()
{
	AfxEnableControlContainer();

	// Standard initialization

#ifdef _AFXDLL
	Enable3dControls();			// Call this when using MFC in a shared DLL
#else
	Enable3dControlsStatic();	// Call this when linking to MFC statically
#endif

	CGeomeSheet GeoSheet("Geometric Calculations");

	CQuadrilateral	Quad;
	CCircular		Circ;
	CGeome3D		G3D;

	GeoSheet.AddPage(&Quad);
	GeoSheet.AddPage(&Circ);
	GeoSheet.AddPage(&G3D);

	m_pMainWnd = &GeoSheet;

	GeoSheet.SetWizardMode();

	int nResponse = GeoSheet.DoModal();
	if (nResponse == IDOK)
	{
	}
	else if (nResponse == IDCANCEL)
	{
	}

	// Since the dialog has been closed, return FALSE so that we exit the
	//  application, rather than start the application's message pump.
	return FALSE;
}
```

7. Test the application



8. Return to MSVC

Chapter 13: Control Design

✍ Forms

✍ Dialog Box’ Messages and Events

✍ Floating Windows

☞ Control Design

13.1 Forms

13.1.1 Introduction

Like a dialog box, a form is the primary object used to host Windows controls to allow user interaction with the computer. Like a dialog box, a form does not mean anything except for the controls it is hosting. This means that a form by itself does not present any significant functionality but it can be equipped with characteristics that make it a valuable alternative to other view types.

When using a form, the user typically interacts with the controls it is hosting. These controls can be positioned anywhere on the form. If there are too many controls or the controls are positioned so the occupied area is larger than the form can display at one time, the form would be equipped with scroll bars.

A form is created using the **CFormView** class which is derived indirectly from **CView** through **CScrollView**.

13.1.2 Form Creation

There are three main ways you can provide a form to an application.

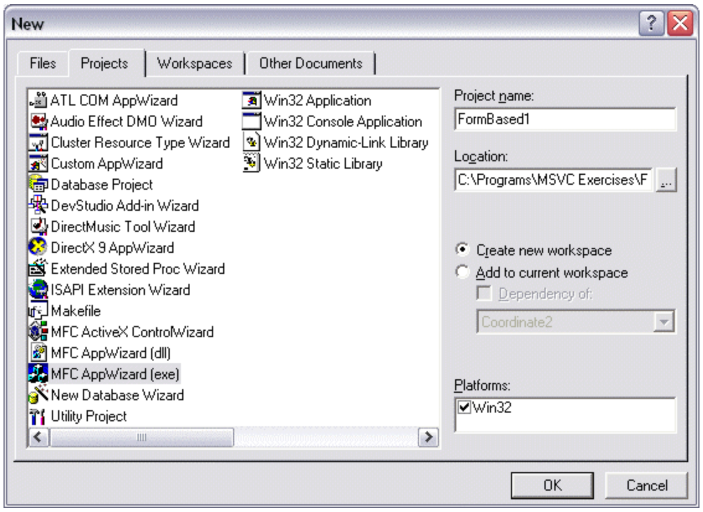
The easiest way to create a form is by using the AppWizard. To do this, when creating the application, set the Base Class to **CFormView**. A document/view application would be created with its view based on the **CFormView** class. The wizard would have created a Child window with no borders and no title bar (over all, you should not attempt to change the properties of this window). Also, if you create an application that supports databases, AppWizard would create a form that can be used on that database.

If you have already created an application, for example based on **CView**, you can either add a form view to the application, change the application's base view class into form view, or replace its view class with a form view. If you add a form view to the application, the user would be prompted to select one of both documents when the applications comes up.

The last option you have, which you should hardly do is to work from scratch in creating a form view-based application. The reason you should not spend time doing this is the amount of work involved that would not necessarily improve your application.

Practical Learning: Creating a Form-Based Application

1. Display the New or the New Project dialog box and specify that you want to use MFC AppWizard or MFC Application
2. Set the Project Name to FormBased1


New

Project name: FormBased1

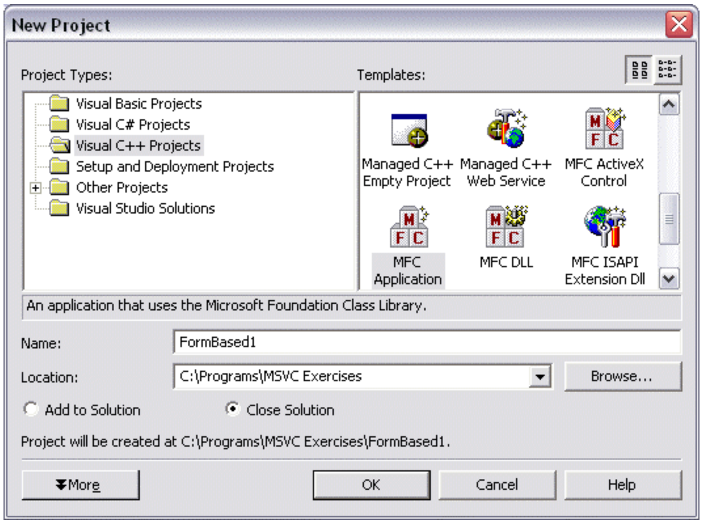
Location: C:\Programs\MSVC Exercises\F

● Create new workspace

○ Add to current workspace

□ Dependency of: Coordinate2

Platforms: ☑ Win32


New Project

An application that uses the Microsoft Foundation Class Library.

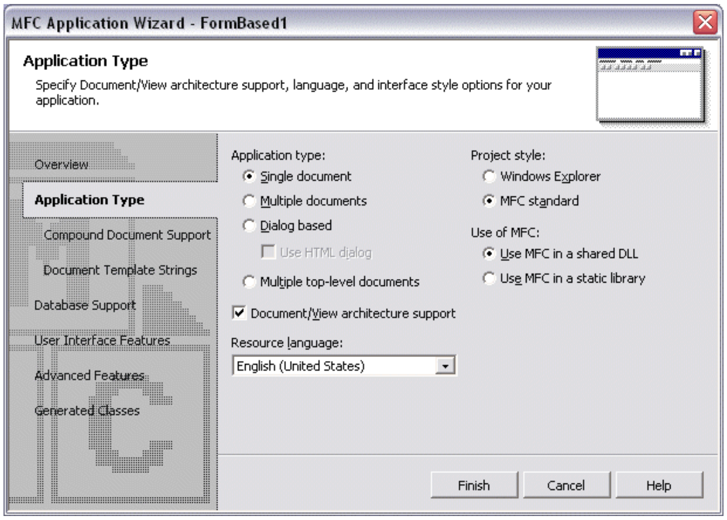
Name: FormBased1

Location: C:\Programs\MSVC Exercises

○ Add to Solution ● Close Solution

Project will be created at C:\Programs\MSVC Exercises\FormBased1.

3. Click OK
4. Specify that you want to create a Single Document type of application


MFC Application Wizard - FormBased1

Application Type

Specify Document/View architecture support, language, and interface style options for your application.

Application type:

● Single document

○ Multiple documents

○ Dialog based

□ Use HTML dialog

○ Multiple top-level documents

☑ Document/View architecture support

Resource language: English (United States)

Project style:

○ Windows Explorer

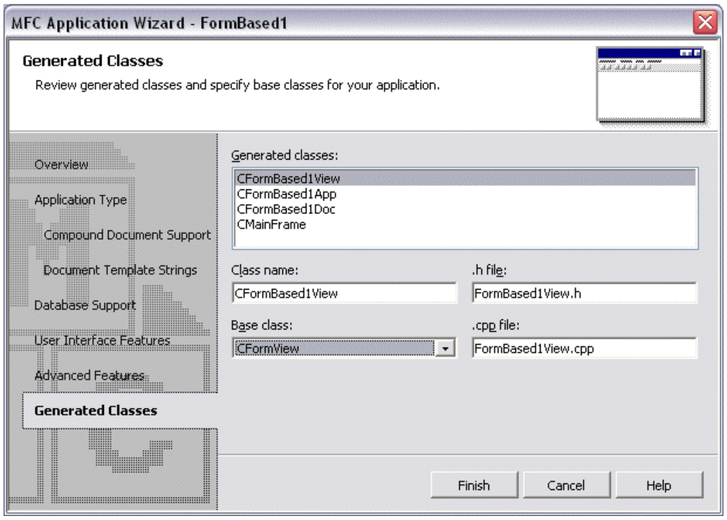
● MFC standard

Use of MFC:

● Use MFC in a shared DLL

○ Use MFC in a static library

5. In the last step, in the Base Class combo box, select CformView


MFC Application Wizard - FormBased1

Generated Classes

Review generated classes and specify base classes for your application.

Generated classes:

CFormBased1View
CFormBased1App
CFormBased1Doc
CMainFrame

Class name: CFormBased1View

.h file: FormBased1View.h

Base class: CFormView

.cpp file: FormBased1View.cpp

6. Click Finish
7. To display the form, in the Resource View, expand the Dialog folder and double-click IDD_FORMBASED1_FORM
8. In the Properties window, notice that its Style is set to Child and its Border is set to None
9. Press Ctrl + F5 to test the application
10. After viewing it, return to MSVC
11. Controls Designs on Forms and Dialog Boxes

13.2 Dialog Box Messages and Events

In this section, unless specified otherwise, the expressions “dialog box” or “dialog-based object” refer to the regular dialog box, the property sheet, the property page, or the flat dialog-window that composes a form.

13.2.1 The Dialog Box as a Control Initializer

As the most regularly used host of Windows Controls, a dialog box most be able to handle messages that not only pertain to its own existence but also to the availability, visibility, and behaviors of the controls it is holding. For example, if you create a control and position it on the dialog box, sometimes you will need to initialize, give it primary values. Although the dialog box provides a constructor and even inherits the **WM_CREATE** message along with the **PreCreateWindow()** event from its parent the **CWnd** class, if you need to initialize a control, the CDialog box and its cousins CPropertySheet, CPropertyPage, and the form provide a special event you can use to do this.

WM_INITDIALOG: The **WM_INITDIALOG** message fires the **OnInitDialog()** event after a dialog, a property sheet, a property page, or a form has been created but before this window is displayed. It is the equivalent of the **PreCreateWindow()** event used by other frame and view-based window. This event should be your favorite place to initialize a control that is hosted by the dialog-based object. Do not initialize controls in your dialog's constructor. Use the constructor only to allocate memory space for a dynamic control using the **new** operator.

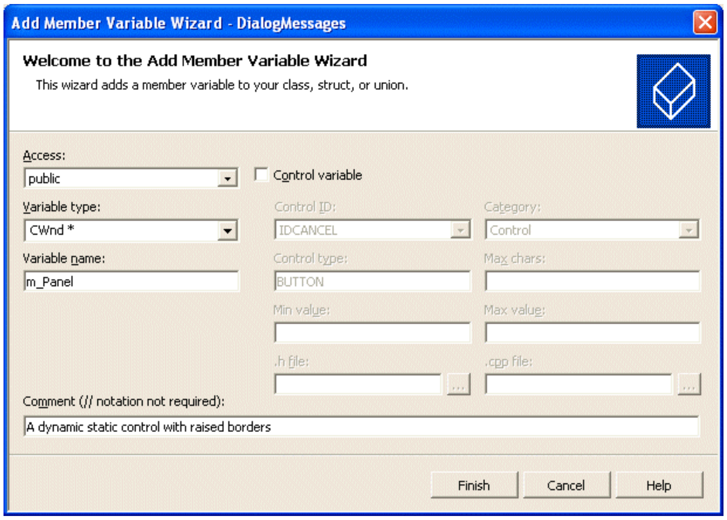
If you create a dialog-based or a form-based application, AppWizard automatically generates the **WM_INITDIALOG** message for you. If you add a dialog box, a form, a proeprty sheet, or a property page to an existing application, this message may not be added automatically. You would have to fire it. In reality, the **WM_INITDIALOG** message is inherited from the CDialog class. Therefore, you would only be overriding it.

Practical Learning: Generating Dialog Messages

1. Start a new MFC Application and name it **DialogMessages**
2. Create it as a **Single Document** and click Finish
3. To add a new object, display the Add Resource dialog box and double-click Dialog
4. Change its ID to **IDD_DYNAMIC_DLG** and its **Caption** to **Dynamic Objects**
5. Add a class for it. Name it **CDynamicDlg** and base it on the **CDialog** class. Click Finish
6. To make sure you can display the dialog box, display the IDR_MAINFRAME menu. Under View, add a separator. Then add a menu item with the caption as **&Dynamic…** and press Enter
8. Right-click the Dynamic menu item and click **Add Event Handler…**
9. Accept the Message type as COMMAND. Accept the suggested Function Handler Name. In the Class List, click CMainFrame then click Add and Edit
10. Implement the event as follows:

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "DialogMessages.h"

#include "MainFrm.h"
#include "DynamicDlg.h"
. . .

void CMainFrame::OnViewDynamic()
{
	// TODO: Add your command handler code here
	CDynamicDlg Dlg;

	Dlg.DoModal();
}
```

11. Test the application and return to MSVC
12. Click the Class View tab and, if necessary, expand the DynamicMessages node. Right-click CDynamicDlg and click Add Variable…
13. In the Variable Type combo box, type **CWnd *** and, in the Variable Name edit box, type **m_Panel**



Add Member Variable Wizard - DialogMessages

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: public

☐ Control variable

Variable type: CWnd *

Variable name: m_Panel

Control ID: IDCANCEL

Category: Control

Control type: BUTTON

Max chars:

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required): A dynamic static control with raised borders

14. Click Finish

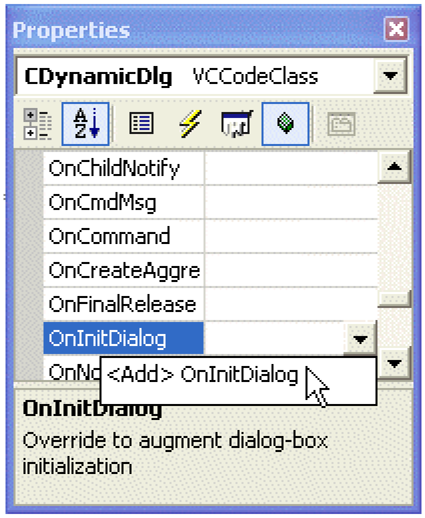
15. In the constructor of the dialog, use the new operator to allocate memory for the dynamic control. Then, in the destructor, destroy the control using the delete operator:

```
CDynamicDlg::CDynamicDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CDynamicDlg::IDD, pParent)
        , m_Panel(NULL)
{
        m_Panel = new CWnd;
}

CDynamicDlg::~CDynamicDlg()
{
        delete m_Panel;
}
```

7. To fire the **OnInitDialog** event for the dialog, in the combo box of the Properties window, CDynamicDlg should be displaying. If not, in Class View, click CDynamicDlg

   In the Properties window, click the Overrides button

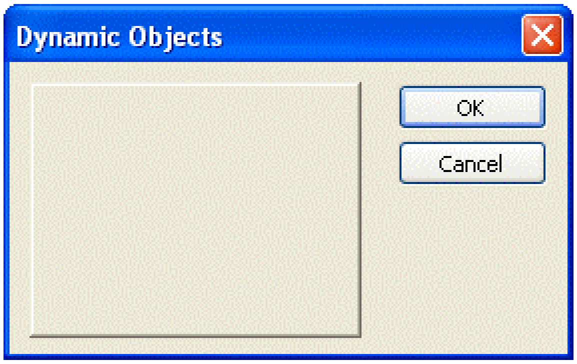
8. Scroll down in the Proeprties window and click OnInitDialog. Click its arrow and click <Add> OnInitDialog



9. Implement the event as follows:

```
BOOL CDynamicDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO:  Add extra initialization here
	CRect Recto(10, 10, 176, 138);

	m_Panel->Create("STATIC", NULL,
	                 WS_CHILD | WS_VISIBLE | WS_DLGFRAME,
	                Recto, this, 0x166, NULL);

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

10. Execute the application and display the Dynamic Objects dialog box using the main menu:



11. To close the dialog box, click its system Close button ☒ or ☒. Also, close the application and return to MSVC

12. Open the Geometry application. If you do not have it, open the Geometry1 application from the exercises that accompany this book

13. As done for the above CDynamicDlg class, Add the OnInitDialog() event for the CGeomeSheet, the CCircular, and the CGeome3D classes

14. Save All

13.2.2 Other Dialog-Based Windows Messages

WM_PAINT: After creating the dialog box, if you want to paint it or draw something on it, such as changing its background color, use the **WM_PAINT** message. If you create a dialog or form-based application, this message is automatically generated by the wizard. If you add a dialog-based object to your application and need the **OnPaint()** event, you would have to add it yourself

WM_CLOSE: As mentioned already, a regular dialog box, including a property sheet or a wizard, is equipped with the system Close button ☒ on its title bar. This allows the user to close it any time. As a programmer, one of your jobs is to control your application, be able to predict "bad moves" from the user. From example, imagine you create a dialog-based object without a Cancel button but with OK (like the most basic message box). If the user clicks the system Close button ☒, you may not know what the user did. The WM_CLOSE message fires the **OnClose()** event when the user clicks the system Close button ☒. It is important to understand that **WM_CLOSE** is a message and not a method, meaning it sends a message, it does not take an action. This implies that the **OnClose()** event fires when the user makes an attempt to close the dialog but before the dialog is actually closed.

You can use the **OnClose()** event to find out what has been done on the dialog prior to the user's attempt to closing it. You can also use it to deny closing the dialog, to warn the user about something, or to do anything that you judge necessary. If you perform some processing, such as validating some values, when the user attempts to close the dialog box, if you still want to close the dialog, call the parent event handler with **CDialog::OnClose()**. In fact, this line of code is added to the event if you generate it using the wizard. If you want to conditionally close the dialog, you can write a conditional statement that can check whether something is written in order to close it. If you do not want to close the dialog box, do not call the parent **CDialog::OnClose()** event.

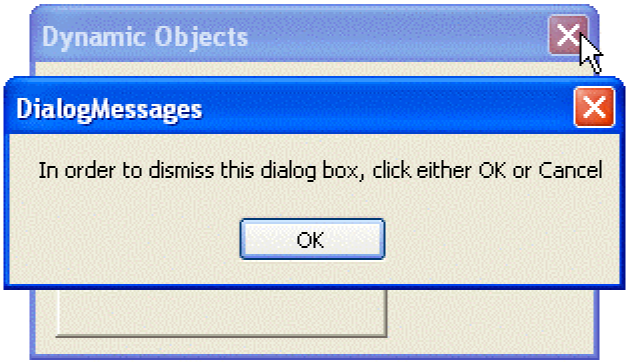
WM_DESTROY: Once a dialog, in fact any (CWnd) window object, has been closed, it must be destroy so the memory it was using can be reclaimed. If you want to do something before the object is destroyed, use the **WM_DESTROY** to fire the **OnDestroy()** event.

Practical Learning: Firing Windows Events

1. Open the **DialogMessages** application created above
   In the Class View tab, click CDynamicDlg to display it in the combo box of the Properties window.
   To access the **WM_CLOSE**messages, in the Properties window, click the Messages button
2. Click the **WM_CLOSE** item to display its combo box. Click the arrow of the combo box and click <Add> OnClose
3. To prevent the user from closing the dialog box using the system Close button, implement the event as follows:

```
void CDynamicDlg::OnClose()
{
    // TODO: Add your message handler code here and/or call default
    MessageBox("In order to dismiss this dialog box, "
               "click either OK or Cancel");

//  CDialog::OnClose();
}
```

16. Execute the application. Display the Dynamic Objects dialog and click its system Close button to close



In order to dismiss this dialog box, click either OK or Cancel

4. Close the dialog box using either the OK or the Cancel buttons. Also, close the application and return to MSVC

5. Open the Geometry application. If you do not have it, open the Geometry1 application that accompanies this book

6. As done for the CDynamicDlg class above, add a WM_PAINT message for the CQuadrilateral, the CCircular, and the CGeome3D classes

7. Save all

13.2.3 Control-Related Messages

WM_CTLCOLOR: Once a control has been created, it can be displayed on the screen. At any time, whether before the control is displayed or after it has been displayed, if you need to perform some type of drawing such as changing its color, you can use the **WM_CTLCOLOR** message which fires the **OnCtlColor()** event.

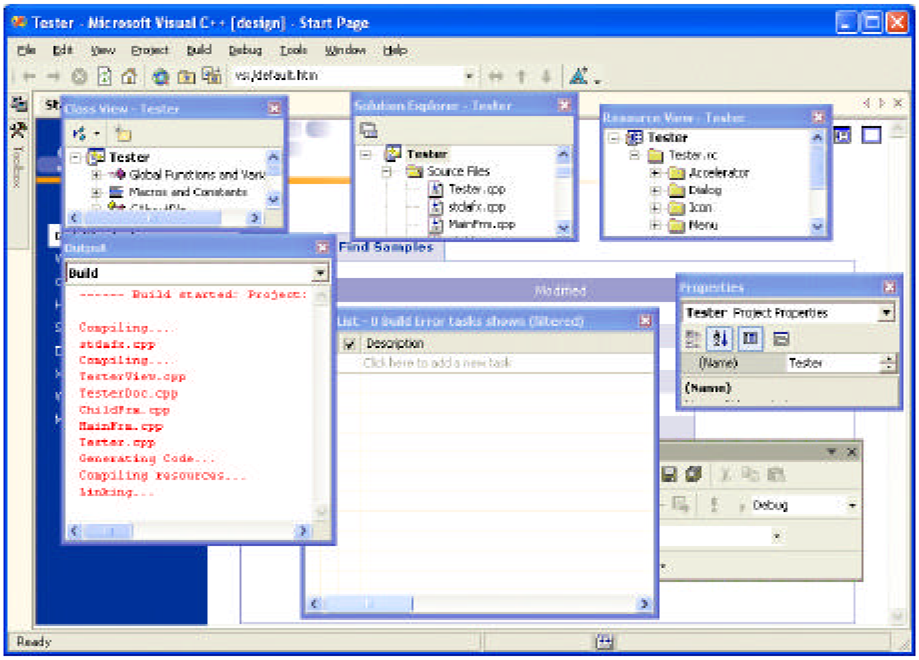
Scroll Bar Messages: As we will learn when studying scroll bars and thumb-based controls, a dialog-based object can be equipped with scroll bars, just like a frame of a view-based application. We will also see that these controls do not handle their own scrolling messages. Ther rely on their host.

If a dialog object is equipped with one or two scroll bars or one of its controls is based on scrolling operations, when the user clicks a scroll bar, the **OnHScroll()** or the **OnVScroll()** event is sent based on the **WM_HSCROLL** or the **WM_VSCROLL** messages respectively.

13.3 Floating Windows

13.3.1 Introduction

A floating window is an object that is part of an application and behave as a semi-dialog box. Such a window does not constitute an application by itself. It is used to complete the application it is part of. A floating window behaves like a dialog box with the main difference that the user does not have to close the floating window in order to access the rest of the application, which is the case for a modal dialog box. Here is an application that uses many floating windows:



```
------ Build started: Project:

Compiling...
stdafx.cpp
Compiling...
TesterView.cpp
TesterDoc.cpp
ChildFrm.cpp
MainFrm.cpp
Tester.cpp
Generating Code...
Compiling resources...
Linking...
```

There are various types of floating windows used on different applications. They include modeless dialog boxes, tool bars, and dialog bars.

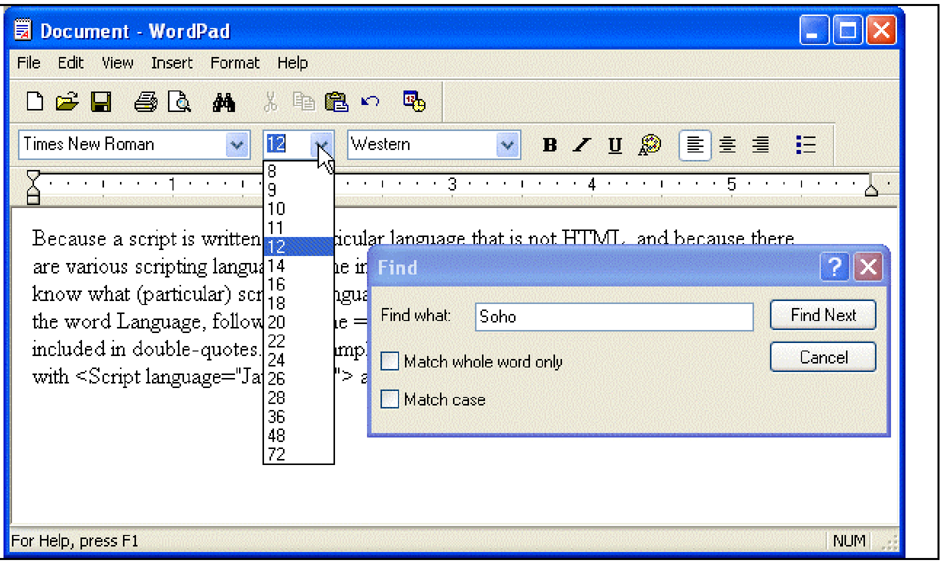
Practical Learning: Introducing Floating Windows

1. Start a new project named **Floating**
2. Create it as a Single Document and click Finish

13.3.2 The Modeless Dialog Box

A modeless dialog box allows the user to access the main application or any other possible object of the application even if this dialog box is displaying. The user can decide to close it when the object is not needed anymore or keep it on as long as necessary. Here is an example:

The Find dialog box of WordPad is an example of a modeless dialog box. Although it is displaying on this picture, the user can still click and activate any of the windows in the background



Find what: Soho

☐ Match whole word only

☐ Match case

Because a script is written icular language that is not HTML, and because there
are various scripting langua e in
know what (particular) scr ngua
the word Language, follow e =
included in double-quotes. mp.
with <Script language="Ja "> a

Because a modeless dialog box is closed when the user judges it necessary, making it possible to forget, the operating system needs to make sure that this object closes when its parent application closes so its memory resource can be freed and made available to other applications. Based on this, the creation of a modeless dialog is a little different than that of the modal dialog box.

There are two main techniques you can use to create a modeless dialog box. You can first declare a **DLGTEMPLATE** variable and “fill it up”. This structure is defined as follows:

```
typedef struct {
  DWORD style;
  DWORD dwExtendedStyle;
  WORD  cdit;
  short x;
  short y;
  short cx;
  short cy;
} DLGTEMPLATE, *LPDLGTEMPLATE;
```

After building the template from its variable, you can call the **CDialog::CreateIndirect()** method to create the modeless object. This method comes in two versions as follows:

```
BOOL CreateIndirect(LPCDLGTEMPLATE lpDialogTemplate, CWnd* pParentWnd = NULL);
BOOL CreateIndirect(HGLOBAL hDialogTemplate, CWnd* pParentWnd = NULL);
```

Because of the amount of work involved with using this type of dialog box, we will ignore it.

Another technique you can use consists of first creating a dialog resource and associating a class to it. To formally create a modeless dialog box and make it part of the application, you can call the **Create()** method of the **CDialog** class. It is provided in two syntaxes that are:

```
BOOL Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);
BOOL Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL);
```

As done with the **CDialog** constructor, the *nIDTemplate* parameter is the identifier you used when visually creating the dialog resource. If you are using a dialog box from a template, specify its name as the *lpszTemplateName* argument. In order to effectively use a modeless dialog box, it must be called from another window. Otherwise you can just create a regular dialog box. The object from which you will be calling the dialog box is also responsible for closing it (or "cleaning" it). Therefore, the class that calls the modeless dialog box "owns" it. This means that you can specify it as the *pParentWnd* argument.

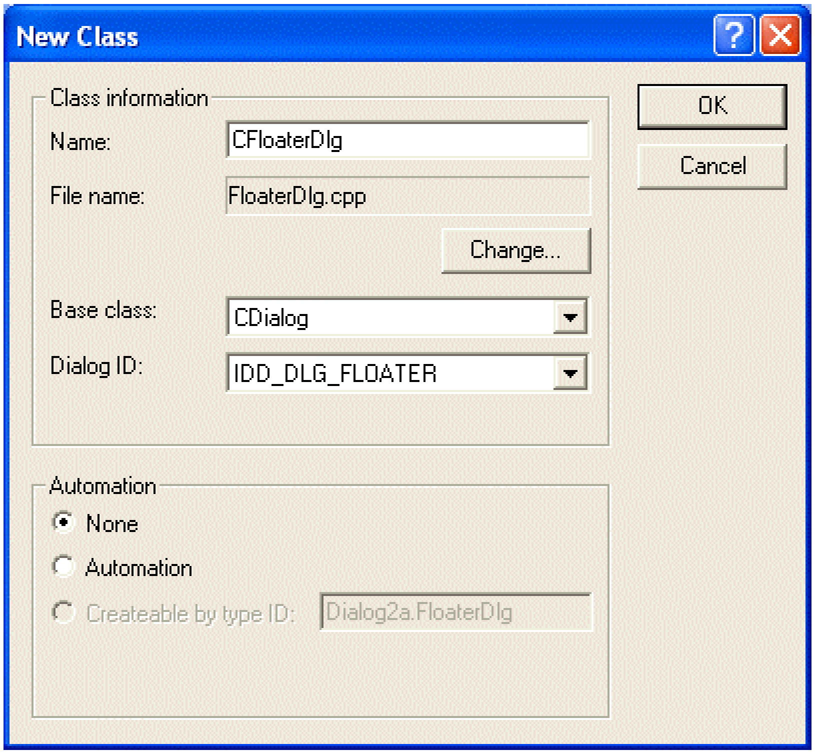
When creating a modeless dialog box, declare a pointer to its class. Then call the **Create()** method, passing the identifier or the template being used. You can do this in the constructor of the class that will be its parent. Here is an example:

```
CDialog5Dlg::CDialog5Dlg(CWnd* pParent /*=NULL*/)
	: CDialog(CDialog5Dlg::IDD, pParent)
{
	CCoolMode* Cool = new CCoolMode();

	Cool->Create(IDD_DLG_MDLS,this);
}
```

To formally display the object, you have various options. When creating it, at design time, you can set its **Visible** property to True (actually you would be applying the **WS_VISIBLE** style). If you did not make it visible, or if the modeless dialog box is hidden at a particular time, you can programmatically display it.

Practical Learning: Using Dialog Box Methods

1. To add a new dialog box, display the Add Resource dialog box and double-click Dialog
2. Click the OK button and press Delete twice to dismiss both the OK and the Cancel buttons
3. Using the Properties window, change the ID of the dialog box to **IDD_DLG_FLOATER** and its Caption to **Floater**
4. Check the **Visible** check box or set it to True
5. Check the **Tool Window** check box or set it to True
6. Set the **Border** property to **Resizing**
7. Add a class for the dialog box. Call it CFloaterDlg and base it on the CDialog class



Class information
Name: CFloaterDlg
File name: FloaterDlg.cpp
Base class: CDialog
Dialog ID: IDD_DLG_FLOATER

Automation
● None
○ Automation
○ Createable by type ID: Dialog2a.FloaterDlg

8. Click OK
9. To prevent the floating window from being closed, in the Class View, click CFloaterDlg to select it. In the Properties window, click the Messages button
10. Click the arrow of the **WM_CLOSE** combo box and select the only item in the list
11. When the user tries to close the window, we will hide it instead since the window will be closed when the parent application closes. Therefore, implement the event as follows:

```
void CFloaterDlg::OnClose()
{
	// TODO: Add your message handler code here and/or call default
	ShowWindow(FALSE);

	CDialog::OnClose();
}
```

12. Display the MainFrm.h header file and declare a CFloaterDlg variable named Floater

```
//
#pragma once
#include "FloaterDlg.h"

class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
	CMainFrame();
	DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:
```

```
// Operations
public:
	CFloaterDlg *Floater;
```

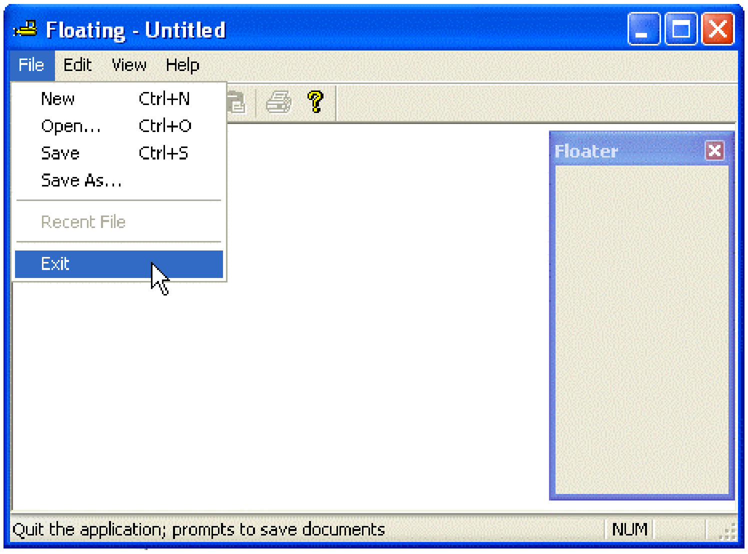
13. Display the MainFrm.cpp file and initialize the Floater variable using the new operator. Then, use the **OnCreate()** event to create the floating window:

```
CMainFrame::CMainFrame()
{
	// TODO: add member initialization code here
	Floater = new CFloaterDlg;
}

. . .

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	. . .

	Floater->Create(IDD_DLG_FLOATER, this);

	return 0;
}
```

14. To give the user the ability to display or hide the Floater window, display the IDR_MAINFRAME menu and click View
15. Add a separator under the Status Bar menu item. In the box under the new separator, add a menu item with a caption of **&Floater…**and press Enter
16. Right-click the Floater menu item and click **Add Event Handler…**
17. Accept the Message type as COMMAND. Accept the suggested Function Handler Name. In the Class List, click CMainFrame then click Add and Edit
18. Change the MainFrm.cpp file as follows:

```
void CMainFrame::OnViewFloater()
{
	// TODO: Add your command handler code here
	// Display the window, even if it is already displaying
	Floater->ShowWindow(SW_SHOW);
}
```

19. Test the application



20. Return to MSVC

13.3.3 Modeless Property Sheets

Like a regular dialog box, a property sheet can be created as a floating window. A modeless property sheet does not display the OK, the Cancel or the Apply button. It provides property pages that can be used to better manage available space.

A modeless property sheet is created using the same approach as a modal property sheet. You can start by creating dialog boxes to be used as property pages. The dialog boxes must have the same characteristics as those of a modal property sheet and you should create a class for each one of them; a class based on **CPropertyPage**. After creating the pages and their classes, you will need a property sheet class. You can directly use the **CPropertySheet** class or derive your own class from it. To display the property sheet as modeless, instead of calling the **DoModal()** method, use the **CPropertySheet::Create()** member function. Its syntax is:

```
BOOL Create(CWnd* pParentWnd = NULL,
            DWORD dwStyle = (DWORD)–1, DWORD dwExStyle = 0);
```

The *pParentWnd* argument is the object that owns the property sheet. This is usually the class that calls or displays the property sheet. If you pass this argument with a NULL value, then the desktop owns the property sheet.

The *dwStyle* argument allows you to specify the style used to display the property sheet. You can use the same window styles we reviewed for the dialog box. You do not have to provide a value for this argument. In that case, you can pass it as –1. This would provide a title bar with the system Close button, a frame, and a border to the property sheet by combining the **WS_CAPTION**, the **WS_SYSMENU**, the **DS_MODALFRAME**, the **WS_POPUP**, the **WS_VISIBLE**, and the **DS_CONTEXT_HELP** styles

The *dwExStyle* argument allows you to specify the extended styles to use on the property sheet. If you do not pass this argument or pass it as 0, the property sheet would have a border based on the default **WS_EX_DLGMODALFRAME** style.

Practical Learning: Creating a Floating Property Sheet

1. Display the Add Resource dialog box. Expand the Dialog node and double-click IDD_PROPPAGE_SMALL. Delete its TODO line. Change its **ID** to **IDD_SMALL_PAGE** and its **Caption** to **Small**. Add a class for the dialog box. **Name** it **CSmallPge** and base it on **CPropertyPage**

2. In the same way, add another IDD_PROPPAGE_SMALL dialog box. Delete its TODO line. Change its ID to IDD_MEDIUM_PAGE and its Caption to Medium. Add a class for the dialog box. **Name** it **CMediumPge** and base it on **CPropertyPage**

3. Add a class named **CFloatingSheet** and based on **CPropertySheet**

17. Change the header file as follows:

```
#pragma once

#include "SmallPage.h"
#include "MediumPage.h"

// CFloatingSheet

class CFloatingSheet : public CPropertySheet
{
    DECLARE_DYNAMIC(CFloatingSheet)

public:
//  CFloatingSheet(UINT nIDCaption, CWnd* pParentWnd = NULL, UINT iSelectPage
= 0);
    CFloatingSheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
    virtual ~CFloatingSheet();

protected:
    DECLARE_MESSAGE_MAP()
public:
    afx_msg void OnClose();
    CSmallPage      pgSmall;
    CMediumPage pgMedium;
};
```

4. In its source file, add each page as follows:

```
// FloatingSheet.cpp : implementation file
//

#include "stdafx.h"
#include "Floating.h"
#include "FloatingSheet.h"


// CFloatingSheet

IMPLEMENT_DYNAMIC(CFloatingSheet, CPropertySheet)
/*
CFloatingSheet::CFloatingSheet(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
}
```

```
*/
CFloatingSheet::CFloatingSheet(LPCTSTR pszCaption, CWnd* pParentWnd, UINT
iSelectPage)
	:CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
	AddPage(&pgSmall);
	AddPage(&pgMedium);
}

CFloatingSheet::~CFloatingSheet()
{
}


BEGIN_MESSAGE_MAP(CFloatingSheet, CPropertySheet)
	ON_WM_CLOSE()
END_MESSAGE_MAP()


// CFloatingSheet message handlers


void CFloatingSheet::OnClose()
{
	// TODO: Add your message handler code here and/or call default
	ShowWindow(FALSE);

	// CPropertySheet::OnClose();
}
```

5. Display the IDR_MAINFRAME menu and, under the Floater menu item of the View menu, add a new item with a caption of **Fl&oating** and press Enter
6. Right-click the Floating menu item and click **Add Event Handler…**
7. Accept the Message type as COMMAND. Accept the suggested Function Handler Name. In the Class List, click CMainFrame then click Add and Edit
8. Access the MainFrm.h header file. Then, declare a pointer to the property sheet class and name it Fsheet:

```
// MainFrm.h : interface of the CMainFrame class
//
#pragma once
#include "FloaterDlg.h"
#include "FoatingSheet.h"

class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
	CMainFrame();
	DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:
	CFloaterDlg *Floater;
	CFloatingSheet *FSheet;
```

9. In the constructor of the CMainFrame class, initialize the CFloatingSheet pointer using the new operator:

```
CMainFrame::CMainFrame()
{
	// TODO: add member initialization code here
	Floater = new CFloaterDlg;
	FSheet  = new CFloatingSheet("Properties");
}
```

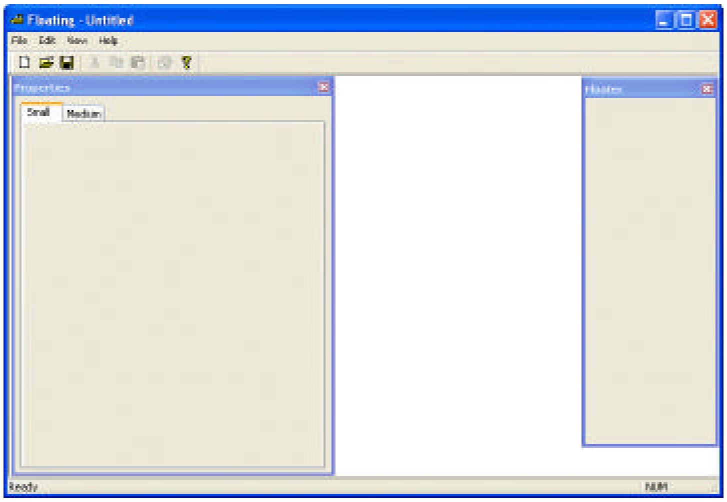
10. On the **CMainFrame::OnCreate()** event, create and and display the property sheet with the followsing:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	. . .

	Floater->Create(IDD_DLG_FLOATER, this);
	FSheet->Create(this,
				WS_CHILD | WS_POPUP | WS_CAPTION |
				WS_SYSMENU | WS_VISIBLE,
				WS_EX_TOOLWINDOW);

	return 0;
}
```

11. Implement the OnViewFloating() event as follows:

```
void CMainFrame::OnViewFloating()
{
	// TODO: Add your command handler code here
	// Display the property sheet, even if it is already displaying
	FSheet->ShowWindow(SW_SHOW);
}
```

12. Test the application



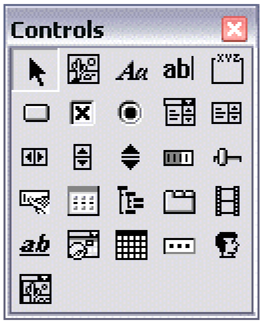
13. Return to MSVC

13.4 Control Design

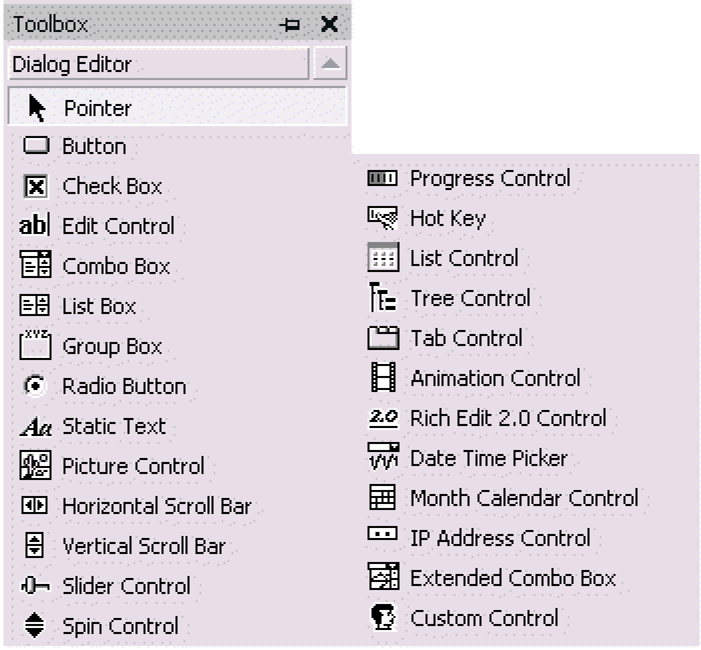
13.4.1 Controls Selection and Addition

The controls used to provide functionality to your application are provided by an object called the Controls toolbox. The Controls toolbox of both MSVC 6 and MSVC 7 gives access to 25 controls:

Visual C++ 6 Controls



Visual C++ 7 Controls

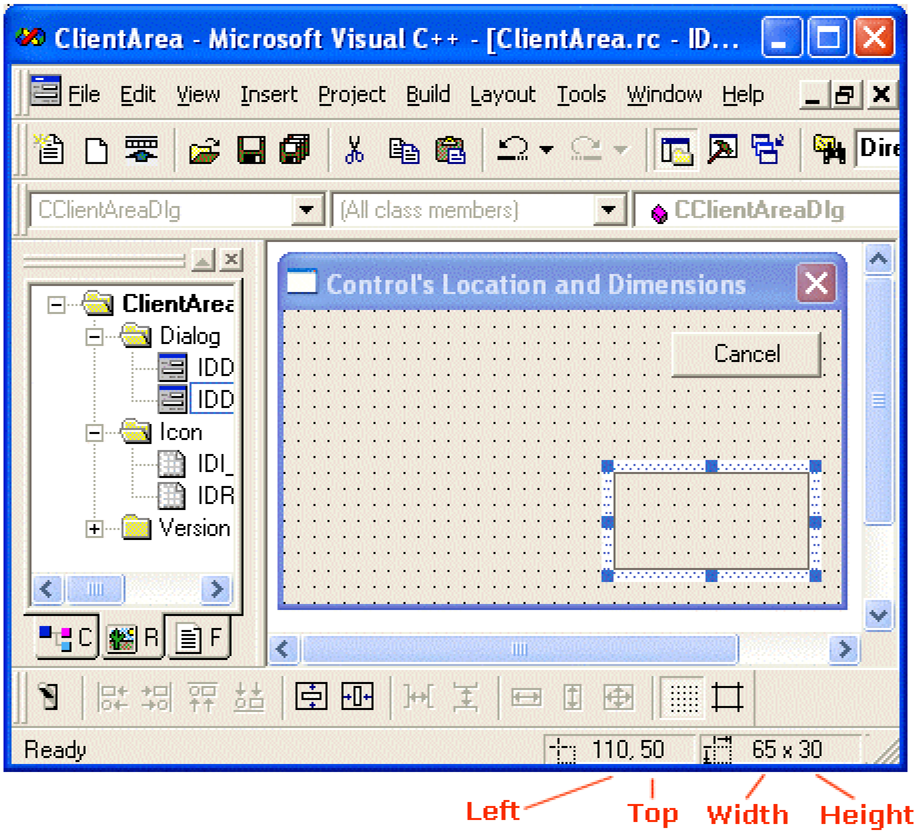


More objects, called ActiveX controls, can be accessed and added to the Controls toolbox but such additional objects are made available only to the current application. To make Windows controls available to your users, you will add them to their host object also called a parent. To do this, you mostly click the desired control from the Controls toolbox and click an area on the host. You can keep adding controls on the dialog box as necessary.

If you want to add a control over and over again, press and hold Ctrl. Then click the control on the Controls toolbox and release Ctrl. Then click in the desired area on the host. Every time you click, the control would be added to the form or dialog box. Once you have added enough controls, click the control again on the Controls toolbox to deselect it. You can also press Esc.

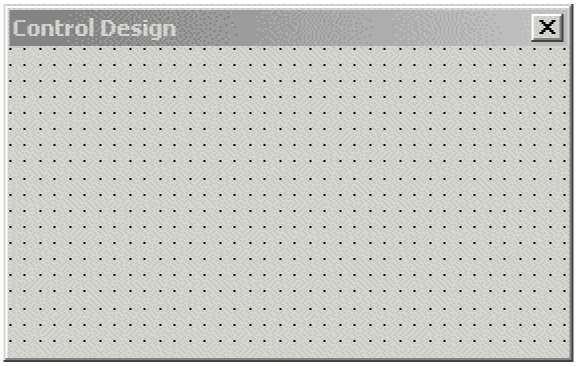
You cannot select more than one control to add on a host.

Another technique you can use to visually add a control is to “design” it. To do this, after clicking the control from the Controls toolbox, you can click and hold the mouse on the parent window, then drag left, up, right, or down. While you are dragging, you can refer to the right section of the status bar for the current location and dimensions of the control:


ClientArea - Microsoft Visual C++ - [ClientArea.rc - ID...
File Edit View Insert Project Build Layout Tools Window Help
CClientAreaDlg
(All class members)
CClientAreaDlg
ClientArea
Dialog
IDD
IDD
Icon
IDI_
IDR
Version
Control's Location and Dimensions
Cancel
Ready
110, 50
65 x 30
Left
Top
Width
Height

13.4.2 Control's Location and Size Using Grids

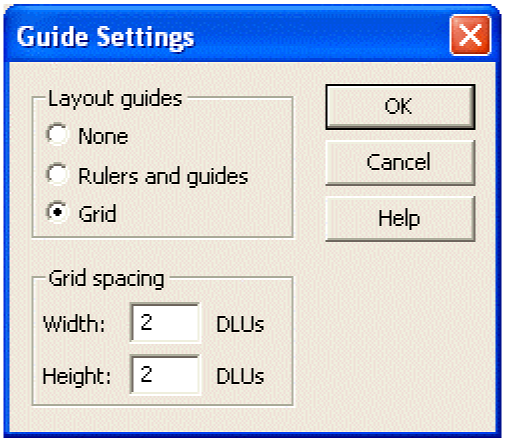
To help position the controls you are adding at design time, you can display some guiding grid dots. To do this, you can click the Toggle Grid button ▦. In this case, the control can be positioned and resized anywhere in the area inside the borders of the parent window, using the grids as guides, as seen on the above red rectangle. No control can be positioned outside of the client area. The dots on this view control the location and position of the control:



When you click the body of the parent, the top-left corner of the control would be positioned on a dot and the dimensions (width and height) of the control would align with the next dots on its right and bottom. To avoid this default behavior, before clicking the body of the parent, press and hold Alt; then click and drag to draw a rectangle shape of the control. The location would be where you clicked, even if you clicked between two dots. The dimensions would stop where you released the mouse, even it if was between two dots.

After placing a control on a parent window that displays grids, the borders of the child control can only be aligned with the black dots. The grids are separated using a measure called the Dialog Box Unit or DLU. By default, two vertically aligned dots are separated by a height of 5 DLUs. Two horizontally aligned dots are separated by a width of 5 DLUs. If these measures are too high, there are two options you can use:

- After placing the control, to control its location with more precision, press the up, left, right, or down arrow keys (on the keyboard). In this case, the control would move by one pixel in the direction of your choice. To control its size, press and hold Shift. Then press the up or down arrow keys to move only the bottom border of the control; or press the left or right keys to move the right border of the control.
- To modify the distance between dotted grids, display the Guide Settings dialog box available from the Layout (MSVC 6) or Format (MSVC 7) menu then modify the Width and/or Height values in the Grid Spacing section:

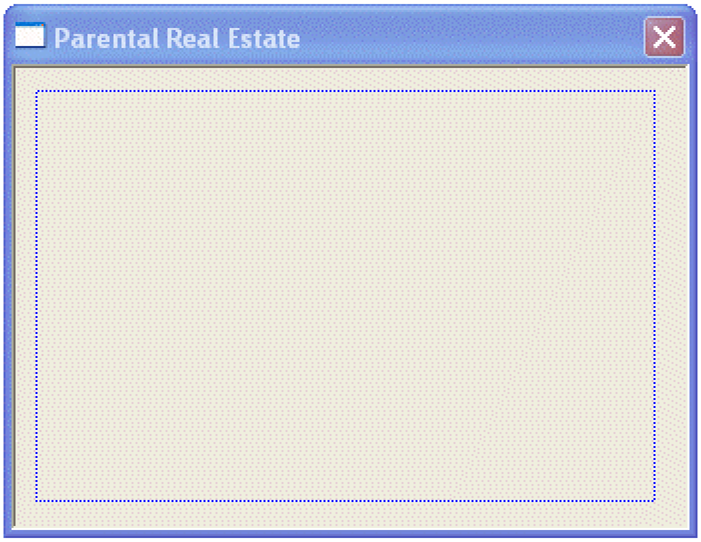


Layout guides
○ None
○ Rulers and guides
● Grid

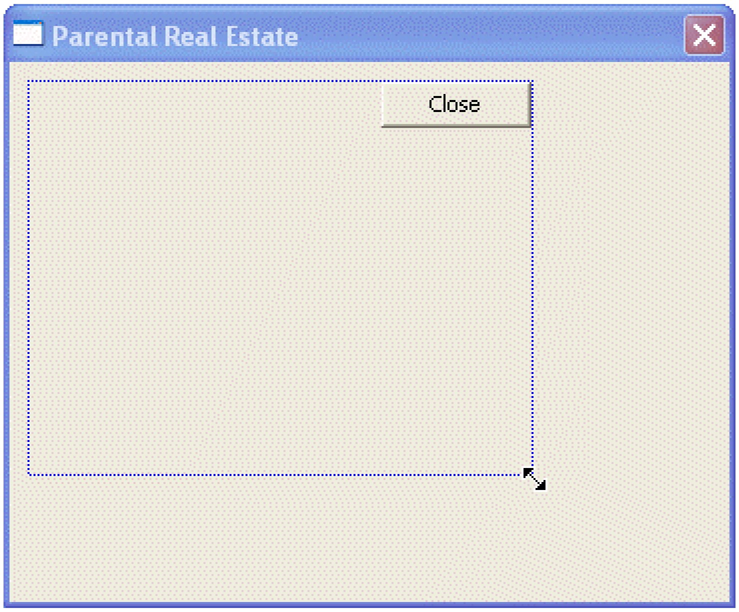
Grid spacing
Width: 2 DLUs
Height: 2 DLUs

13.4.3 Control's Location and Size Without Grids

Visual C++ also allows you to position controls without using grids. If you do not want to use the grids, you can hide them. This is done by clicking the Toggle Guides button, a (blue dotted) rectangle appears in the client area:



This (blue dotted) rectangle allows you to effectively control the area available to your controls. To set the location and dimensions of the available area, click one of the (blue dotted) rectangle borders or corners and drag in the desired direction. If a control is positioned on a border that is moving, the control would be repositioned accordingly:



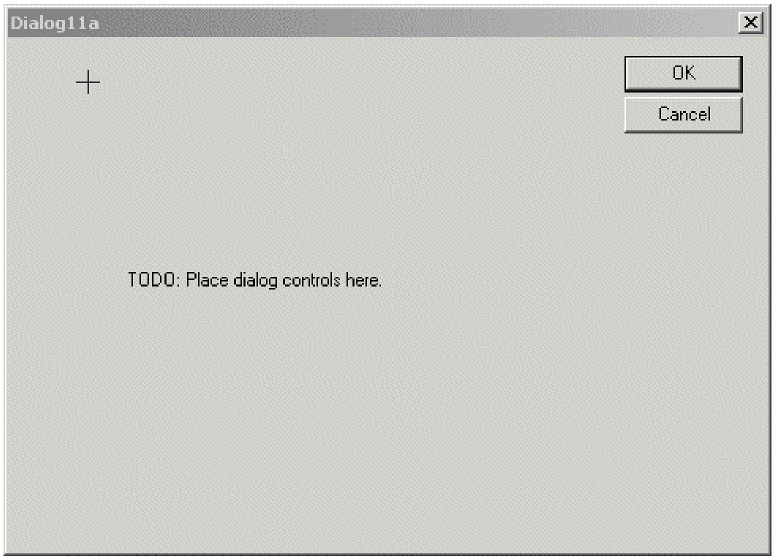
After setting the (blue dotted) rectangular location and area, you can add but cannot move a control outside of that (blue dotted) rectangle. The idea is to allow you to design a control or a group of controls in a (temporary) confined area. The (blue dotted) rectangle provides an effective means of aligning controls. After using it, if you want to add and manipulate controls outside of it, you should display the grids.

Practical Learning: Adding Controls to a Dialog Box

1. Create a new dialog-based application and name it Dialog11a

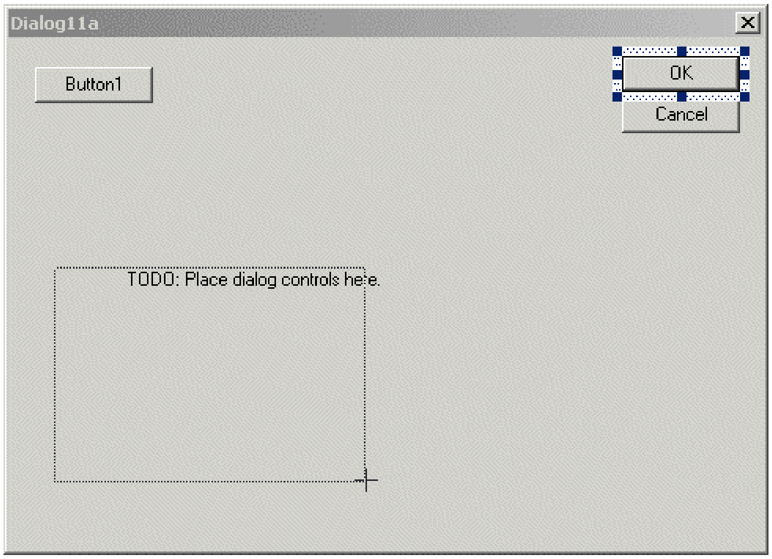
2. Click the Toggle Grid button ▦

3. On the Controls window, click the button control and click somewhere (no precision necessary) in the top left corner of the dialog box:



TODO: Place dialog controls here.

4. On the Controls window, click the Edit Box control
5. On the dialog box, click and hold the mouse in the middle-left (no precision needed) section of the dialog box
6. Drag down and right (no precision necessary):



TODO: Place dialog controls here.

7. Release the mouse
8. From the Controls window, click the Vertical Scroll Bar control and release the Ctrl key
9. On the dialog box, click three times an unoccupied area (no precision necessary) of the dialog box
10. To de select the Vertical Scroll Bar control , press Esc

13.4.4 Selecting Controls on a Parent Window

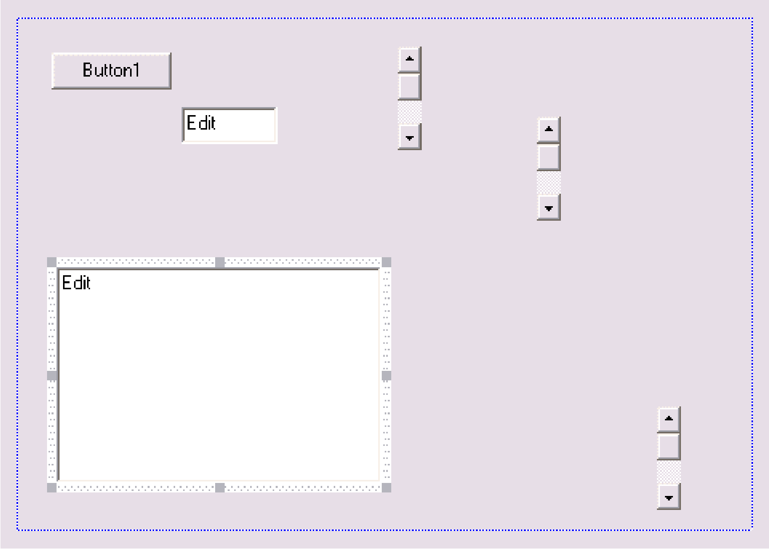
To visually manipulate a control, you will first need to select it. To select a control, simply click it. A control that is selected is surrounded with 8 (blue) handles.

To select more than one control in the same area, click on the dialog box and draw a "fake" rectangle that encloses all of the needed controls. The first control from the selected group has 8 (blue) handles while the other control(s) from the same selected group has (have) 8 white handles (each).

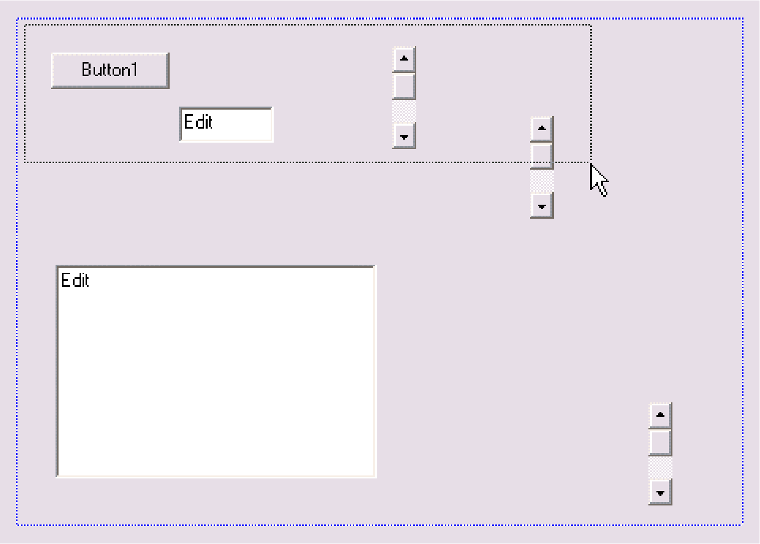
To select controls at random, click one of them. Press and hold either Shift or Ctrl. Then clickeach one of the needed controls.

Practical Learning: Selecting Controls on a Parent

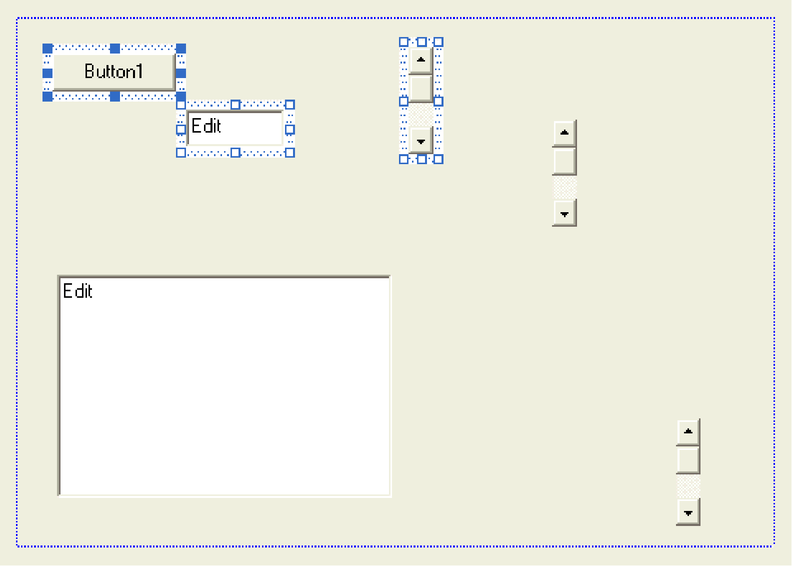
1. Open the Form11a application
2. If the form is not displaying, in the Resource View, expand the Form11a resources and expand the Dialog folder. Then double-click IDD_FORM11A_FORM
   On the form, click the lower Edit box. Notice that it displays 8 handles around:



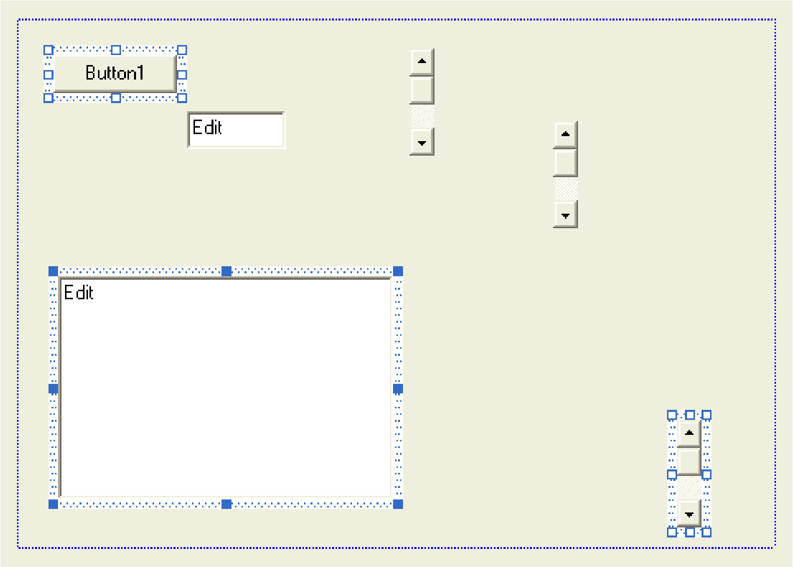
3. To select controls in a range, click and hold the mouse from the upper left section of the dialog box. Then drag down and to the right, drawing a rectangle that includes all three buttons in the top section of the dialog box:



4. Notice that the first control of the selection has 8 blue handles around:



5. Click an unoccupied area on the dialog box to unselect everything
6. To select controls at random, click the Button1 in the top-left section
7. Press and hold Shift
8. Then click the lower-right scroll bar. Click the lower left edit control
9. Release Shift



10. Click an empty area on the form to deselect the controls

13.4.5 Controls Resizing

After you have added a control to a dialog box, i assumes either its default size or the size you drew it with. To help with the sizes of controls on the form or dialog box, Visual C++ provides a visual grid made of black points. To can display or hide the grid by clicking the Toggle Grid button ▦. The dimensions of grid points are in dialog box units (DLUs) and are set at a default value of 5. To change this spacing, display the Guide Settings dialog box from the Layout menu and set the desired values.

To change the size of a control, first select it. Then position the mouse on one of its handles. The mouse would assume a sizing cursor that indicates the possible type of resizing you can apply. The mouse cursors are:

Cursor	Description
⤡	Moves the seized border in the North-West <-> South-East direction
↕	Shrinks or heightens the control
⤢	Moves the seized border in the North-East <-> South-West direction
↔	Narrows or enlarges the control

To resize a control, that is, to give it a particular width or height, position the mouse on one of the handles and drag in the desired direction.

If the Toggle Grid button is down, in which case the dialog or form would display the grid indicators, a control can be moved or resized only on the dotted lines. If you do not want to follow the grid indicators, you have two alternatives. You can hide the grid indicators by Toggle Guides button. On the other hand, you can press and hold Alt, then click the control or one its sizing handles and drag in the desired direction.

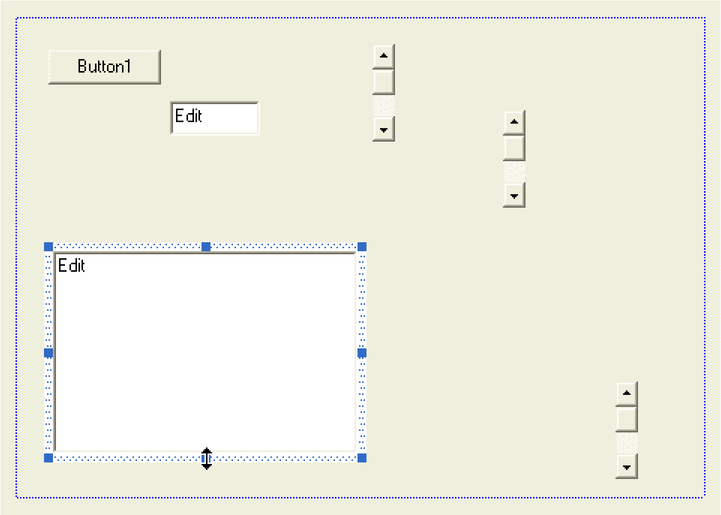
To resize more than one control at the same time. Firs select them. Then use the following buttons from the Dialog toolbar:

Button	Name	Description

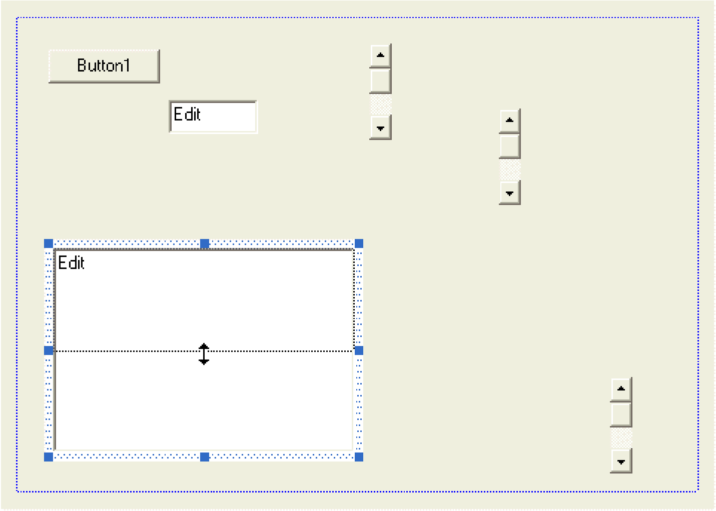
	Make Same Width	Applies the same width to all selected controls The width used will be be that of the last control selected
	Make Same Height	Applies the same height to all selected controls. The height used will be be that of the last control selected
	Make Same Size	Applies the same width and height to all selected controls. The height used will be be that of the last control selected

Practical Learning: Resizing Controls

1. Click the lower large Edit box to select it.
2. Position the mouse on the middle handle of the lower border:

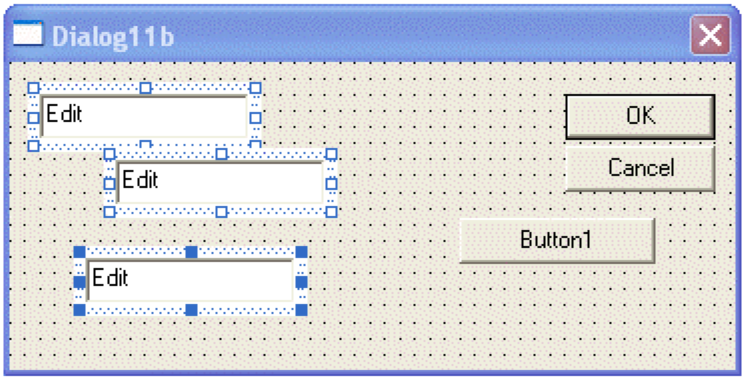


3. Click and drag up until the guiding horizontal line is in the middle of the control:



4. Release the mouse
5. Open the Dialog11b application

6. Display the dialog and click the Toggle Grid button
7. Click the top Edit control. Then press and hold Ctrl
8. Click the middle Edit control and click the bottom Edit control. Release Ctrl
9. On the Dialog toolbar, click the Make Same Width button



13.4.6 Controls Positions

The controls you position on a dialog box or a form assume their given place. Most of the time, these positions are not practical. You can move them around to any positions of your choice.

To move a control, click and drag it, while the mouse cursor is a cross , in the desired direction until it reaches the intended position.

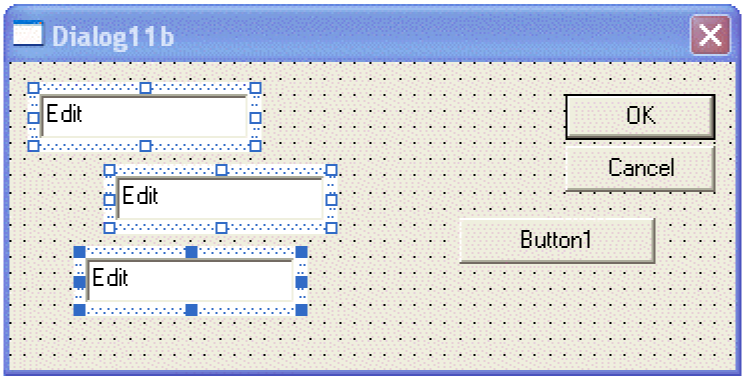
To move a group of controls, first select them. Then drag the selection to the desired location.

To help with positioning the controls, Visual C++ provides the Dialog toolbar with the following buttons:

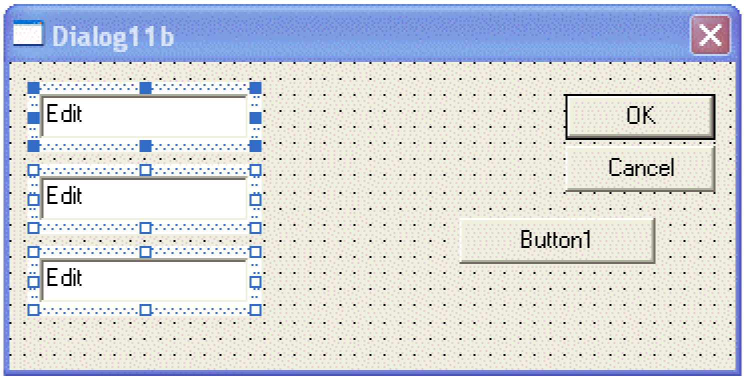
Button	Name	Button	Name	Button	Name
	Align Left		Align Tops		Vertical
	Align Right		Align Bottoms		Horizontal
	Across		Down		

Practical Learning: Moving Controls

1. While the edit controls are still selected, on the Dialog toolbar, click the Down button



2. Click an empty area on the dialog box to deselect everything.
3. Click the lower Edit control. Then press and hold Shift
4. Click the middle Edit control and click the top Edit control then release Shift
5. On the Dialog toolbar, click the Align Left button



6. Save everything and close the application

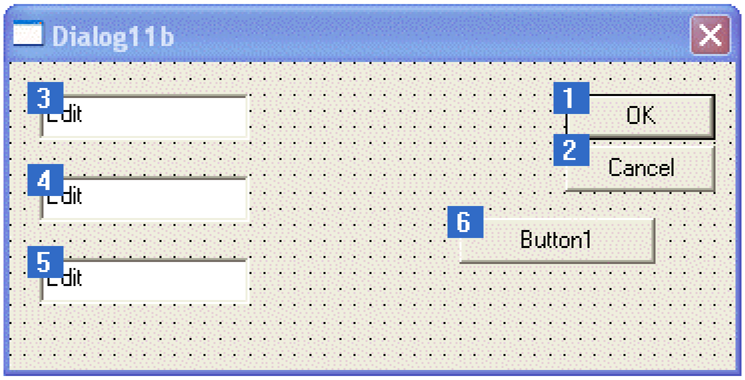
13.4.7 Tab Ordering

The controls you add to a form or a dialog box are positioned in a sequence that follows the order they were added. When you add a control on the host that already has other controls, regardless of the section or area you place the new control, it is sequentially positioned at the end of the existing controls. If you do not fix it, the user would have a hard time navigating the controls.

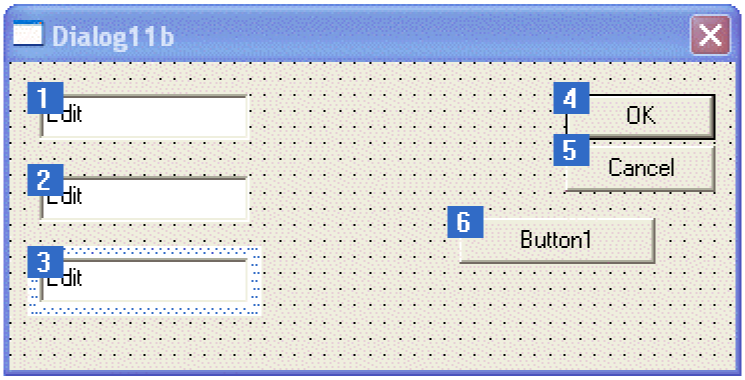
The sequence of controls navigation is called the tab order. While designing a form or a dialog box, to change the sequential order of controls, on the main menu, click Layout or Format and click Tab Order or press Ctrl + D.

Practical Learning: Controlling Tab Order

1. On the main menu, click Layout (Visual C++ 6) or Format (Visual C++ 7) and click Tab Order



2. To change the order or sequence of controls, click the top Edit control. Notice that its order changes to 1
3. Click the middle Edit control, then the lower Edit control



4. Save everything

Chapter 14: Controls Functionality

- ✍ Handling Controls
- ✍ Controls Management
- ✍ Controls Instance and Handle
- ☞ Access to Controls Information

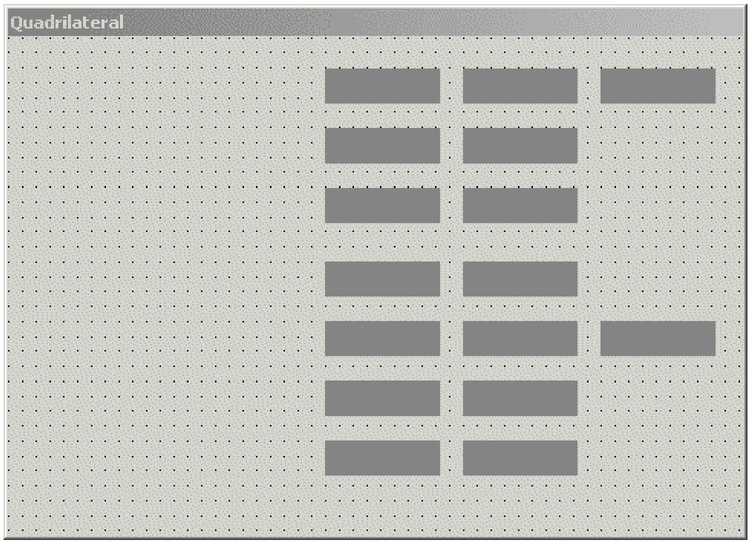
14.1 Handling Controls

14.1.1 Introduction

There are two particularly important pieces of information about every control of your application. The information stored in the control and the action or type of action the user can perform on that control. At its most basic level, at one time in the life of an application, a control holds a value that can be of particular interest either to you or to the user. On the other hand, when interacting with the computer, based on your application, the user will usually face different types of controls that do various things and produce different results. These are two features of controls (their values and the actions the user can perform on them) that you should know as much as possible, about the controls you choose to involve in your application.

Practical Learning: Introducing Controls Variables

1. Open the Geometry application. If you do not have it, open the Geometry1 application from the accompanying exercises of this book
2. Test the application to make sure it is working fine. Then close it and return to MSVC
3. Click the Resourve View tab, expand the Dialog folder and double-click IDD_QUADRILATERAL. Delete the existing control on the dialog box
4. On the Controls toolbox, click the Custom Control button and click in the top center section of the dialog box. On the Properties window:
   Change its ID to **IDC_LBL_SSIDE**
   Set its Caption to **&Side:**
   Set the Class name as **Static**
5. Using only the Custom Control object, design the rest of the property page as follows (the controls are listed from left to right then from up -> down):



ID	Caption	Class	Style	ExStyle

IDC_LBL_SSIDE	&Side:	Static	0x50010000	0x0
IDC_EDT_SSIDE		Edit	0x50810000	0x0
IDC_BTN_SCALC	&Calculate	Button	0x50010000	0x0
IDC_LBL_SPRM	&Perimeter:	Static	0x50010000	0x0
IDC_EDT_SPRM		Edit	0x50810000	0x1
IDC_LBL_SAREA	&Area:	Static	0x50010000	0x0
IDC_EDT_SAREA		Edit	0x50800000	0x1
IDC_LBL_RLENGTH	&Length:	Static	0x50010000	0x0
IDC_EDT_RLENGTH		Edit	0x50810000	0x0
IDC_LBL_RHEIGHT	&Height:	Static	0x50010000	0x0
IDC_EDT_RHEIGHT		Edit	0x50810000	0x0
IDC_BTN_RCALC	Calc&ulate	Button	0x50010000	0x0
IDC_LBL_RPRM	P&erimeter:	Static	0x50010000	0x0
IDC_EDT_RPRM		Edit	0x50810000	0x1
IDC_LBL_RAREA	A&rea:	Static	0x50010000	0x0
IDC_EDT_RAREA		Edit	0x50810000	0x1

6. Save All

7. Click anywhere on the dialog box to make sure it has focus. Press Ctrl + A to select all controls. Press Ctrl + C to copy the selection

8. Open the IDD_CIRCULAR dialog box. Delete its control and drag its lower-right corner to give a 320 x 200 size

9. Click anywhere in the body of the dyalog box. Press Ctrl + V to paste the selection

10. Change the IDs and captions of the controls as follows (from left to right and up -> down):

ID	Caption	Class	Style	ExStyle
IDC_LBL_CRADIUS	&Radius:	Static	0x50010000	0x0
IDC_EDT_CRADIUS		Edit	0x50810000	0x0
IDC_BTN_CCALC	&Calculate	Button	0x50010000	0x0
IDC_LBL_CCIRC	Circum&ference:	Static	0x50010000	0x0
IDC_EDT_CCIRC		Edit	0x50810000	0x1
IDC_LBL_CAREA	&Area:	Static	0x50010000	0x0
IDC_EDT_CAREA		Edit	0x50800000	0x1
IDC_LBL_VRADIUS	Radiu&s:	Static	0x50010000	0x0
IDC_EDT_VRADIUS		Edit	0x50810000	0x0
IDC_LBL_HRADIUS	Ra&dius:	Static	0x50010000	0x0
IDC_EDT_HRADIUS		Edit	0x50810000	**0x0**
IDC_BTN_ECALC	Calc&ulate	Button	0x50010000	0x0
IDC_LBL_ECIRC	Circu&mference:	Static	0x50010000	0x1
IDC_EDT_ECIRC		Edit	0x50810000	0x1
IDC_LBL_EAREA	Ar&ea:	Static	0x50010000	0x0
IDC_EDT_EAREA		Edit	0x50810000	0x0

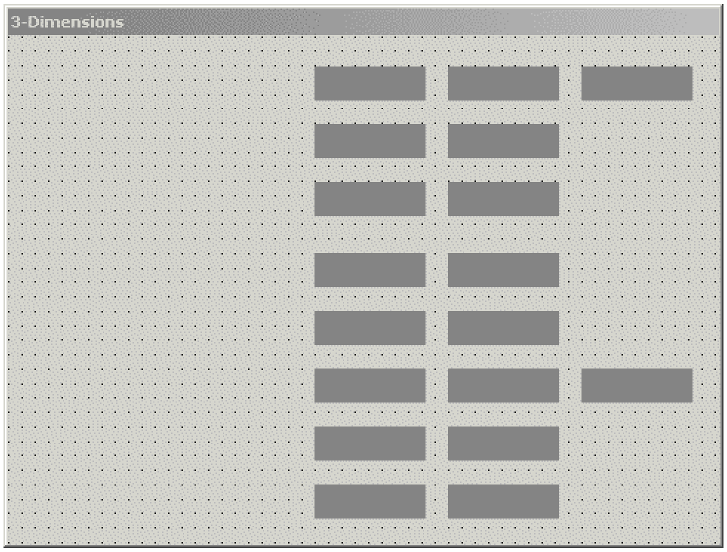
11. Save All. Just in case you may have change the contents of the clipboard, click the body of the dialog box, press Ctrl + A and press Ctrl + C

12. Open the IDD_G3D dialog box. Delete its control and set its size to 320 x 210. Then, click its body

13. Press Ctrl + V to paste

14. Select a left and right combination of controls on the dialog box. Copy and past them under the existing controls

15. Change the IDs, captions and styles of the controls as follows (from left to right and up -> down):



ID	Caption	Class	Style	ExStyle
IDC_LBL_USIDE	&Side:	Static	0x50010000	0x0
IDC_EDT_USIDE		Edit	0x50810000	0x0
IDC_BTN_UCALC	&Calculate	Button	0x50010000	0x0
IDC_LBL_UAREA	&Area:	Static	0x50010000	0x0
IDC_EDT_UAREA		Edit	0x50810000	0x1
IDC_LBL_UVOL	&Volume:	Static	0x50010000	0x0
IDC_EDT_UVOL		Edit	0x50800000	0x1
IDC_LBL_BLENGTH	&Length:	Static	0x50010000	0x0
IDC_EDT_BLENGTH		Edit	0x50810000	0x0
IDC_LBL_BHEIGHT	&Height:	Static	0x50010000	0x0
IDC_EDT_BHEIGHT		Edit	0x50810000	**0x0**
IDC_LBL_BWIDTH	&Width	Static	0x50010000	0x0
IDC_EDT_BWIDTH		Edit	0x50810000	**0x0**
IDC_BTN_BCALC	Calc&ulate	Button	0x50010000	0x0
IDC_LBL_BAREA	A&rea:	Static	0x50010000	0x1
IDC_EDT_BAREA		Edit	0x50810000	0x1
IDC_LBL_BVOL	V&olume:	Static	0x50010000	0x0
IDC_EDT_BVOL		Edit	0x50810000	0x0

16. Close MSVC completely. When asked to save, click Yes as many times as you are asked.

14.1.2 Control's Control Variables:

After visually adding a control to your application, if you want to refer to it in your code, you can declare a variable based on, or associated with, that control. The MFC library allows you to declare two types of variables for some of the controls used in an application: a value or a control variables.

A control variable is a variable based on the class that manages the control. For example, a static control is based on the **CStatic** class. Therefore, if you add a static control to your application and you want to refer to it later on, you can declare a **CStatic** variable in the header of its parent window. Here is an example:

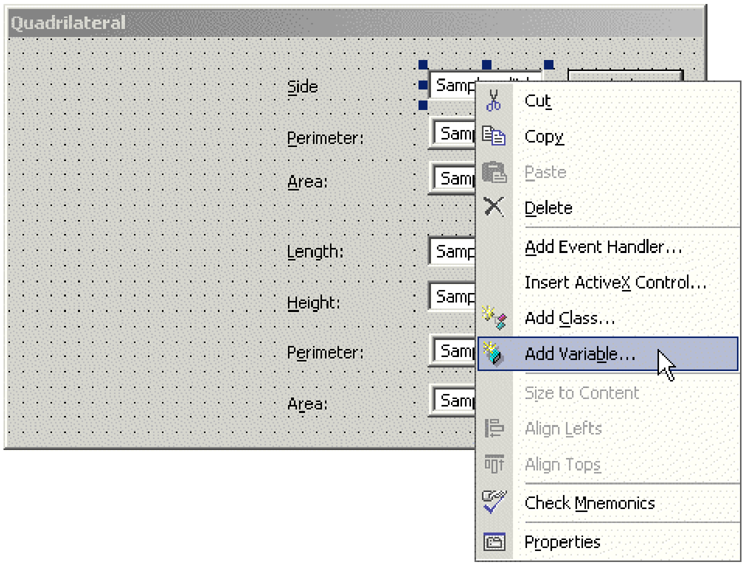
```
class CExchangeDlg : public CDialog
{
	. . .
public:
	CStatic stcAdvanced;
};
```

The control variable is available for all controls.

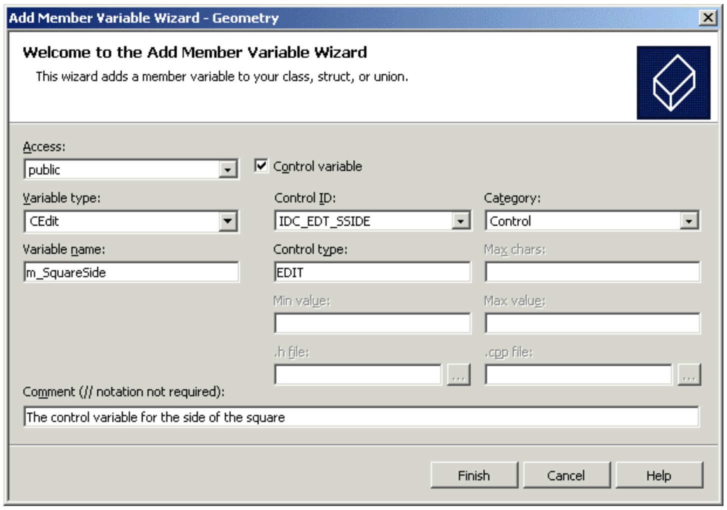
It is a habit for Visual C++ programmers to start the name of a control variable with m_

Practical Learning: Adding Control Variables

1. Open MSVC again and open the Geometry application created above. If you do not have it, open the Geometry2 application that accompanies this book
2. Open the IDD_QUADRILATERAL dialog box and adjust the locations and dimensions as you see fit or open the Geometry2 application that accompanies this book
3. To declare a control variable, on the dialog box, right-click the edit box on the right side of Side and click Add Variable…



4. In the Add Member Variable Wizard,
in the Variable Type combo box, select **CEdit**
in the Category combo box, select **Control**
in the Variable Name edit box, type **m_SquareSide**



Add Member Variable Wizard - Geometry

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: public

☑ Control variable

Variable type: CEdit

Control ID: IDC_EDT_SSIDE

Category: Control

Variable name: m_SquareSide

Control type: EDIT

Max chars:

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required): The control variable for the side of the square

5. Click Finish
6. In the same way, **Add** a **CEdit Control Variable** for each edit box on the propety page and name them, from top to bottom, as **m_SquarePerimeter**, **m_SquareArea**, **m_RectLength**, **m_RectHeight**, **m_RectPerimeter**, and **m_RectArea**
7. Open the Quadrilateral.h file and check that the variables were added:

```
// Quadrilateral.h : header file
//

#pragma once
#include "afxwin.h"

// CQuadrilateral dialog
class CQuadrilateral : public CPropertyPage
{
// Construction
public:
	CQuadrilateral(CWnd* pParent = NULL);	// standard constructor

// Dialog Data
	enum { IDD = IDD_QUADRILATERAL };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);  // DDX/DDV support

// Implementation
protected:
	HICON m_hIcon;

	// Generated message map functions
	virtual BOOL OnInitDialog();
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
public:
	// The control variables for the square
	CEdit m_SquareSide;
	CEdit m_SquarePerimeter;
	CEdit m_SquareArea;
	// The control variables for the rectangle
	CEdit m_RectLength;
```

```
        CEdit m_RectHeight;
        CEdit m_RectPerimeter;
        CEdit m_RectArea;
};
```

8. Save All

14.1.3 The Control's Data Exchange

After declaring the variable, you must "map" it to the control it is referring to, otherwise the variable would behave just like any other and it cannot directly access the control you wanted it to refer to. To specify what control your variable refers to, you must call the **DDX_Control()** framework function. The syntax of this function is:

```
void AFXAPI DDX_Control(CDataExchange* pDX, int nIDC, CWnd& rControl);
```

The first argument, *pDX*, is a pointer to **CDataExchange**. The *nIDC* argument is the identifier of the control your variable will refer to. The *rControl* argument is the name you gave to your variable. An example of calling this function would be:

```
DDX_Control(pDX, IDC_STATIC_ADV, stcAdvanced);
```

The *pDX* argument in reality handles the mapping. It creates the relationship between your *rControl* variable and the *nIDC* control that *rControl* must refer to. Besides that, *pDX* insures that information can flow easily between both entities. The **CDataExchange** class, to which the *pDX* argument points, is a parent-less class, meaning it is based neither on **CObject** nor on **CWnd**.

The **DDX_Control()** function can be called for each variable you intend to map to a control. When calling any of these functions, the mappings must be performed in the **CWnd::DoDataExchange()** event. Its syntax is:

```
virtual void DoDataExchange(CDataExchange* pDX);
```

As you can see, this event is passed a **CDataExchange** pointer. This pointer in turn will become the first argument to the **DDX_Control()** function. Behind the scenes, this allows the dialog box, the parent of the controls to better manage the exchange of information between the application and the controls.

This review helps to have an idea of how variables would be declared and associated with the intended controls. In reality, this job is usually, and should always be, handled by Visual C++ for you. When you create a dialog-based object, such as a dialog box, a form, a property sheet, or a property page, the **CDialog::DoDataExchange()** event is created and made ready for you. To declare a variable you want to associate to a control, unless you have a good reason to proceed manually, use either the ClassWizard in MSVC 6 or the Add Member Variable Wizard in MSVC 7 to add the variable. When you do this, the wizard will take care of all the mapping for you.

After adding a variable using a wizard and once the variable mapping has been performed, if you change the name of the variable in the header file, you must manually change its name in the **DoDataExchange()** event.

Practical Learning: Checking Control Variables

1. Open the Quadrilateral.cpp file and check the DoDataExchange event:

```
void CQuadrilateral::DoDataExchange(CDataExchange* pDX)
{
	CPropertyPage::DoDataExchange(pDX);
	DDX_Control(pDX, IDC_EDT_SSIDE, m_SquareSide);
	DDX_Control(pDX, IDC_EDT_SPRM, m_SquarePerimeter);
	DDX_Control(pDX, IDC_EDT_SAREA, m_SquareArea);
	DDX_Control(pDX, IDC_EDT_RLENGTH, m_RectLength);
	DDX_Control(pDX, IDC_EDT_RHEIGHT, m_RectHeight);
	DDX_Control(pDX, IDC_EDT_RPRM, m_RectPerimeter);
	DDX_Control(pDX, IDC_EDT_RAREA, m_RectArea);
}
```

2. Save All

14.1.4 Control's Value Variables

Another type of variable you can declare for a control is the value variable. Not all controls provide a value variable. The value variable must be able to handle the type of value stored in the control it is intended to refer to. For example, because a text-based control is used to handle text, you can declare a text-based data type for it. This would usually be a CString variable. Here is an example:

```
class CExchangeDlg : public CDialog
{
	. . .
public:
	CStatic stcAdvanced;
	CString strFullName;
};
```

On the other hand, as we will learn that some controls handle a value that can be true or false at one time, namely the check box, you can declare a Boolean variable for such controls. Some other controls are used to hold a numeric value such as a natural or a floating-point number. You can declare an integer-based or a float-based value variable for such controls. Some other controls are not meant to hold an explicit or recognizable type of data, an example would be the tab control. For such controls, there is no value variable available. For such controls, you can only declare a control variable.

After declaring the value variable, as done with the control variable, you must "map" it to the intended control. To do this, you must call an appropriate framework function. The functions are categorized based on the type of data held by the variable. For example, if the control holds text and you had declared a CString value variable for it, you can call the DDX_Text() function to map it. The DDX_Text() function is provided in various versions as follows:

```
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, BYTE& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, short& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, int& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, UINT& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, long& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, DWORD& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, CString& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, float& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, double& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, COleCurrency& value );
void AFXAPI DDX_Text( CDataExchange* pDX, int nIDC, COleDateTime& value );
```

The first argument, pDX, is a pointer to CDataExchange. The second argument, nIDC, is the identifier the of the control you want to refer to. The third argument, value, if the name of the value variable you had previously declared.

If a control holds a valid value, you can declare both a control variable and a value variable if you need them. Here is an example:

```
class CExchangeDlg : public CDialog
{
	. . .
public:
	CStatic stcAdvanced;
	CString valAdvanced;
	CString strFullName;
};
```

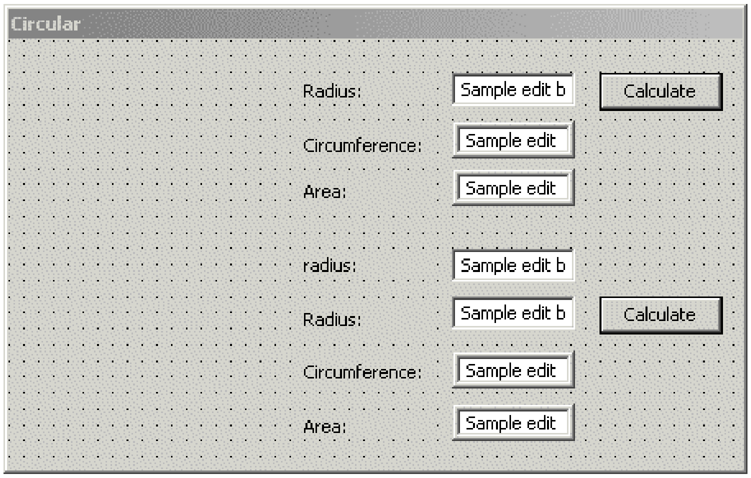
When mapping such variable, make a call to **DDX_Control()** for the control variable and a separate call to **DDX_*X*()** for the value variable. Because there are various types of DDX_ functions for the controls, *X* stands for the type of control referred to.

Like the **DDX_Control()**, the **DDX_** X**()** function can be called for each variable you intend to map to a control and this is performed in the **CWnd::DoDataExchange()** event. This **CDataExchange** pointer is passed as the first argument to the **DDX_ *X*()** functions. This argument allows the dialog boxto update the values held by the controls.

Once again, avoid declaring and mapping value variables manually. Instead, use either (MSVC 6) the ClassWizard or (MSVC 7) the Add Member Variable Wizard to add a variable that you want to associate with a control on a dialog-based object.

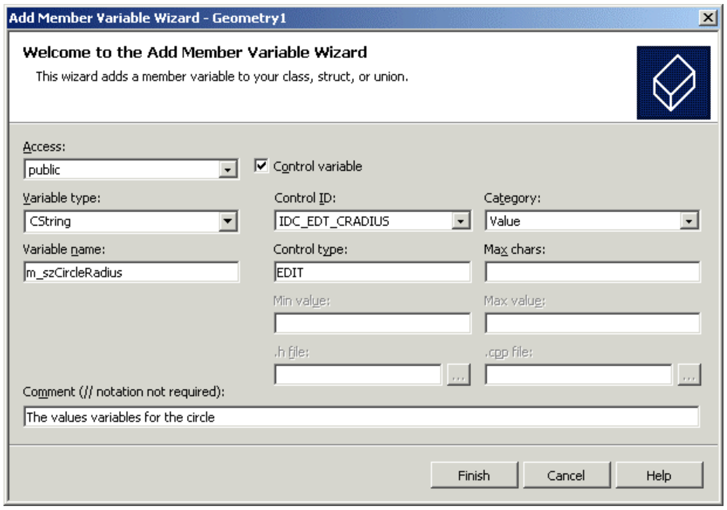
Practical Learning: Adding Value Variables

1. From the Resource View tab, open the IDD_CIRCULAR dialog box and adjust the locations and dimensions of the controls as you see fit



2. To declare a value variable, on the dialog box, right-click the edit box on the right side of the top Radius and click Add Variable…

3. Click the arrow of the Category combo box and select Value
   In the Variable Type combo box, if necessary, select CString
   In the Variable Name edit box, type **m_szCircleRadius**



Add Member Variable Wizard - Geometry1

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: public
☑ Control variable
Variable type: CString
Control ID: IDC_EDT_CRADIUS
Category: Value
Variable name: m_szCircleRadius
Control type: EDIT
Max chars:
Min value:
Max value:
.h file:
.cpp file:
Comment (// notation not required): The values variables for the circle

4. Click Finish

5. In the same way, Add a CString Value Variable for each edit box. From top to bottom, name them **m_szCircleCircumference**, **m_szCircleArea**, **m_szEllipseradius**, **m_szEllipseRadius**, **m_szEllipseCircumference**, and **m_szEllipseArea** respectively

6. Open the Circular.h header file and check the new value variable

7. Also open the Circular.cpp source file and check the content of the DoDataExchange() event.

14.1.5 Controls Event Handlers

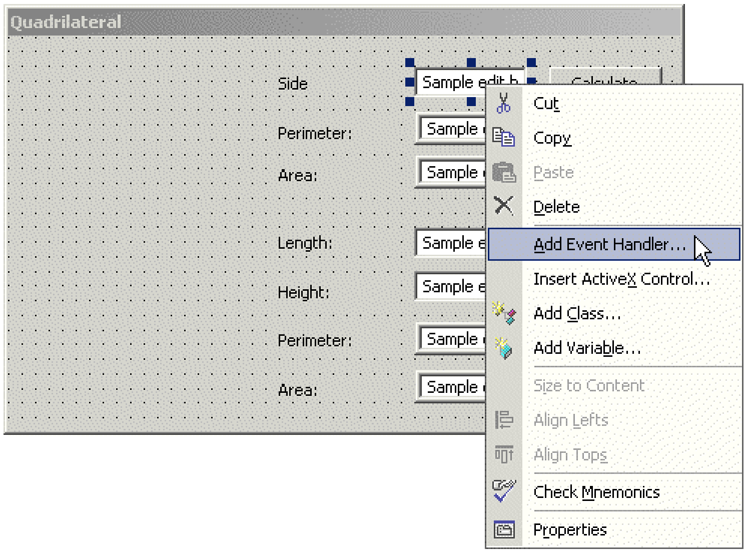
As we reviewed when introduced messages, an event is an action resulting from a generated message. As there are various types of controls, there are also different types of messages available on controls and even the type of window considered. After adding a control to your application, whether you visually added it or created it dynamically, you will also decide how the handle the possible actions that the user can perform on the control.

There are various ways you can write an event for a message. If the message is sent from a class, first select the class in Class View. Then, in the Properties window, click either the Events , the Messages , or the Overrides button, depending on the type of member function you want to write.

Some of the most regular actions the user can perform on a control is to click it. Object that receive click messages range from static controls to command buttons, as we will see eventually. To generate code for a control that is positioned on a dialog-based object, display its parent window. Then, right-click the control and, on the context menu, click Add Event Handler. This would display the Event Handler Wizard with as much information as you need to set or select to configure the event. Once you have finish specifying the necessary items for the event, click Add And Edit. This would take you to the new event in the Code Editor where you can write the rest of the code for the event. If you use the wizard to generate the event, the necessary arguments and part of the code for the event would be supplied to you. If you decide to write the event from scratch, you will need to especially remember the arguments of the event, otherwise the code may not work at all.

Practical Learning: Generating Events

1. From the Resource View tab, display the IDD_QUADRILATERAL dialog box
   On the dialog, right-click the edit box on the right side of Side and click Add Event Handler…



2. See the types of messages for the edit box. Click Cancel
3. Right-click the top Calculate control and click Add Event Handler…
4. Read and accept everything that is set on the wizard
5. Click Add And Edit
6. In the same way, generate an event for all Calculate controls of the IDD_QUADRILATERAL, the IDD_CIRCULAR, and the IDD_G3D dialog boxes
7. Save All

14.2 Controls Management

14.2.1 Control's Identification

So far, we have seen various techniques of visually adding or dynamically creating controls for your application. We also saw that every control should have identification. In fact, if you want to avoid providing an identification for a control, you may have to use the Win32 API's **CreateWindow()** or **CreateWindowEx()** function to create the control. We saw that the identifier can be created either using the Properties window, using the Resources Symbols dialog box, using the String Table, or when dynamically creating the control. We also saw how to declare or add a control variable for an MFC control.

Imagine a control with its identifier has already been added to a dialog box but you want to change that identifier, for any reason you judge necessary. To do this, you would call the CWnd::SetDlgCtrlID() method. Its syntax is:

```
int SetDlgCtrlID(int nID);
```

This method assigns or changes the identifier of the control that called it. Imagine the control variable of a control is named m_Edition, you can assign a new identifier to it as follows:

```
BOOL CDlgControls::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here

	m_Edition.SetDlgCtrlID(0x1882);

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

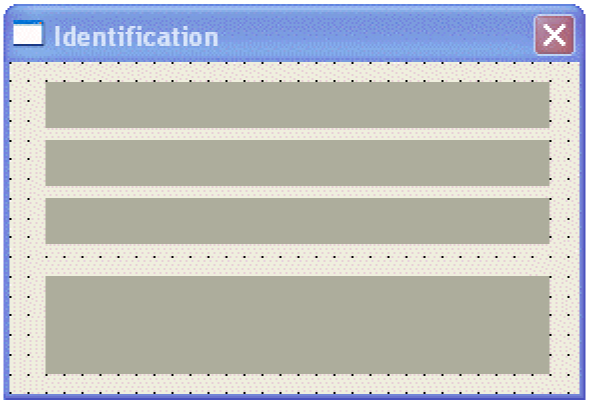
If a control has been created and you need its identifier, you can find it out by calling the **CWnd::GetDlgCtrlID()** method. Its syntax is:

```
int GetDlgCtrlID() const;
```

This method simply returns the identifier of the control that called it. The value is returned as an integer. You will mostly use this method to find out if the identifier of a control matches the one the user has just accessed.

Practical Learning: Using Controls Identifiers

1. Start a new MFC Application named **Identification** and create it as Dialog Based without an About Box
2. Delete the TODO line, the OK and the Cancel buttons
3. From the Controls toolbox, add four Custom Controls and draw them horizontally on the dialog box as follows:



4. Set the **Class** name of the top three custom controls to **ScrollBar**

5. Set the **Class** name of the bottom custom control to **Static** and change its **Caption** to **Burkina Faso**

6. Add a **CScrollBar Control Variable** to the top control and name it **m_ScrollRed** (if you are using some versions of MSVC 7 and the class name is not selected as CScrollBar, you may have to type it; if you are using MSVC 6, you should only select the class name as CScrollBar)

7. Add a **CScrollBar Control Variable** to the second control from top and name it **m_ScrollGreen**

8. Add a **CScrollBar Control Variable** to the third control from top and name it **m_ScrollBlue**

9. A **CStatic Control Variable** (if you are using some versions of MSVC 7 and the class name is not selected as CStatic, you can to type it; if you are using MSVC 6, you should select the CStatic class name) to the bottom control and name it **m_Country**

10. To change the identifiers of the top three controls, access the **OnInitDialog()** event of the dialog box and call the **SetDlgCtrlID()** method for each control giving the **1804**, **6255**, and **42** values respectively:

```
BOOL CIdentificationDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_ScrollRed.SetDlgCtrlID(1804);
	m_ScrollGreen.SetDlgCtrlID(6255);
	m_ScrollBlue.SetDlgCtrlID(42);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

11. When the user clicks one of the scroll bars, we will need to know which one was clicked and display a message accordingly

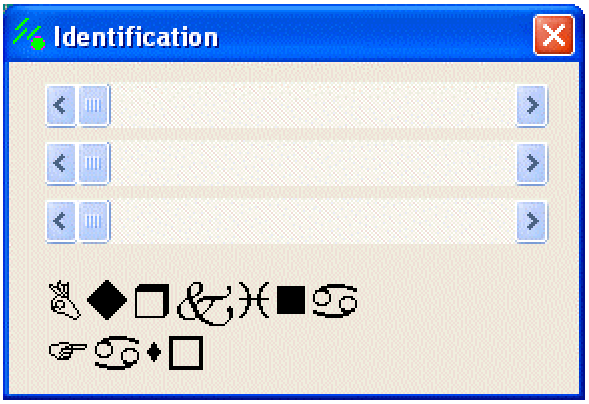
    Using the Messages button, generate the **WM_HSCROLL** message for the dialog box to **<Add>** an **OnHScroll** event.

12. Implement it as follows:

```
void CIdentificationDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
	// TODO: Add your message handler code here and/or call default
	CFont font;

	// Change the font of the label based on which scroll bar was clicked
	if( pScrollBar->GetDlgCtrlID() == 1804 )
	{
		font.CreatePointFont(260, "Garamond");
		m_Country.SetFont(&font);
	}
	else if( pScrollBar->GetDlgCtrlID() == 6255 )
	{
		font.CreatePointFont(220, "Arial");
		m_Country.SetFont(&font);
```

```
    }
    else if( pScrollBar->GetDlgCtrlID() == 42 )
    {
            font.CreatePointFont(180, "Wingdings");
            m_Country.SetFont(&font);
    }

    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

13. Test the application by executing it and trying to click the arrow buttons on the scroll bar controls

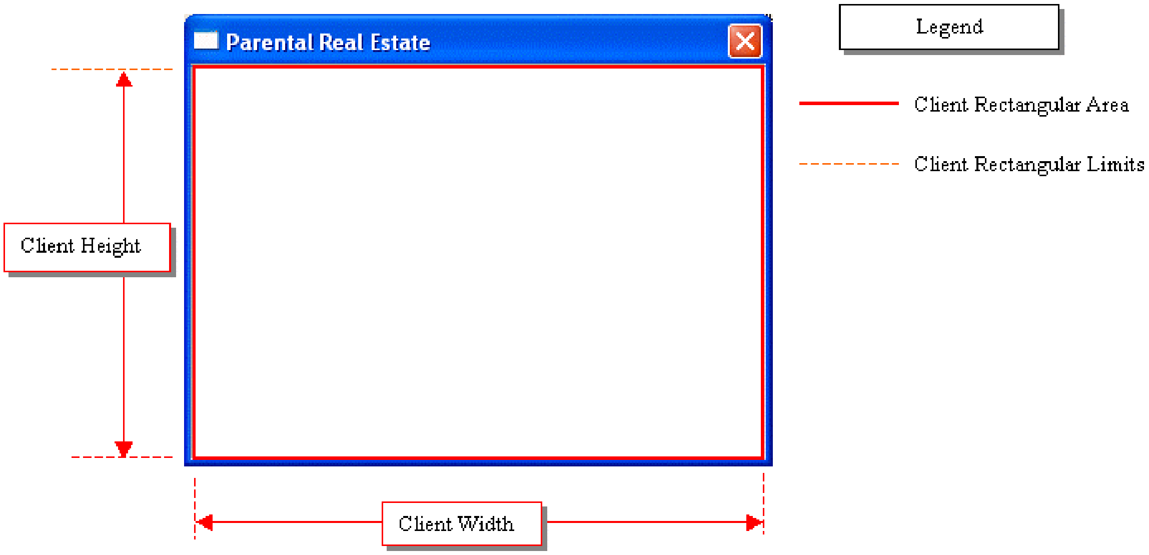


14. Close the application and return to MSVC

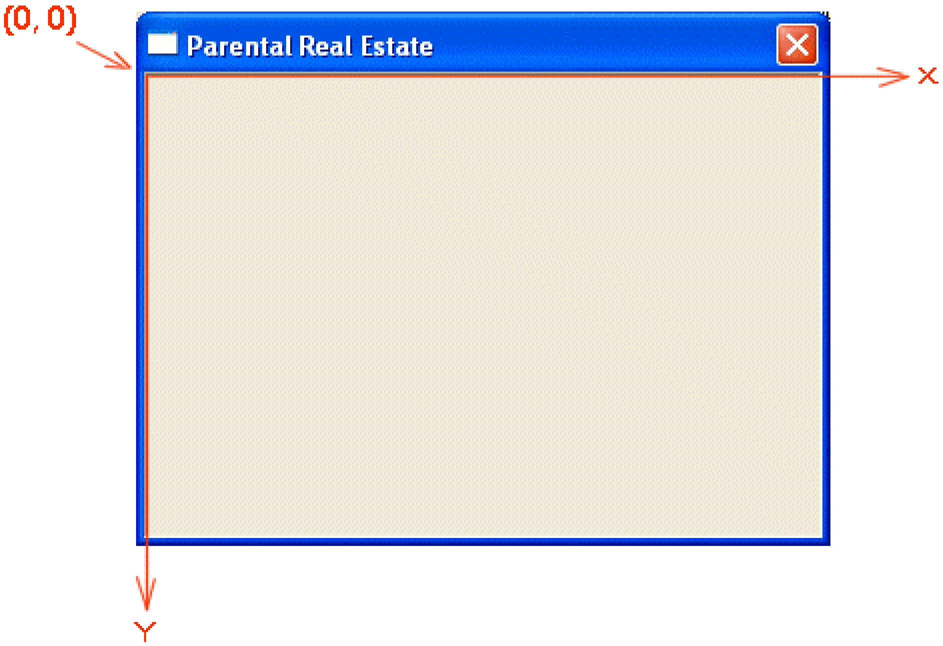
14.2.2 The Client Area

Besides the identifier, we learned that, to create a control, you must provide it with "physical" presence. If you add the control visually, it assumes the position where you place it. If you are programmatically creating the control, you must know how much space its parent window is making available to its children before even deciding about its location and dimensions.

To provide its parental support to the child controls, the parent window allocates an amount of space in a rectangular shape called the client area:



The controls you add are confined to the client area offered by the parent window. After visually adding a control to a parent window, it assumes a location and takes some dimensions in this area. The origin of the rectangular client area is on the upper-left corner of the parent window. The horizontal measurements move from the origin to the right. The vertical measurements move from the origin to the bottom:



A control can be added only in the client area. To find out how much room a parent window is making available to its children, that is, to get the dimensions (and screen location) of the client area, you can call the **CWnd::GetClientRect()** member function. Its syntax is:

```
void GetClientRect(LPRECT lpRect) const;
```

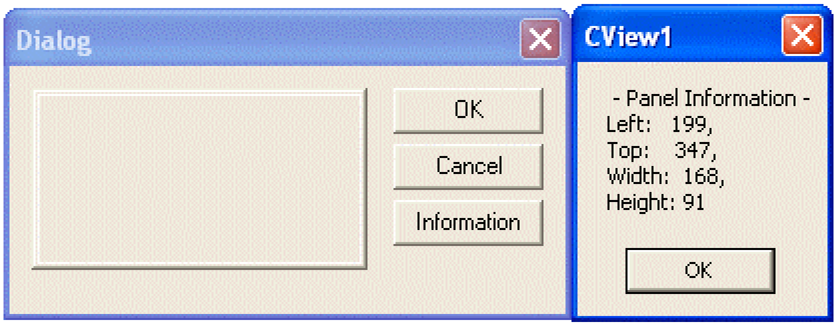
This member function takes as argument a **RECT** or **CRect** variable and stores the location and dimension of the client rectangular area in it. In the following example, the

GetClientRect() function is called to get the dimensions of the view's client area of a frame-based application and use the resulting rectangle to paint that area:

```
void CCView1View::OnPaint()
{
	CPaintDC dc(this); // device context for painting

	// TODO: Add your message handler code here
	CRect Recto;
	CBrush SelectedBrush(SelectedColor);
	CPen   SelectedBlue(PS_SOLID, 1, SelectedColor);

	GetClientRect(&Recto);
	CBrush *pOldBrush = dc.SelectObject(&SelectedBrush);
	CPen   *pOldPen   = dc.SelectObject(&SelectedBlue);

	dc.Rectangle(Recto);

	dc.SelectObje ct(pOldBrush);
	// Do not call CView::OnPaint() for painting messages
}
```

Once the control is already positioned on the client area, to get its location and dimensions, you can call the **CWnd::GetWindowRect()** method. Here is an example:

```
void CTabDlg::OnBtnInfo()
{
	// TODO: Add your control notification handler code here
	CRect Recto;
	char LocDim[80];

	m_Panel.GetWindowRect(&Recto);

	sprintf(LocDim, " - Panel Information -\nLeft:   %d,"
	                "\nTop:    %d,\nWidth:  %d,\nHeight: %d",
	                Recto.left, Recto.top, Recto.Width(), Recto.Height());

	MessageBox(LocDim);
}
```



```
 - Panel Information -
Left:   199,
Top:    347,
Width:  168,
Height: 91
```

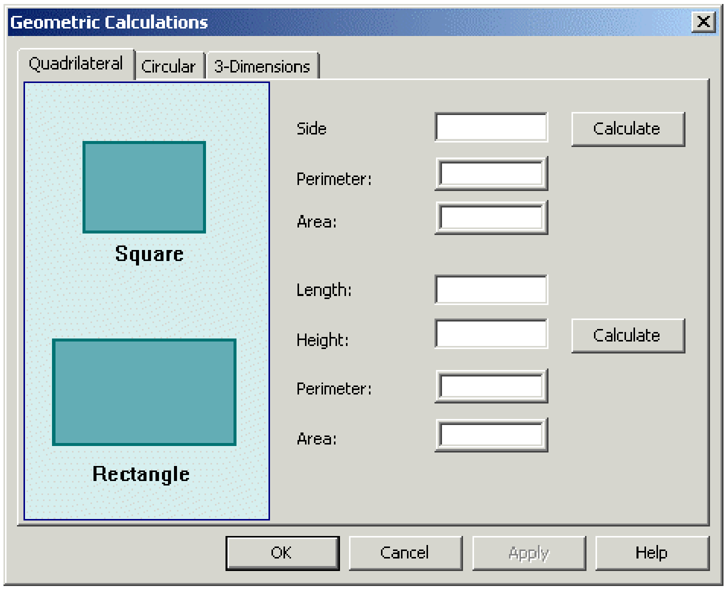
Practical Learning: Using the Client Area

1. Open the Geometry application you were working on earlier
2. Open the Quadrilateral.cpp source file and change its OnPaint() event as follows:

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

    // Create a light green brush
    CBrush BrushLightGreen(RGB(212, 235, 235));
    // Create a navy pen
    CPen   PenNavy(PS_SOLID, 1, RGB(0, 0, 128));
    // Create a green brush
    CBrush BrushGreen(RGB(100, 175, 180));
    // Create a dark green pen
    CPen PenGreen(PS_SOLID, 2, RGB(0, 115, 115));

    // Get the location and dimensions of the client rectangle
    GetClientRect(&Recto);

    // Select the light green brush
    CBrush *pOldBrush = dc.SelectObject(&BrushLightGreen);
    // Select the navy pen
    CPen *pOldPen = dc.SelectObject(&PenNavy);

    // Draw a rectangular shape on the left side of the property page
    dc.Rectangle(0, 0, 162, Recto.Height());

    // Select the green brush
    pOldBrush = dc.SelectObject(&BrushGreen);
    // Select the dark green pen
    pOldPen = dc.SelectObject(&PenGreen);

    // Draw the square
    dc.Rectangle(40, 40, 120, 100);
    // Draw the rectangle
    dc.Rectangle(20, 170, 140, 240);

    // Set the back mode to transparent for the text
    dc.SetBkMode(TRANSPARENT);
    // Display indicative labels
    dc.TextOut(60, 105, "Square");
    dc.TextOut(45, 250, "Rectangle");

    // Restore the old GDI tools
    dc.SelectObject(pOldPen);
    dc.SelectObject(pOldBrush);

    if (IsIconic())
    {

            SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

            // Center icon in client rectangle
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;
```

```
                // Draw the icon
                dc.DrawIcon(x, y, m_hIcon);
        }
        else
        {
                CPropertyPage::OnPaint();
        }
}
```

3. Test the application:



4. Close the application and return to MSVC

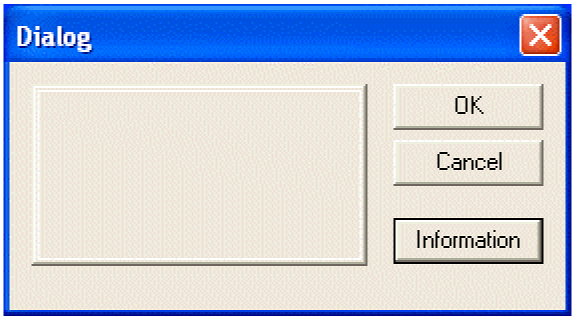
14.2.3 The Screen and Client Coordinates

When calling either the **GetClientRect()** or the **GetWindowRect()** methods to get the location and the dimensions of a control or another object, it is important to know the origin of the produced rectangle. By default, the rectangle returned by the **GetWindowRect()** method called by a control has its origin on the top left corner of the monitor and not on the top-left corner of the parent window. Consider the following event. It gets the location and dimensions of a control and stores them in a **CRect** variable. Then it paints a rectangle (it is supposed to paint the control) located on, and equal to the dimensions of, the control:

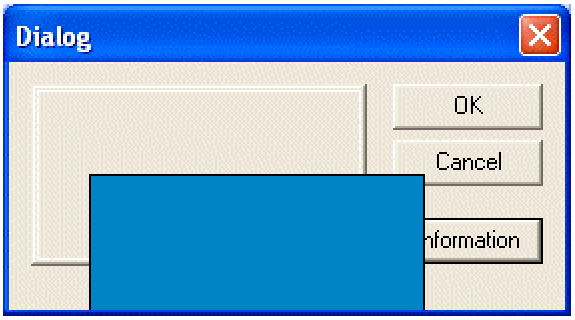
```
void CTabDlg::OnBtnInfo()
{
        // TODO: Add your control notification handler code here
        CRect Recto;

        m_Panel.GetWindowRect(&Recto);

        CClientDC dc(this);
        CBrush BlueBrush(RGB(0, 128, 192));

        CBrush *pOldBrush = dc.SelectObject(&BlueBrush);
        dc.Rectangle(Recto);

```

```
	dc.SelectObject(pOldBrush);
}
```

After executing the program and moving the dialog box somewhere to the middle center of the screen and clicking the button, the result is as follows:



After moving the dialog box close to the top-left section of the screen and clicking the button again, the result is the following:



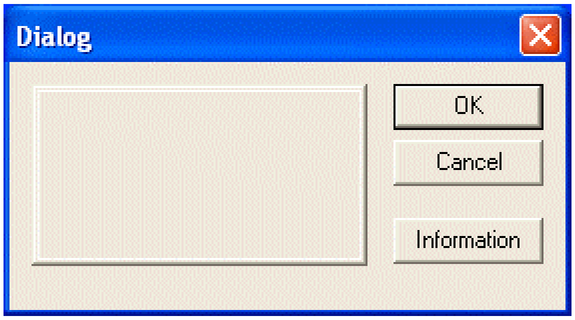
This demonstrates that, although the control is a child of the dialog box, the rectangle returned by the **GetWindowRect()** method is based on the screen and not the client coordinates of the parent window. This is not an anomaly. It is purposely done so you can specify what origin you want to consider.

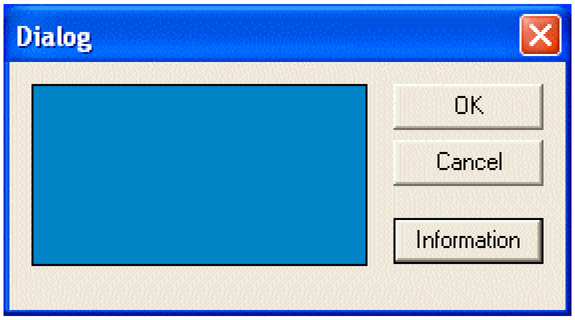
As seen in previous lessons, the origin of the screen is positioned on the top-left corner of the monitor. This is referred to as, or is said that the location uses, screen coordinates. The origin of a client area is placed on its top-left corner. This is referred to as, or is said that the location uses, client coordinates. For example, the origin used by the above **GetWindowRect()** method is based on the screen. If you want the rectangle resulting from a call to either the **GetClientRect()** or the **GetWindowRect()** methods to be based on the client area (on client coordinates) of the control that called it, you can transfer the origin from the screen to the client. This is conveniently done with a call to the **CWnd::ClientToScreen()** method. It is overloaded as follows:

```
void ClientToScreen(LPPOINT lpPoint) const;
void ClientToScreen(LPRECT lpRect) const;
```

If the location you had requested is a point, pass its **POINT** or its **CPoint** variable to the **ClientToScreen()** method. If the value you requested is a rectangle, pass its **RECT** or its **CRect** variable. Here is an example:

```
void CTabDlg::OnBtnInfo()
{
	// TODO: Add your control notification handler code here
	CRect Recto;

	m_Panel.GetWindowRect(&Recto);

	CClientDC dc(this);
	CBrush BlueBrush(RGB(0, 128, 192));

	CBrush *pOldBrush = dc.SelectObject(&BlueBrush);

	ScreenToClient(Recto);
	dc.Rectangle(Recto);

	dc.SelectObject(pOldBrush);
}
```





This time, even if the dialog box moves, the **GetWindowRect()** method returns the same rectangle.

If the location and/or dimension are given in client coordinates, to convert them to screen coordinates, call the **ScreenToClient()** method. It is overloaded as follows:

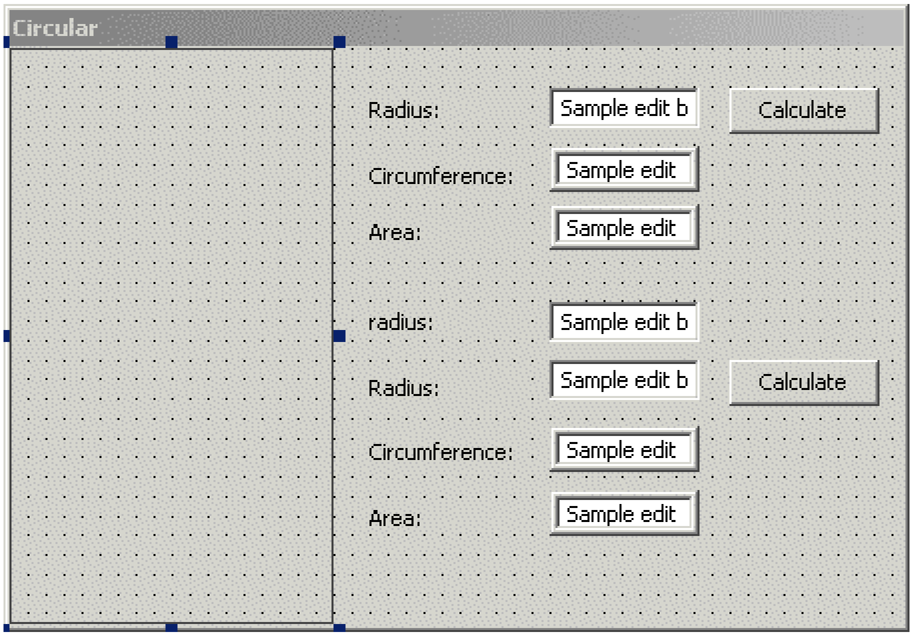
```
void ScreenToClient(LPPOINT lpPoint) const;
void ScreenToClient(LPRECT lpRect) const;
```

This method follows the opposite logic of the **ClientToScreen()** method.

Practical Learning: Using Client and Screen Coordinates

1. The Geometry application should still be opened.
   From the Resource View tab open the IDD_CIRCULAR dialog box

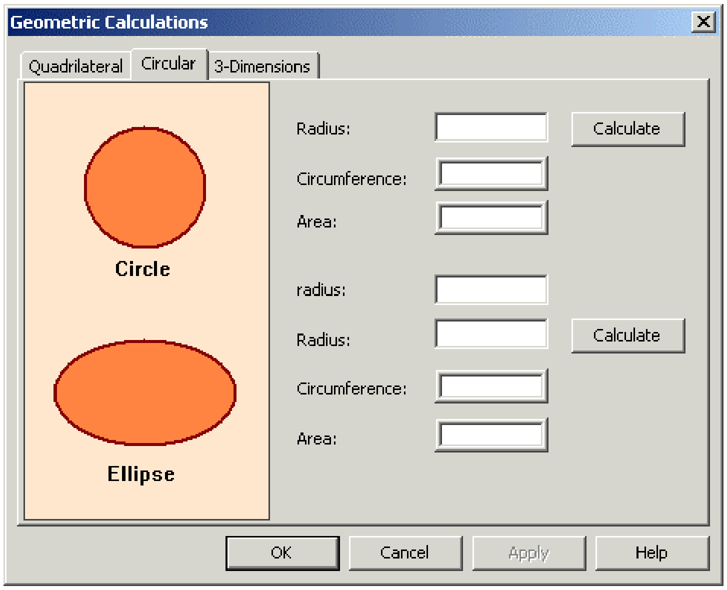
2. On the Controls toolbox, click the Picture control and draw a rectangular shape on the left side of the dialog box



3. Change the ID of the new control to **IDC_SHP_CIRCULAR** and Add a Control Variable for it named **m_ShapeCircular**
4. Access the OnPaint event of the CCircular class and implement it as follows:

```
void CCircular::OnPaint()
{
	CPaintDC dc(this); // device context for painting
	// TODO: Add your message handler code here
	// Do not call CPropertyPage::OnPaint() for painting messages

	// The location and dimension variable for the control
	CRect ShapeRect;
	// Create a cream color brush
	CBrush BrushCream(RGB(255, 230, 205));
	// Create an orange brush
	CBrush BrushOrange(RGB(255, 128, 64));
	// Create a brown pen
	CPen   PenBrown(PS_SOLID, 2, RGB(128, 0, 0));

	// Get the location and dimension of the control
	m_ShapeCirc.GetWindowRect(&ShapeRect);

	// Convert the location and dimensions to client coordinates
	ScreenToClient(&ShapeRect);

	// Select the cream brush to use
	CBrush *pOldBrush = dc.SelectObject(&BrushCream);
	// Paint the control's background to light blue
	dc.Rectangle(ShapeRect);

	// Select the brown pen to paint
	CPen *pOldPen = dc.SelectObject(&PenBrown);
	// Select an orange brush
	pOldBrush = dc.SelectObject(&BrushOrange);

	// Draw the circle
	dc.Ellipse(40, 30, 120, 110);
	// Draw the ellipse
	dc.Ellipse(20, 170, 140, 240);

	// Set the back mode to transparent for the text
	dc.SetBkMode(TRANSPARENT);
	// Display indicative labels
	dc.TextOut(60, 115, "Circle");
	dc.TextOut(55, 250, "Ellipse");

	// Dismiss the GDI objects and restore the originals
	dc.SelectObject(pOldBrush);
	dc.SelectObject(pOldPen);
}
```

5. Test the application



6. Close it and return to MSVC

14.2.4 The Window: Its Location and Dimensions

We have reviewed various ways of specifying a control's bcation and its dimensions, eitther at design or run time. Once a window or a control has been positioned on the screen or in its confined client area, it keeps these attributes until specified otherwise. When dealing with a main window, such as the frame of an application, a dialog box, a property sheet, or a wizard, the user can move it around the screen as necessary and if possible. This is usually done by dragging the title bar.

When the user grabs the title bar of a window and starts dragging to move it, the window sends the **WM_MOVING** message as we saw in Lesson 4. The **WM_MOVING** event fires the **OnMoving()** event. This event is usually left alone as it allows the user to use an application as regularly as possible. The syntax of the **OnMoving()** event is:

```
afx_msg void OnMoving(UINT nSide, LPRECT lpRect);
```

The OnMoving() event fires while the window is being moved. The *nSide* argument specifies the side of window that is moving. As the window is moving, this event returns its location and dimensions as the values of the *lpRect* member variables.

If you create a certain type of window and you do not want the user to move it around, you can write code for the **WM_MOVING** message. In the following example, the user cannot move the window as its location and dimensions are restored with any attempt to move it (if you want to write the OnMoving event for a dialog box in MSVC 6, you may have to manually declare and define the event as follows):

```
class CTabDlg : public CDialog
{
// Construction
public:
	CTabDlg(CWnd* pParent = NULL);   // standard constructor

. . .

// Implementation
protected:

	// Generated message map functions
	//{{AFX_MSG(CTabDlg)
	virtual BOOL OnInitDialog();
	afx_msg void OnMoving(UINT nSide, LPRECT lpRect);
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

. . .

BEGIN_MESSAGE_MAP(CTabDlg, CDialog)
	//{{AFX_MSG_MAP(CTabDlg)
	ON_WM_MOVING()
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()
. . .
void CTabDlg::OnMoving(UINT nSide, LPRECT lpRect)
{
	CRect CurRect;

	// Find out the location and the dimensions of the window
	GetWindowRect(&CurRect);

	// You ain't moving nothin'
	lpRect->left = CurRect.left;
	lpRect->top  = CurRect.top;
	lpRect->right = CurRect.right;
	lpRect->bottom = CurRect.bottom;
}
```

To programmatically move a window, call the **CWnd::SetWindowPos()** method. Its syntax is:

```
BOOL SetWindowPos(const CWnd* pWndInsertAfter,
                  int x, int y, int cx, int cy, UINT nFlags);
```

The *pWndInsertAfter* argument is used to specify the window that will positioned in the Z coordinate on top of the window that called this method. If you have the class name or the CWnd name of the other window, pass it as the *pWndInsertAfter* argument. Otherwise, this argument can have one of the following values:

Value	Description
wndBottom	This window will be positioned under all the other windows, unless it is already at the bottom. If this window is a topmost window, it will not be anymore
wndTop	This window will be positioned on top of all the other windows, unless it is already on top
wndTopMost	This window becomes positioned on top of all other window as if it were created with the **WS_EX_TOPMOST** extended style. In other words, even if its parent window is sent under other window, this particular one stays on top.
wndNoTopMost	If this window is not a top most window, it becomes positioned on top of all other windows, except the window that is top most. If this window was top most when it called this method, it is not top most anymore. If there is another top most window on the screen, that one becomes top most but this one becomes positioned under that one.

If you are not trying to reposition the window in the Z coordinate, pass this argument as NULL or include the SWP_NOZORDER value for the *nFlags* argument.

The *nFlags* argument is used to define how the location arguments (x and y) and the dimensions (cx and cy) will be dealt with. These other arguments have the following roles:

Argument	Description	The argument is ignored if nFlags has the following value
x	This specifies the new distance from the left border of the parent to the left border of this window. This depends on the type of window and the type of parent.	SWP_NOMOVE
y	This specifies the new distance from the top border of the parent to the top border of this window. This depends on the type of window and the type of parent.	SWP_NOMOVE
cx	This is the new width of this window	SWP_NOSIZE
cy	The argument is the new height of this window	SWP_NOSIZE

Additionally, the *nFlags* argument can have one of the following values or one of the above nFlags values can be combined with the following values:

Value	Description
SWP_DRAWFRAME	Draws a frame around the window
SWP_FRAMECHANGED	This value sends a WM_NCCALCSIZE message to the window
SWP_HIDEWINDOW	Hides this window
SWP_NOACTIVATE	If the *pWndInsertAfter* value specified that the window should be reposositioned and activated, which is done if the window is to be positioned on top of another, this

	value lets the *pWndInsertAfter* be performed but the window will not be activated
SWP_NOCOPYBITS	Normally, after a window has been repositioned, its controls are restored to their corresponding relative locations and dimensions. It you want this validation to be ignored, pass this value
SWP_NOOWNERZORDER SWP_NOREPOSITION	If this value is passed, this method will not reposition the windows in the z coordinate
SWP_NOREDRAW	When this value is set, the client area of the window will not be redrawn
SWP_NOSENDCHANGING	When this value is set, the window cannot receive a WM_WINDOWPOSCHANGING message
SWP_SHOWWINDOW	Displays the window

In the following example, a window named m_Panel is repositioned and resized:

```
void CTestDialog::OnBtnMovePanel()
{
        // TODO: Add your control notification handler code here
        m_Panel.SetWindowPos(NULL, 40, 72, 100, 86, SWP_NOZORDER);
}
```

14.2.5 The Handle or Pointer to a Window

Once a control has been created, its identifier set, its location and its dimensions specified, you and your users can exploit it. On one hand, the user can type a value, select text, scroll or control or click something. One of your jobs as a programmer is to predict as many actions as the user may want to perform on your control(s) and take appropriate actions. We have learned that one good way you can refer to a control in your code consists of first providing it with an identifier. Another prerequisite you can take is to declare and associate a control and/or a value variable for your control. Sometimes you will not have declared a control variable for a control but at one time you need to refer to it. One way you can do this is to use the control's identifier and cast it to its corresponding class. This can be taken care of by calling the **CWnd::GetDlgItem()** method. It comes in two versions as follows:

```
CWnd* GetDlgItem(int nID) const;
void CWnd::GetDlgItem(int nID, HWND* phWnd) const;
```

By providing the *nID* argument as the identifier of the control to this method, you can get a pointer to its class. To do this, you can declare a pointer to the class of the control, then call the **GetDlgItem()** method. Because GetDlgItem() returns a CWnd pointer, using the features of inheritance, cast this return value to the class of the control.

Practical Learning: Accessing a Window's Handle

1. The Geometry application should still be opened.
   Open the OnClickedBnUcalc() event of the CGeom3D class
2. To get handles to the edit boxes on the dialog, implement the event as follows:

```
void CGeome3D::OnBnClickedBtnUcalc()
{
    // TODO: Add your control notification handler code here
    // Related Calculations of the cube
```

```
	CEdit *edtCubeSide, *edtCubeArea, *edtCubeVolume;

	edtCubeSide   = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_USIDE));
	edtCubeArea   = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_UAREA));
	edtCubeVolume = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_UVOL));
}
```

3. Change the content of theOnBnClickedBcalc event as follows:

```
void CGeome3D::OnBnClickedBtnBcalc()
{
	// TODO: Add your control notification handler code here

	// Related Calculations of the box
	CEdit *edtBoxLength, *edtBoxWidth, *edtBoxHeight,
	      *edtBoxArea, *edtBoxVolume;

	edtBoxLength = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BLENGTH));
	edtBoxWidth  = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BHEIGHT));
	edtBoxHeight = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BWIDTH));
	edtBoxArea   = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BAREA));
	edtBoxVolume = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BVOL));
}
```

4. Save All

14.2.6 The Text of a Control

For you the programmer, the control identifier may be one of the most important properties of a window. For the user, this is not the case. For a text-based control, the most important part, as far as the user is concerned, may be its text. For example, if the user is filling an employment application, the text entered on the fields is what would make the difference. Many controls use text. In fact, one of the most obvious items on most windows such as frames or dialog-based objects is the text they display. This text allows the user to identify a window or an object on the screen.

Some controls only display text that the user can/must read in order to use an application. Some other controls allow the user to change their text. Regardless of what such text is used for, you should exercise a good deal of control on the text that a control would display or receive.

When we started reviewing controls, we saw that some of the controls that use text would allow you to change the Caption property at design time. On the other hand, while a using is interacting with your application, depending on various circumstances, at a certain time you may want to change the text that a window or control is displaying or holding; that is, if the control is meant to display text. Changing the text of a window or a control can be taken care of by calling the **CWnd::SetWindowText()** method. Its syntax is:

```
void SetWindowText(LPCTSTR lpszString);
```

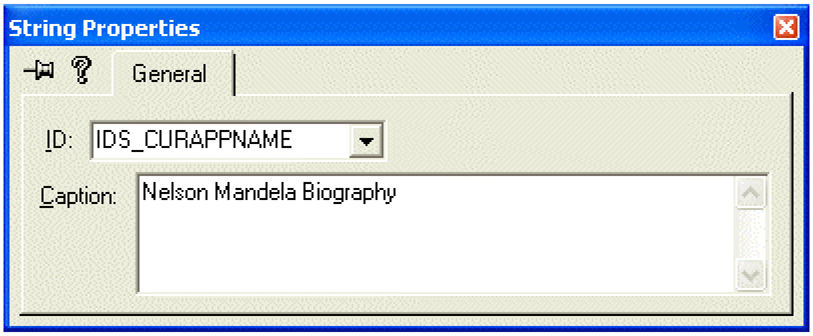
The *lpszString* argument is a null-terminated string that holds the value you want to display. It can be configured using any of the valid null-terminated string operations available. Here is an example that changes the title of a dialog box when the window displays. The text is provided as a null-terminated string passed to the method:

```
BOOL CDismissDlg::OnInitDialog()
{
```

```
            CDialog::OnInitDialog();

            // TODO: Add extra initialization here
            SetWindowText("Windows Fundamentals");

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

Another technique you can use consist of first declaring a null-terminated string variable, assign it a value, and then pass it the *lpszString* argument to the **SetWindowText()** function.

If you are using resources in your MFC application, you can also create a global value in the string table to be used as the window name:



ID: IDS_CURAPPNAME

Caption: Nelson Mandela Biography

You can call the string of such an identifier, store it in a **CString** variable, and then pass it to the **CWnd::SetWindowText()** method. Here is an example:

```
CMainFrame::CMainFrame()
{
            // Declare a window class variable
            WNDCLASS WndCls;
            const char *StrWndName = "Application Name";

            . . .

            const char *StrClass = AfxRegisterWndClass(WndCls.style, WndCls.hCursor,
                                    WndCls.hbrBackground, WndCls.hIcon);

            Create(StrClass, StrWndName);

            CString Str;
            Str.LoadString(IDS_CURAPPNAME);
            SetWindowText(Str);
}
```

To change the name of a window, instead of calling **SetWindowText(),** you can call the **CWnd::SendMessage()** method. Since you want to change the text, the message argument must be **WM_SETTEXT**. The *wParam* argument is not used. The lParam argument holds the string that will be the new value of the window name. You must cast the string to **LPARAM**. Here is an example that allows the user to click a menu item that changes the title of the frame window:

```
void CMainFrame::OnEditChangeTitle()
```

```
{
	// TODO: Add your command handler code here

	char NewTitle[] = "Introduction to Windows Programming";

	SendMessage(WM_SETTEXT, NULL, reinterpret_cast<LPARAM>(NewTitle));
}
```

To retrieve the name of a window (always remember that the name of a window is not the name of a class) or the text stored in a control, you can call the **CWnd::GetWindowText()** function. Its syntax is:

```
int GetWindowText(LPTSTR lpszStringBuf, int nMaxCount) const;
```

The *lpszStringBuf* is the null-terminated string that will store the window name. The *nMaxCount* is the minimum number of characters of the *lpszStringBuf*. If you specify more characters than the name is made of, the compiler would reduce this number to the actual length of the string. Therefore, it is safe to provide a high number.

Practical Learning: Changing a Control's Text

1. The Geometry application should still be opened.
   To perform window text operations, change the Quadrilateral.cpp source file as follows:

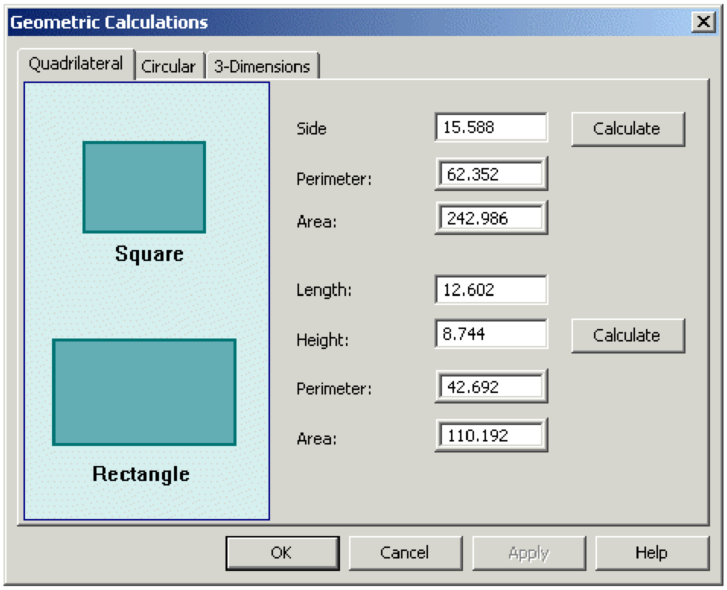
```
// Quadrilateral.cpp : implementation file
//

. . .

// CQuadrilateral message handlers

BOOL CQuadrilateral::OnInitDialog()
{
	CPropertyPage::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_SquareSide.SetWindowText("0.00");
	m_SquarePerimeter.SetWindowText("0.00");
	m_SquareArea.SetWindowText("0.00");
	m_RectLength.SetWindowText("0.00");
	m_RectHeight.SetWindowText("0.00");
	m_RectPerimeter.SetWindowText("0.00");
	m_RectArea.SetWindowText("0.00");

	return TRUE;  // return TRUE  unless you set the focus to a control
}

. . .

void CQuadrilateral::OnBnClickedBtnScalc()
{
    // TODO: Add your control notification handler code here
    double SquareSide, SquarePerimeter, SquareArea;
    char   StrSquareSide[10], StrSquarePerimeter[10], StrSquareArea[10];

    m_SquareSide.GetWindowText(StrSquareSide, 10);
    SquareSide = atof(StrSquareSide);
    SquarePerimeter = SquareSide * 4;
    SquareArea = SquareSide * SquareSide;

    sprintf(StrSquarePerimeter, "%.3f", SquarePerimeter);
    sprintf(StrSquareArea, "%.3f", SquareArea);

    m_SquarePerimeter.SetWindowText(StrSquarePerimeter);
    m_SquareArea.SetWindowText(StrSquareArea);
}

void CQuadrilateral::OnBnClickedBtnRcalc()
{
    // TODO: Add your control notification handler code here

    double RectLength, RectHeight, RectPerimeter, RectArea;
    char   StrRectLength[10], StrRectHeight[10],
                StrRectPerimeter[10], StrRectArea[10];

    m_RectLength.GetWindowText(StrRectLength, 10);
    RectLength = atof(StrRectLength);
    m_RectHeight.GetWindowText(StrRectHeight, 10);
    RectHeight = atof(StrRectHeight);

    RectPerimeter = 2 * (RectLength + RectHeight);
    RectArea = RectLength * RectHeight;

    sprintf(StrRectPerimeter, "%.3f", RectPerimeter);
    sprintf(StrRectArea, "%.3f", RectArea);

    m_RectPerimeter.SetWindowText(StrRectPerimeter);
    m_RectArea.SetWindowText(StrRectArea);
}
```

2. Completer the controls events of the Geome3D.cpp source as follows:

```
// Geome3D.cpp : implementation file
//

. . .

// CGeome3D message handlers
void CGeome3D::OnBnClickedBtnUcalc()
{
    // TODO: Add your control notification handler code here
    // Related Calculations of the cube
    double CubeSide, CubeArea, CubeVolume;
    CEdit *edtCubeSide, *edtCubeArea, *edtCubeVolume;
    char StrCubeSide[10], StrCubeArea[10], StrCubeVolume[10];

    edtCubeSide   = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_USIDE));
    edtCubeArea   = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_UAREA));
    edtCubeVolume = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_UVOL));

    edtCubeSide->GetWindowText(StrCubeSide, 10);
```

```
	CubeSide = atof(StrCubeSide);

	CubeArea = CubeSide * 6;
	CubeVolume = CubeSide * CubeSide * CubeSide;

	sprintf(StrCubeArea, "%.3f", CubeArea);
	sprintf(StrCubeVolume, "%.3f", CubeVolume);

	edtCubeArea->SetWindowText(StrCubeArea);
	edtCubeVolume->SetWindowText(StrCubeVolume);
}

void CGeome3D::OnBnClickedBtnBcalc()
{
	// TODO: Add your control notification handler code here

	// Related Calculations of the box
	double BoxLength, BoxWidth, BoxHeight, BoxArea, BoxVolume;
	CEdit *edtBoxLength, *edtBoxWidth, *edtBoxHeight,
	          *edtBoxArea, *edtBoxVolume;
	char StrLength[10], StrWidth[10], StrHeight[10],
	     StrArea[10], StrVolume[10];

	edtBoxLength = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BLENGTH));
	edtBoxWidth  = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BHEIGHT));
	edtBoxHeight = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BWIDTH));
	edtBoxArea   = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BAREA));
	edtBoxVolume = reinterpret_cast<CEdit *>(GetDlgItem(IDC_EDT_BVOL));

	edtBoxLength->GetWindowText(StrLength, 10);
	edtBoxWidth->GetWindowText(StrWidth, 10);
	edtBoxHeight->GetWindowText(StrHeight, 10);

	BoxLength = atof(StrLength);
	BoxWidth  = atof(StrWidth);
	BoxHeight = atof(StrHeight);

	BoxArea   = 2 * ((BoxLength + BoxWidth) +
	                 (BoxLength + BoxHeight) +
	                 (BoxWidth  + BoxHeight));
	BoxVolume = BoxLength * BoxWidth * BoxHeight;

	sprintf(StrArea, "%.3f", BoxArea);
	sprintf(StrVolume, "%.3f", BoxVolume);

	edtBoxArea->SetWindowText(StrArea);
	edtBoxVolume->SetWindowText(StrVolume);
}
```

3. Execute the application. Test the controls in the Quadrilateral property page by providing numeric values in the Side, the Length, and the Height edit boxes before clicking their corresponding button



Side: 15.588
Perimeter: 62.352
Area: 242.986
Length: 12.602
Height: 8.744
Perimeter: 42.692
Area: 110.192

4. Also test the calculations of the 3-Dimensions property page
5. After using it, close the application and return to MSVC

14.2.7 Controls Values Update

The controls used in your application are designed to work as an ensemble, exchanging data with one another and with the window that hosts them. For their various operations to work, the dialog-based object that is hosting the controls needs to be able to update their values. If you are using control variables, because these controls are based on **CWnd**, they can perform their own validations.

If you are using value controls, and if the user must be able to change the values held by a dialog's controls, you can decide how and when the values should be updated by the parent window. This can be done by calling the **CWnd::UpdateData()** method. Its syntax is:

```
BOOL UpdateData(BOOL bSaveAndValidate = TRUE);
```

The *bSaveAndValidate* argument specifies whether the parent window, usually a dialog-based object, must update the values of variables at the time this method is called. This member variable works in conformance with the **CDialog::DoDataExchange()** event of the dialog that owns the controls. If it is called with no argument or the TRUE default value, this indicates that the dialog is ready to communicate with the variables mapped in **DoDataExchange().** After such a call, you can let the user do what is necessary on the controls. When this method is called a FALSE value for the *bSaveAndValidate* argument, it indicates that the dialog box can take ownership of operations and the controls have stopped updating their information.You can handle this when the user clicks a button such as Submit or OK after changing values on controls. Normally, the call with a FALSE argument means that the dialog box is being either initialized or reinitialized, which is done when its **OnInitDialog()** event fires.

Practical Learning: Updating Controls Data

1. To see examples of updating data, implement the events of the Circular.cpp source file as follows:

```
// Circular.cpp : implementation file
//

#include "stdafx.h"
#include "Geometry1.h"
#include "Circular.h"

const double PIValue = 3.14159;
// CCircular dialog

IMPLEMENT_DYNAMIC(CCircular, CPropertyPage)
CCircular::CCircular()
      : CPropertyPage(CCircular::IDD)
      , m_szCircleRadius(_T("0.00"))
      , m_szCircleCircumference(_T("0.00"))
      , m_szCircleArea(_T("0.00"))
      , m_szEllipseradius(_T("0.00"))
      , m_szEllipseRadius(_T("0.00"))
      , m_szEllipseCircumference(_T("0.00"))
      , m_szEllipseArea(_T("0.00"))
{
}

CCircular::~CCircular()
{
}

void CCircular::DoDataExchange(CDataExchange* pDX)
{
      CPropertyPage::DoDataExchange(pDX);
      DDX_Text(pDX, IDC_EDT_CRADIUS, m_szCircleRadius);
      DDX_Text(pDX, IDC_EDT_CCIRC, m_szCircleCircumference);
      DDX_Text(pDX, IDC_EDT_CAREA , m_szCircleArea);
      DDX_Text(pDX, IDC_EDT_VRADIUS, m_szEllipseradius);
      DDX_Text(pDX, IDC_EDT_HRADIUS, m_szEllipseRadius);
      DDX_Text(pDX, IDC_EDT_CCIRC2, m_szEllipseCircumference);
      DDX_Text(pDX, IDC_EDT_EAREA, m_szEllipseArea);
}

BEGIN_MESSAGE_MAP(CCircular, CPropertyPage)
      ON_BN_CLICKED(IDC_BTN_CCALC, OnBnClickedBtnCcalc)
      ON_BN_CLICKED(IDC_BTN_ECALC, OnBnClickedBtnEcalc)
END_MESSAGE_MAP()

// CCircular message handlers

void CCircular::OnBnClickedBtnCcalc()
{
      // TODO: Add your control notification handler code here
      UpdateData();

      double Radius, Circumference, Area;

      Radius = atof(m_szCircleRadius);
```

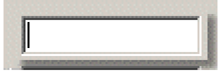
```
	Circumference = Radius * 2 * PIValue;
	Area = Radius * Radius * PIValue;

	m_szCircleCircumference.Format("%.3f", Circumference);
	m_szCircleArea.Format("%.3f", Area);

	UpdateData(FALSE);
}

void CCircular::OnBnClickedBtnEcalc()
{
	// TODO: Add your control notification handler code here
	UpdateData();

	double radius, Radius, Circumference, Area;

	radius = atof(m_szEllipseradius);
	Radius = atof(m_szEllipseRadius);

	Circumference = (radius + Radius) * PIValue;
	Area = radius * Radius * PIValue;

	m_szEllipseCircumference.Format("%.3f", Circumference);
	m_szEllipseArea.Format("%.3f", Area);

	UpdateData(FALSE);
}
```

2. Test the application by changing the values of the Circular property pages
3. After using it, close the application and return to MSVC

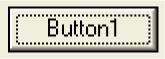
14.2.8 Window's Focus

A control is said to have focus if it is ready to receive input from the user. For example, if a text control, such as an edit box, has focus and the user presses a character key, the corresponding character would be displayed in the control.

Controls show different ways of having focus. For example, when an edit box has focus, a caret is blinking in it:



When a button has focus, it displays a dotted rectangle around its caption:



There are two main ways a control receives focus: based on a user's action or an explicit request from you. To give focus to a control, the user usually presses Tab, which allows navigating from one control to another. To programmatically give focus to a control, call the **CWnd::SetFocus()** method.

```
CWnd* SetFocus( );
```

This method gives focus to the control that called it. In the following example, an edit box identified as IDC_EDIT1 will receive focus when the user clicks the button:

```
void CFormView1View::OnButton2()
{
	// TODO: Add your control notification handler code here
	CButton *btnFirst;

	btnFirst = (CButton *)GetDlgItem(IDC_EDIT1);
	btnFirst->SetFocus();
}
```

Once a control receives focus, it initiates a **WM_SETFOCUS** message, which fires an **OnSetFocus()** event. The syntax of the **CWnd::OnSetFocus()** event is:

```
afx_msg void OnSetFocus( CWnd* pOldWnd );
```

You can use this event to take action when, or just before, the control receives focus. In the following example, when an edit box receives focus, a message box is displayed:

```
void CFormView1View::OnSetFocusEdit3()
{
	// TODO: Add your control notification handler code here
	MessageBox("The Result edit box should not receive focus!!!");
}
```

At anytime, to find out what control has focus, call the **CWnd::GetFocus()** method. Its syntax is:

```
static CWnd* PASCAL GetFocus();
```

This method returns a handle to the control that has focus at the time the method is called. While the user is interracting with your application, the focus changes constantly. For this reason, you should avoid using the return type of this method from various events or member functions. In other words, do not globally declare a CWnd variable or pointer, find out what control has focus in an event Event1 and use the returned value in another event Event2 because, by the time you get to Event2, the control that had focus in Event1 may have lost focus. In fact, the dialog box that holds the control or the main application may have lost focus. Therefore, use the **GetFocus()** method only locally.

14.2.9 The Window's Visibility

After a window or a control has been created, for the user to take advantage of it, it must be made visible. As we will learn in other lessons, when it comes to their visibility, there are two types of windows: those the user can see and interact with, and those invisible conttrols that work only behind the scenes and cannot be displayed to the user.

During control design and when we reviewed their styles, we saw that a window can be made displayed to the user by setting its **Visible** property to True or by adding it the **WS_VISIBLE** style.

If you did not set the **Visible** property to True or did not add the **WS_VISIBLE** style, the control would be hidden (but possibly available). Therefore, if at any time a window is hidden, you can display it by calling the **CWnd::ShowWindow()** method. Its syntax is:

```
BOOL ShowWindow(int nCmdShow );
```

This method is used to display or hide any window that is a descendent of **CWnd**. Its argument, *nCmdShow*, specifies what to do with the appearance or disappearance of the object. Its possible values are:

Value	Description
SW_SHOW	Displays a window and makes it visible
SW_SHOWNORMAL	Displays the window in its regular size. In most circumstances, the operating system keeps track of the last location and size a window such as Internet Explorer or My Computer had the last time it was displaying. This value allows the OS to restore it
SW_SHOWMINIMIZED	Opens the window in its minimized state, representing it as a button on the taskbar
SW_SHOWMAXIMIZED	Opens the window in its maximized state
SW_SHOWMINNOACTIVE	Opens the window but displays only its icon. It does not make it active
SW_SHOWNA	As previous
SW_SHOWNOACTIVATE	Retrieves the window's previous size and location and displays it accordingly
SW_HIDE	Used to hide a window
SW_MINIMIZE	shrinks the window and reduces it to a button on the taskbar
SW_RESTORE	If the window was minimized or maximized, it would be restored to its previous location and size

To use one of these constants, pass it to the **ShowWindow()** method. For example, to minimize a window that is minimizable, you would use code as follows:

```
ShowWindow(SW_SHOWMINIMIZED);
```

Remember that this method is used to either hide or to display a control by passing the appropriate constant, **SW_HIDE** to hide and **SW_SHOW** to display it. Here is an example that displays a control that missed the **WS_VISIBLE** property when it was created:

```
void CSecondDlg::OnFirstControl()
{
	// TODO: Add your control notification handler code here
	CWnd *First = new CWnd;
	CString StrClsName = AfxRegisterWndClass(CS_VREDRAW | CS_HREDRAW,
				LoadCursor(NULL, IDC_CROSS),
					(HBRUSH)GetStockObject(BLACK_BRUSH),
					LoadIcon(NULL, IDI_WARNING));

	First->Create(StrClsName, NULL, WS_CHILD);
	First->ShowWindow(SW_SHOW);
}
```

When the **ShowWindow()** method is called with the **SW_SHOW** value, if the control was hidden, it would become visible; if the control was already visible, nothing would happen. In the same way, when this method is called with the **SW_HIDE** argument, the control would be hidden, whether it was already hidden or not.

If you want to check the visibility of a control before calling the **ShowWindow()** method, you can call the **CWnd::IsWindowVisible()** method. Its syntax is:

```
BOOL IsWindowVisible() const;
```

This method returns TRUE if the control that called it is already visible. If the control is hidden, the method returns FALSE.

14.2.10 The Window's Availability

We saw that when a control has been created, it is available to the user who can interact with its value. This is because a control usually has its **Disable** property to False or unchecked. A control is referred to as disabled if the user can see it but cannot change its value.

If for any reason a control is disabled, to enable it, you can call the **CWnd::EnableWindow()** method. In fact, the **EnableWindow()** method is used either to enable or to disable a window. Its syntax is:

```
BOOL EnableWindow(BOOL bEnable = TRUE);
```

Here is an example that disables a control called Memo:

```
void CSecondDlg::OnDisableMemo()
{
	// TODO: Add your control notification handler code here
	Memo->EnableWindow(FALSE);
}
```

When calling the **EnableWindow()** method, if you pass the **FALSE** value, the control is disabled, whether it was already disabled or not. If you pass the **TRUE** constant, it gets enabled even it was already enabled. Sometimes you may want to check first whether the control is already enabled or disabled. This can be accomplished by calling the **CWnd::IsWindowEnabled().** Its syntax is:

```
BOOL IsWindowEnabled( ) const;
```

This method checks the control that called it. If the control is enabled, the member function returns TRUE. If the control is disabled, this method returns FALSE. Here is an example:

```
void CSecondDlg::OnDisableMemo()
{
	// TODO: Add your control notification handler code here
	if( Memo->IsWindowEnabled() == TRUE )
		Memo->EnableWindow(FALSE);
	else // if( !Memo->IsWindowEnabled() )
		Memo->EnableWindow();
}
```

Here is a simplified version of the above code:

```
void CSecondDlg::OnDisableMemo()
{
	// TODO: Add your control notification handler code here

	Memo->EnableWindow(!Memo->IsWindowEnabled());
}
```

14.3 Access to a Controls Instance and Handle

14.3.1 The Instance of an Application

When you create a Win32 application using the **WinMain()** function, or if you create an MFC application using the **CWinApp** class, when the application comes up, it creates an instance, which is usually the **hInstance** argument of the **WinMain()** function**.** In the same way, when you create an MFC application, which is done using a class based on **CWinApp**, and when the application comes up, it creates an instance. Sometimes you will need to refer to the instance of your application. We have already mentioned that you can do this by calling the **CWinApp::m_hInstance** member variable. Alternatively, for an MFC application, you can call the **AfxGetInstanceHandle()** global function to get a handle to the instance of your application. This could be accessed as follows:

```
BOOL CDialog1Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	HINSTANCE InstanceOfThisApp = AfxGetInstanceHandle();

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

14.3.2 The Handle to a Window

We saw with Win32 applications that, when creating a parent window, if you want to refer to that parent control, you should use the value returned by the **CreateWindow()** or the **CreateWindowEx()** function, which is an **HWND** value. If you are creating an MFC application, you usually call the **Create()** method of the window you are creating. As the parent of all window classes of an MFC application, **CWnd** provides a member variable called **m_hWnd**. It is defined as follows:

```
HWND m_hWnd;
```

This public variable is inherited by all classes that are based on **CWnd**, which includes all MFC window objects. Consequently, **m_hWnd** gives you access to the handle to the window you have created. For example, the **CDialog** class, which is based on **CWnd** but is the most used host of Windows controls, can provide its **m_hWnd** variable as the parent of its control. Here is an example:

```
BOOL CClientAreaDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);  // Set big icon
	SetIcon(m_hIcon, FALSE); // Set small icon

	// TODO: Add extra initialization here

	CWnd* stcLogo = new CWnd;

	stcLogo->CreateEx(WS_EX_DLGMODALFRAME, "STATIC", NULL,
	                  WS_CHILD | WS_VISIBLE | WS_BORDER,
	                  240, 90, 90, 40, m_hWnd, 0x1888);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

In the same way, to get a handle to any control of your application, access its **m_hWnd** member variable.

If you had created a window using the Win32 API's **CreateWindow()** or **CreateWindowEx()** function, or if for any reason an **HWND** object exists in your application, you can convert such a window to a **CWnd** pointer using the **CWnd::FromHandle()** method. Its syntax is:

```
static CWnd* PASCAL FromHandle(HWND hWnd);
```

Here is an example:

```
BOOL CDialog1Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	HWND ThisWnd;

	ThisWnd = CreateWindow("EDIT", NULL,
	          WS_CHILD | WS_VISIBLE | WS_BORDER,
	          20, 100, 140, 200, m_hWnd, NULL, AfxGetInstanceHandle(), NULL);

	CWnd *NewWnd;

	NewWnd->FromHandle(ThisWnd);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

14.4 Getting Access to a Control

14.4.1 Retrieving Control Information

To get information about a control, you can call the **GetWindowLong()** function. Its syntax is:

```
LONG GetWindowLong(HWND hWnd, int nIndex );
```

After this function executes, it returns a **LONG** value. If you prefer to get a pointer and if you are working on a 64-bit environ, use the **GetWindowLongPtr()** function instead. Its syntax is:

```
LONG_PTR GetWindowLongPtr(HWND hWnd, int nIndex);
```

The first argument, *hWnd*, is a handle to the control whose information you are seeking. The *nIndex* argument specifies the type of information you are looking for. Its possible values are:

nIndex Value		Description
GetWindowLong()	GetWindowLongPtr()	
GWL_EXSTYLE		This is used to get information about the extended style(s) used on the control
GWL_STYLE		This is used to get information about the style(s) used on the control
GWL_WNDPROC	GWLP_WNDPROC	Remember that a window procedure is a function pointer. If such a procedure was used to handle the message(s) for the *hWnd* control, use this constant to get the address of that procedure
GWL_HINSTANCE	GWLP_HINSTANCE	This gives access to the handle to the current application
GWL_HWNDPARENT	GWLP_HWNDPARENT	This constant can be used to get a handle to the parent window. For example, you can use it the get a handle to a dialog box that is hosting the *hWnd* control.
GWL_ID	GWLP_ID	This gives you the ID of the *hWnd* control
GWL_USERDATA	GWLP_USERDATA	This gives you a 32-bit value used by the current application

If the *hWnd* argument represents a dialog box, *nIndex* can use the following values:

nIndex Value		Description
GetWindowLong	GetWindowLongPtr	
DWL_DLGPROC	DWLP_DLGPROC	This provides the address of, or a pointer, to the procedure of the dialog box
DWL_MSGRESULT	DWLP_MSGRESULT	This provides the return value of the dialog box' procedure
DWL_USER	DWLP_USER	This provides additional information about the application

After calling this function and specifying the type of information you need, it returns a (constant) value you can use as you see fit. Here are two methods of getting a handle to the instance of the application. The second example uses the **GetWindowLong()** function:

```
BOOL CDialog1Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();
```

```
        // Set the icon for this dialog.  The framework does this automatically
        //  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);                         // Set big icon
        SetIcon(m_hIcon, FALSE);              // Set small icon

        // TODO: Add extra initialization here
        HINSTANCE Instance1 = AfxGetInstanceHandle();
        LONG Instance2 = GetWindowLong(m_hWnd, GWL_HINSTANCE);

        m_Instance1.Format("%ld", Instance1);
        m_Instance2.Format("%ld", Instance2);
        UpdateData(FALSE);

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

14.4.2 Changing Control Information

We have seen how easy it can be to define the properties, styles, and other characteristics when creating a control. While the user is interacting with the computer using your application, you may need to change some of these attributes of a window. To change the value of an aspect of the window, you can call the **SetWindowLong()** function. Its syntax is:

```
LONG SetWindowLong(HWND hWnd, int nIndex, LONG dwNewLong);
```

If you want to work with a pointer instead of a value, you can use the **SetWindowLongPtr()** function. Its syntax is:

```
LONG_PTR SetWindowLongPtr(HWND hWnd, int nIndex, LONG_PTR dwNewLong);
```

The *hWnd* argument is a handle to the window whose informtation you want to change. The value of *nIndex* species the type of information you want to change. It can have the following values:

nIndex Value		Description
GetWindowLong	GetWindowLongPtr	
GWL_EXSTYLE		Allows changing the extended style(s) of the window or control
GWL_STYLE		Allows changing the style(s) of the window or control
GWL_WNDPROC	GWLP_WNDPROC	Allows changing the procedure of the control
GWL_HINSTANCE	GWLP_HINSTANCE	Changes the application instance of the control
GWL_ID	GWLP_ID	Allows changing the ID value of the control
GWL_USERDATA	GWLP_USERDATA	Gives access to a 32-bit value associated with the window, allowing it to be changed

If the *hWnd* argument represents a dialog box, the nIndex argument can also have the following values:

nIndex Value		Description
GetWindowLong	GetWindowLongPtr	
DWL_DLGPROC	DWLP_DLGPROC	Allows changing the procedure of the dialog box
DWL_MSGRESULT	DWLP_MSGRESULT	Allows changing the return value of the procedure

DWL_USER	DWLP_USER	Allows changing additional information

Chapter 15: Fundamental Controls

✍ Static Controls

✍ Animation Controls

✍ Group Boxes

☞ Command Buttons

☞ Property Sheets and Wizard Buttons

☞ Bitmap Buttons

15.1 Static Controls

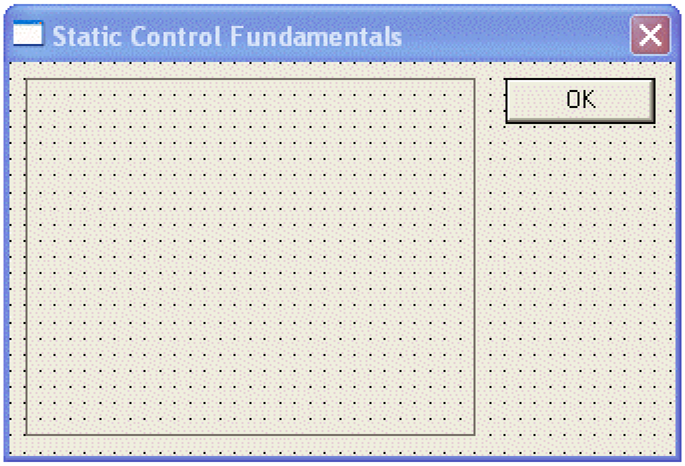
15.1.1 Introduction

A static control is an object that displays information to the user without his or her direct intervention. A static control can be used to display text, a geometric shape, or a picture such as an icon, a bitmap, or an animation. Normally, a user cannot change the value of a static control. For example, if a static control displays text, the user cannot directly change it. In the same way, if a static control is used to show a picture, the user cannot inherently change the picture.

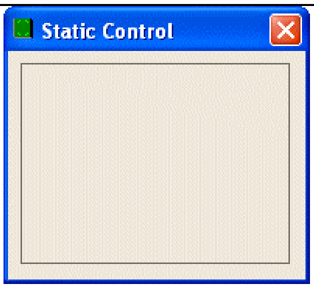
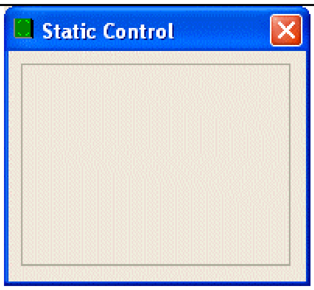
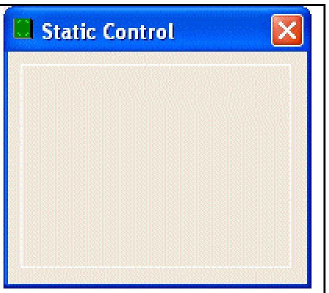
Visual C++ (along with MFC) offers various types of static controls: the Picture control, the Static Text control, and the Group Box control. In the strict sense of the current context, to add a static control to your application, from the Controls toolbox, click the Picture button and click the parent. To programmatically create a static control, declare a pointer to CStatic using the **new** operator. Then call its Create() method to initialize it.

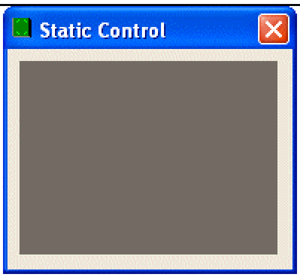
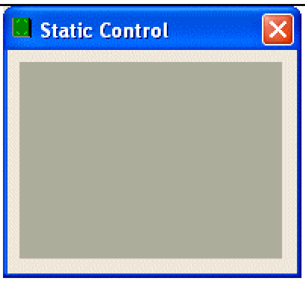
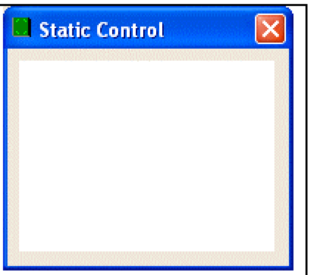
15.1.2 Static Control Properties

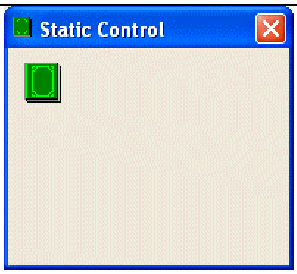
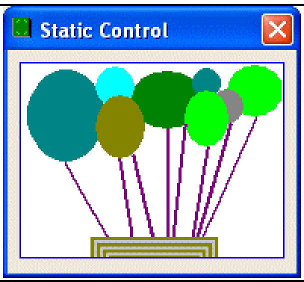
To create a static control, the MFC library provides the CStatic class. Like any other control, there are various ways you can create this control. During design, you can add a Picture control to a form or a dialog box:



The Picture control is the most classic static control of the MFC library. To exploit a static control, you can manipulate some of its characteristics. These characteristics are referred to as its style. The styles applicable on a static control are:

Colored border: You can surround the borders of the control with a black, a gray or a white color. During design, specify the Type as Frame and set the color accordingly using the Color combo box. These characteristics are set as SS_BLACKFRAME, SS_GRAYFRAME, and SS_WHITEFRAME respectively.			

The control is filled with either a black, a gray, or a white color. These characteristics are based on the SS_BLACKRECT, SS_GRAYRECT, and SS_WHITERECT values respectively. At design time, set the Type combo box to Rectangle and select one of these colors in the Color combo box.			

A static control can also be used to display a bitmap. To make this possible, at design time, set the Type to Bitmap and, in the Image combo box, select a bitmap. These properties are controlled through the SS_ICON or SS_BITMAP		

If you want to manipulate the properties of a static control, you should change its identifier from IDC_STATIC to a more meaningful name.

Practical Learning: Using a Static Control

1. Open the Geometry application. If you do not have it, open the Geometry3 application that accompany this book

2. On the main menu, click either Insert -> Resource or Project -> Add Resource…

7. In the Add Resource dialog box, click the Import button.

8. Locate the folder that hold the exercises for this book and display its Pictures folder in the Look In combo box

9. Click Geome3D and click Open

10. In Resource View, and using the Properties window, change the bitmap's ID to IDB_GEOME3D

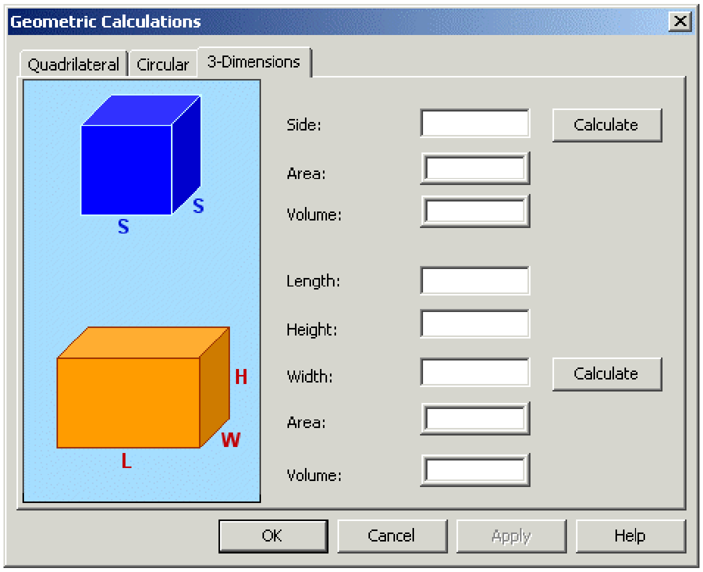
11. Display the IDD_G3D dialog box

    On the Controls toolbox, click the Picture button and click on the top left section of the dialog box

12. On the Properties window, change the **Type** to **Bitmap**

13. Click the arrow of the Image combo box and select IDB_GEOME3D

14. Test the application



15. Close the dialog box and return to MSVC

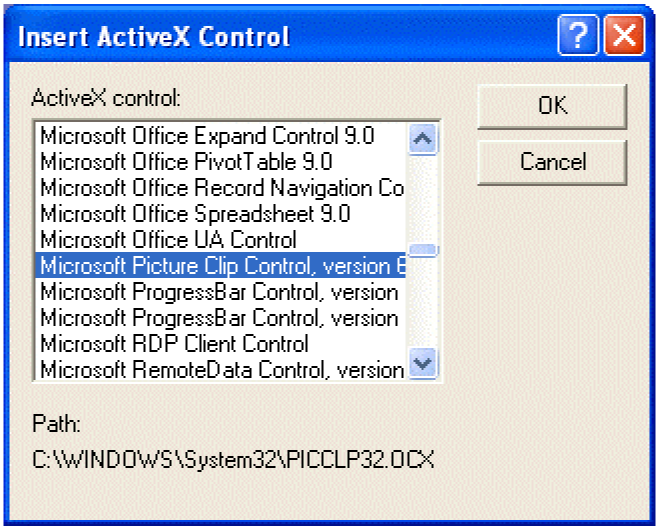
15.1.3 The Picture Control

If you need to display a picture for your application, Visual C++ provides a special control for that purpose. To do this, you can use Microsoft Picture Clip Control available from the Insert ActiveX Control. After adding it to your application, use its Picture property page to specify the picture you want to display.

Practical Learning: Displaying a Picture

Note
Because of the way bitmaps are dealt with in MSVC 7, the following program will not work as expected in MSVC 7

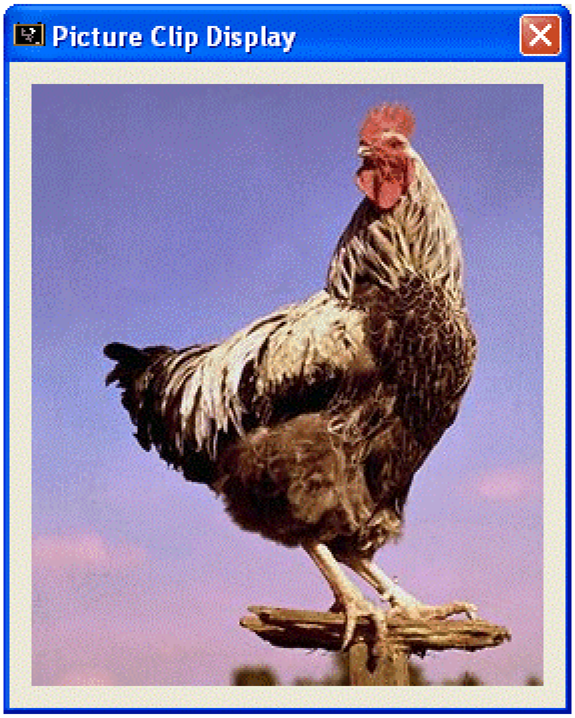
1. Create a new Dialog-based application named **PictClip**and set the Dialog Title to **Picture Clip Display**
2. Delete the TODO line, the OK, and the Cancel buttons
3. Right-click anywhere on the dialog box and click Insert ActiveX Control…
4. In the Insert ActiveX Control dialog box, scroll down and click Microsoft Picture Clip Control, (Version 6):



Microsoft Office Expand Control 9.0
Microsoft Office PivotTable 9.0
Microsoft Office Record Navigation Co
Microsoft Office Spreadsheet 9.0
Microsoft Office UA Control
Microsoft Picture Clip Control, version 6
Microsoft ProgressBar Control, version
Microsoft ProgressBar Control, version
Microsoft RDP Client Control
Microsoft RemoteData Control, version

C:\WINDOWS\System32\PICCLP32.OCX

5. Click OK
6. If you are using MSVC 6, display the Properties window and click the Picture property page. Click the Browse… button.
   If you are using MSVC 7, in the Properties window, click the Picture field. Then click the ellipsis button ...
7. Locate the folder that contains the exercises for this book and display its Pictures folder in the Look In combo box
8. Click GoodDay and click Open
9. Resize the dialog box as you see fit and test the application:



10. Close the dialog and return to MSVC

15.2 Animation Controls

15.2.1 Overview

An animation is a series of pictures put together to produce a video clip. It can be used to display the evolution of an ongoing task to the user. This makes such tasks less boring. For example, making a copy of a CD is usually a long process that can take minutes. To let the user know when such a task is being performed, you can display an animation.

Microsoft Windows provides a few small animations you can use for your applications. These animations, just like many other resources of Visual Studio or Visual C++, are not installed by default. During setup, you can install them if necessary. If you forgot to install them and you need them, run Setup again and select the Add/Remove button. Then click the check box of the Options checked list box. If you want only the videos or some resources, click the Graphics items and click the Change Options, and select the items you want. You can then click Continue to install the videos.

If you need an animation other than those supplied, you may have to create it. Visual C++ is not the place to create an animation. You may need a graphics software to do this.

To use a regular animation, the video must be a standard Microsoft Windows audio/video format: Audio Video Interleaved or AVI. Therefore, it must be a file with the avi extension. If the file has both audio and video, only the video part would be considered.

Practical Learning: Introducing Animations

1. Create a New Dialog-based Project called Video1
2. Delete the TODO line and the OK button
3. Change the caption of the Cancel button to &Close

15.2.2 Animation Control and Properties

An animation first originates from an avi file created by an external application. Therefore, you must already have the video you want to play in your application. To provide an animation for your application, at design time, from the Controls toolbox, click the Animate button and click the desired area on the host.

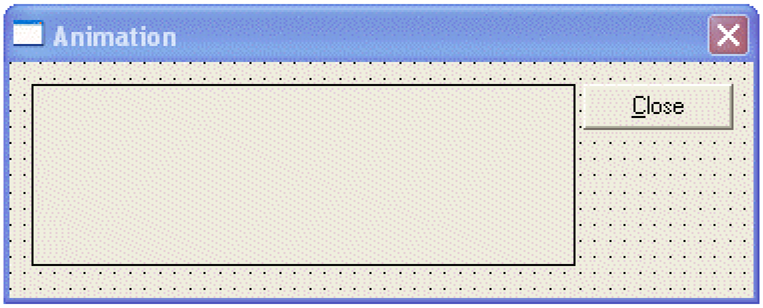
To represent the frame of animation on the dialog box or form, the control draws a rectangular shape. Normally, the animation would be played inside of the area. The person who created the animation likely did not use the same rectangular dimensions when creating the video. Consequently, when it is asked to play, the animation's upper-left corner would be set to correspond to your rectangle's upper-left corner. If you want the animation to be centered in your assigned rectangle, set the control's **Center** property to **True**. This is equivalent to adding the **ACS_CENTER** style. In this case, the center of the video would match the center of your rectangle, even though the animation may still not exactly match the dimensions of your rectangle.

When playing the video, you have the option of displaying the original background color of the video or seeing through. When creating a video, its author can do it with transparency to allow seeing the color of the host. In this case, to display the color of the host while the video is playing, set the **Transparent** property to True. If you are creating the control programmatically, add the **ACS_TRANSPARENT** style.

If you want the video to start displaying immediately once its host (the dialog box or form) comes up, set its **Auto Play** property to True. If you are dynamically creating the control and you want its video to play as soon as its parent comes up, add the **ACS_AUTOPLAY** style

Practical Learning: Animating a Video

1. From the Controls toolbox, click the Animation button and draw a rectangle from the upper left section of the dialog box to the right-middle



2. Using the Properties window, set the Border value to False or uncheck it
3. Set the Auto Play, the Center, and the Transparent values to True or check them

4. Add a (Control) variable for the Animator control and name it **m_Animator**

15.2.3 Animation Methods

The Animator control is based on the **CAnimatorCtrl** class. Therefore, if you want to programmatically create an animation, you must first declare a variable of type, or a pointer to, **CAnimationCtrl**. You can do this in the view or the dialog class. After declaring the variable or pointer, to initialize the object, call its **Create()** method. Here is an example:

```
class CControlsDlg : public CDialog
{
// Construction
public:
        CControlsDlg(CWnd* pParent = NULL);   // standard constructor
        ~CControlsDlg();

        . . .

private:
        CAnimateCtrl *Player;
};
```

```
CControlsDlg::CControlsDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CControlsDlg::IDD, pParent)
{
        //{{AFX_DATA_INIT(CControlsDlg)
                // NOTE: the ClassWizard will add member initialization here
        //}}AFX_DATA_INIT

        Player = new CAnimateCtrl;
}

CControlsDlg::~CControlsDlg()
{
        delete Player;
}

. . .

BOOL CControlsDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        RECT Recto = { 5, 5, 360, 360 };

        Player->Create(WS_CHILD | WS_VISIBLE |
                        ACS_TRANSPARENT | ACS_AUTOPLAY,
                        Recto, this, 0x1884);

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

After creating the control, you must provide a video to play. This is done by opening a video file using the **CAnimateCtrl::Open()** method. It is overloaded with two versions as follows:

```
BOOL Open(LPCTSTR lpszFileName);
BOOL Open(UINT nID);
```

The first version expects the path of the video file. Alternatively, you can first add the file as a resource to your project and use its identifier as argument to the second version. Here is an example:

```
BOOL CControlsDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	RECT Recto = { 5, 5, 360, 360 };

	Player->Create(WS_CHILD | WS_VISIBLE |  ACS_TRANSPARENT,
			Recto, this, 0x1884);
	Player->Open("res\\clock.AVI");

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

As mentioned already, an animation is made of various pictures. Each picture is called a frame. The number of frames that make up an animation can influence its length. A video to play as animation is a file that puts these pictures together. Once you have the video, you can play it in an animation. If you want the animation to start playing as soon as its parent window comes up, you can create it with the **ACS_AUTOPLAY** style. Otherwise, to play the animation, you can call the **CAnimateCtrl::Play()** method. Its syntax is:

```
BOOL Play(UINT nFrom, UINT nTo, UINT nRep);
```

?? The *nFrom* argument specifies the index number of the first frame to play. The frames are stored as a zero-based array. The first frame is 0, the second is 1, etc

?? The *nTo* argument is the last frame to play from the list of frames. If you want to play the video to the end, pass this argument with a value of -1

?? The *nRep* is the number of times the video should be played before stopping. If you want the video to play until you explicitly decide to stop it, pass this argument as -1

Suppose you have a long video or one made of various special pictures, if you want to display just one particular frame of the video, you can call the **CAnimateCtrl::Seek()** method whose syntax is:

```
BOOL Seek(UINT nTo);
```

This method allows the animation to just straight to the *nTo* frame number.

At any time, you can stop the video playing by calling the **CAnimateCtrl::Stop()** method. Its syntax is:

```
BOOL Stop();
```

This member function simply stops the animation.

If you had added your Animator control at design time to the dialog box or form or other parent window, when the parent goes out of scope, it takes the Animator control with it.

If you dynamically created the control, you should make sure that the control is destroyed when its parent also is. This is usually taken care of with the delete operator as done above. Furthermore, when it comes to the Animator control, after using it, to free the memory space it space, you should call the **CAnimateCtrl::Close()** method. Its syntax is:

```
BOOL Close();
```

Practical Learning: Playing a Video File

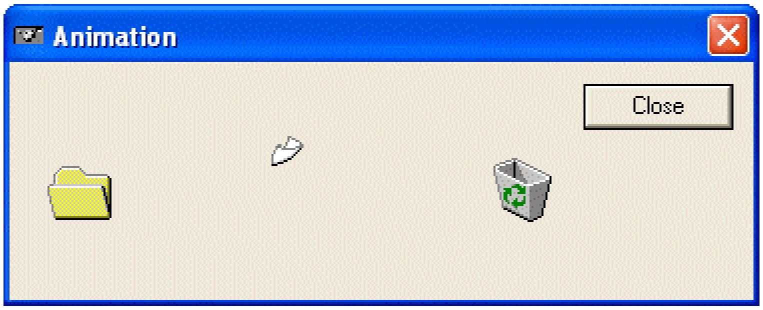
1. In the OnInitDialog() event of the dialog class, load the video with the following line of code:

```
BOOL CAnimation1Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_Animateur.Open("C:\\Program Files\\Microsoft Visual
Studio\\Common\\Graphics\\Videos\\FileDel.AVI");

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. To close and destroy the animation when the user closes the dialog box, add a WM_CLOSE message to the dialog box and implement it as follows:

```
void CAnimation1Dlg::OnClose()
{
	// TODO: Add your message handler code here and/or call default
	m_Animator.Stop();
	m_Animator.Close();

	CDialog::OnClose();
}
```

3. Test the application



4. Return to MSVC

5. To give the user the ability to suspend video playing, add **WM_RBUTTONDOWN** message to the dialog box and implement it as follows:

```
void CAnimation1Dlg::OnRButtonDown(UINT nFlags, CPoint point)
{
	// TODO: Add your message handler code here and/or call default
	m_Animator.Stop();
	CDialog::OnRButtonDown(nFlags, point);
}
```

6. To allow the user to resume playing the video, add a WM_LBUTTONDBLCLK message and implement its as follows:

```
void CAnimation1Dlg::OnLButtonDblClk(UINT nFlags, CPoint point)
{
	// TODO: Add your message handler code here and/or call default
	m_Animator.Play(0, -1, -1);

	CDialog::OnLButtonDblClk(nFlags, point);
}
```

7. Test the application and return to MSVC

15.3 Group Boxes

15.3.1 Introduction

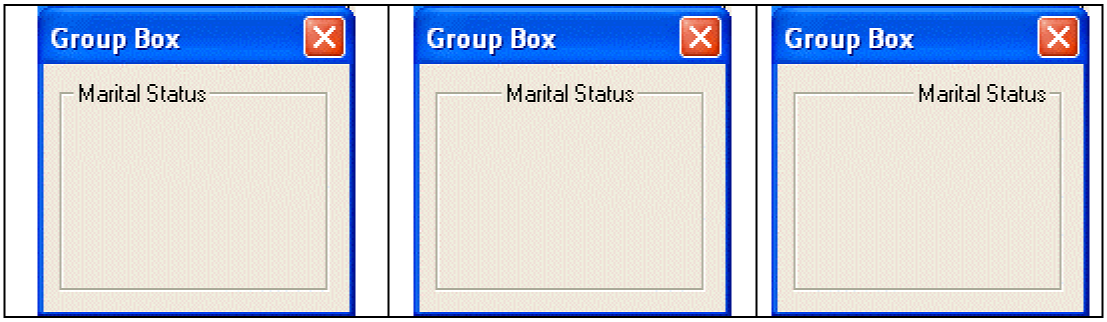
A group box is a static control used to set a visible or programmatic group of controls. A group box primarily draws a rectangular box where it is positioned. This creates an impression of controls that would be treated as a group. A group can then be used for its ability to show a limiting box that encompasses some controls on the dialog box or form but, as we will see when studying radio buttons, a group box can be used to create a group of controls where such controls should or must be treated as one entity.

15.3.2 Group Box Properties and Data Exchange

To create a group box, from the Controls toolbox, click the Group Box button and click the desired area on the form or dialog box. To programmatically create a group box, declare a pointer to CWnd, initialize it with the new operator, and call the **Create()** method. Specify the class name as **STATIC**. In most circumstances, you can use a group control without referring to it programmatically. If you had visually added the control to your application and you plan to refer to it in your code, you should change its identifier to something other than IDC_STATIC

As stated already, one of the most regular roles of a group box is to create a group of controls visibly delimited by a rectangular frame. To indicate what the group represents, the group box is equipped with a **Caption** property for which you supply a string. By default, the text of the group box' caption is aligned to the left. Alternatively, you can position it to the center or the right by selecting the desired value in the Horizontal Alignment combo box of the Properties window:

Horizontal Alignment: Default Horizontal Alignment: Left	Horizontal Alignment: Center	Horizontal Alignment: Right



To programmatically change the caption of the control, call the **CWnd::SetWindowText()** method.

All other properties of a group box are those of a STATIC control.

15.4 Command Buttons

15.4.1 Overview

A Button is a Windows control used to initiate an action. From the user’s standpoint, a button is useful when clicked, in which case the user positions the mouse on it and presses one of the mouse’s buttons.

Note
The MFC library categorizes various controls as buttons. This includes the classic command button, the radio button, and the check box.

There are various kinds of buttons. The most common and regularly used is a rectangular object that the user can easily recognize. In some programming environments, this classic type is called a Command Button. There are other controls that can serve as click controls and initiate the same behavior as if a button were clicked.

From the programmer’s standpoint, a button needs a host, such as a dialog box. To add a button to a dialog box, click it on the Toolbox and click in the desired location on the dialog box.

By default, when you visually create a dialog box, Microsoft Visual C++ adds two buttons: OK and Cancel. If you do not need these buttons, click one and press Delete

15.4.2 Command Buttons Properties and Methods

The most popular button used in Windows applications is a rectangular control that displays a word or a short sentence that directs the user to access, dismiss, or initiate an action or a suite of actions. In Visual C++ applications, this control is implemented using the Button control from the Controls toolbox window. Therefore, to add a button to a container, click the Button control and click on the host, which can be a dialog box or a form. Once you have added the control to your application, you can set its properties using the Properties window.

Like every Window control, a button is recognized by its IDentifier. Because a button is a control, by convention, its identifier's name starts with IDC (the C stands for Control).

From the user’s point of view, the only things important about a button are the message it displays and the action it performs. The word or sentence that displays on top of a button is its **Caption**. By default, after adding a button to a form, the **Caption** property has focus. Therefore, if you start typing, the caption would be changed. At design time, you can set the caption with the necessary string on the **Caption** field of the Properties window. At run time, to change the caption of a button, call the **CWnd::SetWindowText()** method and pass it the necessary string as argument. Here is an example:

```
BOOL CDialog5Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_Submit.SetWindowText("Submit");

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

If you want to access the properties of a control without using an associated variable, you may have to call the **CWnd::GetDlgItem()** method. It comes in two versions as follows:

```
CWnd* GetDlgItem(int nID) const;
void CWnd::GetDlgItem(int nID, HWND* phWnd) const;
```

When calling this method, the first version allows you to assign a CWnd (or derived class) to it. The second returns a handle to the window passed as pointer. In both cases, you must pass the identifier of the control that you want access to. When using the first version, if the control is not a CWnd object, you must cast it its native class. Then you can manipulate the property (or properties) of your choice. Here is an example that accesses a button and changes its caption:

```
BOOL CDialog5Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_Submit.SetWindowText("Submit");

	CButton *btnWhatElse = reinterpret_cast<CButton
*>(GetDlgItem(IDC_BUTTON3));
	btnWhatElse->SetWindowText("What Else?");

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

The second version requires a pointer to a child window that you want to access.

The most popular button captions are OK and Cancel.

The OK caption is set for a dialog box that informs the user of an error, an intermediary situation, or an acknowledgement of an action that was performed on the dialog that hosts the button. Visual C++ makes it easy to add an OK button because in Windows applications, the OK object has a special meaning. To use an OK button, add a button to a form and, from the ID combo box, select the **IDOK** identifier. What makes this constant special is that the MFC library recognizes that, when the user clicks it, if the dialog box is modal, the user is acknowledging the situation. If this dialog box was called from another window using the **DoModal()** method, you can find out if the user had clicked OK and then you can take further action. Therefore, when the user clicks OK, the dialog box calls the **OnOK()** method. Its syntax is:

```
virtual void OnOK();
```

Although it looks like a simple method (and it is) the **OnOK()** method carries the constant value **IDOK** that you can use as a return value of the **DoModal()** method. Therefore, in one step, you can use the **DoModal()** method to display a modal dialog box and find out whether the user clicked OK.

When a dialog box is equipped with an OK button, you should allow the user to press Enter and perform the OK clicking. This is taken care of by setting the **Default Button** property to True or checked.

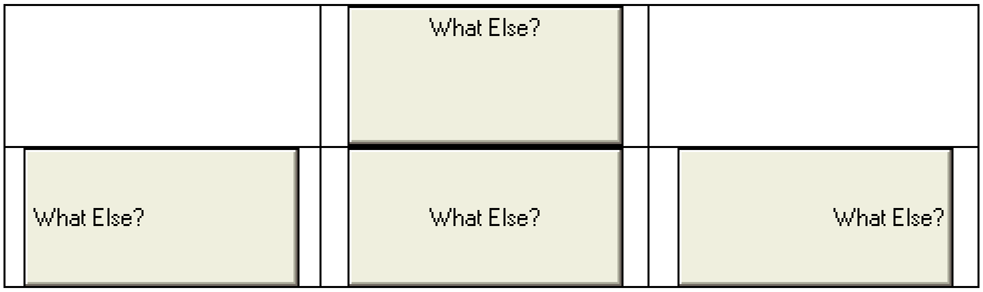
The Cancel caption is useful on a button whose parent (the dialog box) would ask a question or request a follow-up action from the user. A Cancel button is also easy to create by simply adding a button to a dialog box and selecting **IDCANCEL** as its identifier in the ID combo box. Setting a button's identifier to **IDCANCEL** also allows the user to press Esc to dismiss the dialog box.

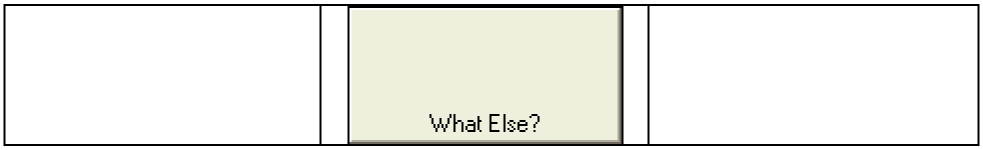
The scenarios described for the OK and the Cancel buttons are made possible only if the compiler is able to check or validate the changes made on a dialog box. To make this validation possible, in your class, you must overload the **CWnd::DoDataExchange()** method. Its syntax is:

```
virtual void DoDataExchange( CDataExchange* pDX );
```

This method is used internally by the application (the framework) to find out if data on a dialog box has been changed since the object was displayed. This method does two things it ensure the exchange of data among controls and it validates the values of those controls. In reality, it does not inherently perform data validation, meaning it would not allow or disallow value on a control. Instead, the compiler uses it to create a table of the controls on the dialog box, their variables and values, allowing other controls to refer to it for data exchange and validating. If you want to find out the data a user would have typed or selected in a control, you would have to write the necessary code.

By default, the caption on a button is positioned in the middle and the center of the control. At design time, you can control this position using the Horizontal and the Vertical Alignments on the Properties window:





	Horz Align: Default Vert Align: Top	
Horz Align: Left Vert Align: Default	Horz Align: Default or Center Vert Align: Default or Center	Horz Align: Right Vert Align: Default
	Horz Align: Default Vert Align: Bottom	

After creating a control, to make sure that it displays when its host control comes up, set its Visible property to True or checked (the default). Otherwise, if you want it to be hidden for example to wait for an intermediary action from the user, you can set its Visible property to False or unchecked. For example, to display a control, whether it is hidden or not, call the CWnd::ShowWindow() method and pass SW_SHOW as its argument:

```
BOOL CDialog5Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_Submit.ShowWindow(SW_SHOW);
	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

In the same way, to hide a control, call it by passing the SW_HIDE constant as argument.

For the user to be able to use a control such as clicking a button, the control must allow it. This characteristic of Windows objects is controlled by the CWnd::EnableWindow() method. Its syntax is:

```
BOOL EnableWindow(BOOL bEnable = TRUE);
```

This method is used to enable or disable a control. The default value of the argument bEnable is set to TRUE, which would display a control. To disable a control, set the argument to FALSE. He re is an example:

```
void CDialog5Dlg::OnLetItBe()
{
	// TODO: Add your control notification handler code here
	m_Submit.EnableWindow(FALSE);
}
```

15.4.3 Buttons Messages

The most regular action users perform on a button is to click it. When a user does this, the button sends a **BN_CLICKED** message. In some but rare circumstances, you may also ask the user to double-click a button. Over all, you will take care of most message handling when the user clicks a button.

There are other messages that you can handle through a button.

To close a dialog box, you can use the Win32 API's **PostQuitMessage()** function. Its syntax is:

```
VOID PostQuitMessage(int nExitCode);
```

This function takes one argument, which is an integer. The argument could be set to almost any integer value although it should be WM_QUIT. Here is an example:

```
void CDialog5Dlg::OnBtnClose()
{
	// TODO: Add your control notification handler code here
	PostQuitMessage(125);
}
```

Although the MFC provides enough messages associated with the various controls, in some circumstances you will need use a message that is not necessarily associated with the control. In such a case, you can call the **CWnd::SendMessage()** method. Its syntax is:

```
LRESULT SendMessage(UINT message, WPARAM wParam = 0, LPARAM lParam = 0);
```

The first argument of this method can be a Win32 message or constant. Examples would be WM_CLOSE or WM_ACTIVATE. The *wParam* and *lParam* arguments can be additional (Win32) messages.

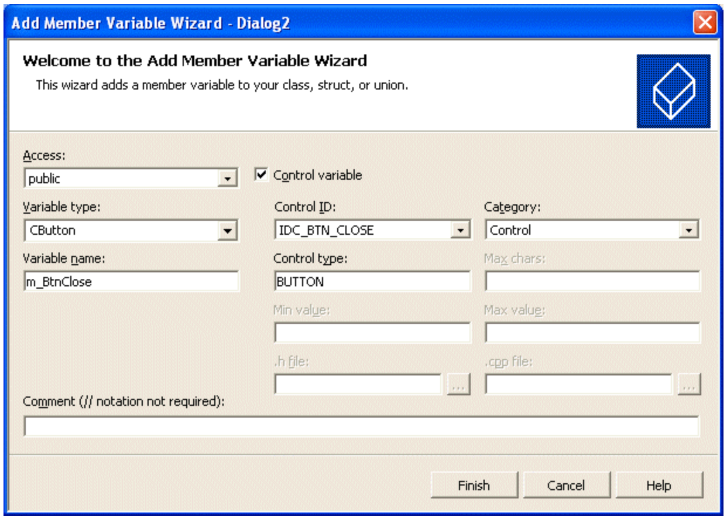
The **WinExec()** function is used to run an application. Its syntax is:

```
UINT WinExec(LPCSTR lpCmdLine, UINT uCmdShow);
```

The *lpCmdLine* argument specifies the name or path of the application you want to display. The *uCmdShow* specifies how the application should be displayed. It uses the same values as the **CWnd::ShowWindow()** method.

Practical Learning: Using Buttons

1. Create a new Dialog Based application named **AppLauncher**
2. On the dialog, delete the TODO line, the OK, and the Cancel buttons
3. On the Controls window, click the Button control ▭ and click in the lower section of the dialog box
4. Display the button's Properties window and change its IDentifier to IDC_BTN_CLOSE
5. Right-click the button and click Add Variable
6. Set the Variable Name to m_BtnClose and make sure the Variable Type is CButton



Add Member Variable Wizard - Dialog2
Welcome to the Add Member Variable Wizard
This wizard adds a member variable to your class, struct, or union.
Finish
Cancel
Help

Access: public

☑ Control variable

Variable type: CButton

Control ID: IDC_BTN_CLOSE

Category: Control

Variable name: m_BtnClose

Control type: BUTTON

Max chars:

Min value:

Max value:

.h file:

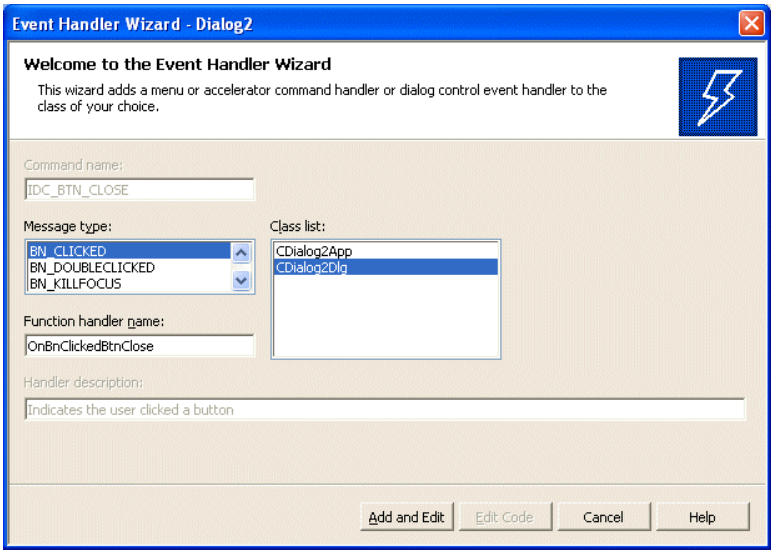
.cpp file:

Comment (// notation not required):

7. Click Finish
8. In the OnInitDialog() event, set the caption of the button to "Close" as follows:

```
BOOL CDialog2Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_BtnClose.SetWindowText("Close");

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

9. Test the application
10. To close the dialog box, click its System Close button
11. Display the dialog box. Right-click the button and click Add Event Handler



Event Handler Wizard - Dialog2

Welcome to the Event Handler Wizard

This wizard adds a menu or accelerator command handler or dialog control event handler to the class of your choice.

Command name: IDC_BTN_CLOSE

Message type: BN_CLICKED, BN_DOUBLECLICKED, BN_KILLFOCUS

Class list: CDialog2App, CDialog2Dlg

Function handler name: OnBnClickedBtnClose

Handler description: Indicates the user clicked a button

12. In the Event Handler Wizard, make sure the Message Type is set to BN_CLICKED and the Class List is set to CDialog2Dlg. Then click the Add And Edit button
13. Implement the OnClick event as follows:

```
void CDialog2Dlg::OnBnClickedBtnClose()
{
	// TODO: Add your control notification handler code here
	PostQuitMessage(WM_QUIT);
}
```

14. To programmatically change the caption of the dialog box, access the OnInitDialog() event and set the Caption of the dialog box to "Application Launcher" as follows:

```
BOOL CDialog2Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_BtnClose.SetWindowText("Close");
	this->SetWindowText("Application Launcher");

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

15. Add a new button to the upper left section of the dialog box
16. Set it ID to IDC_BTN_WORDPAD and its Caption to WordPad
17. Double-click the WordPad button and implement its OnClick event as follows:

```
void CDialog2Dlg::OnBnClickedBtnWordpad()
{
	// TODO: Add your control notification handler code here
	WinExec("Write.exe", SW_NORMAL);
}
```

18. Test the application. When the dialog box displays, click the WordPad button to see it launch
19. Close the dialog box and return to MSVC

15.5 Property Sheet and Wizard Buttons

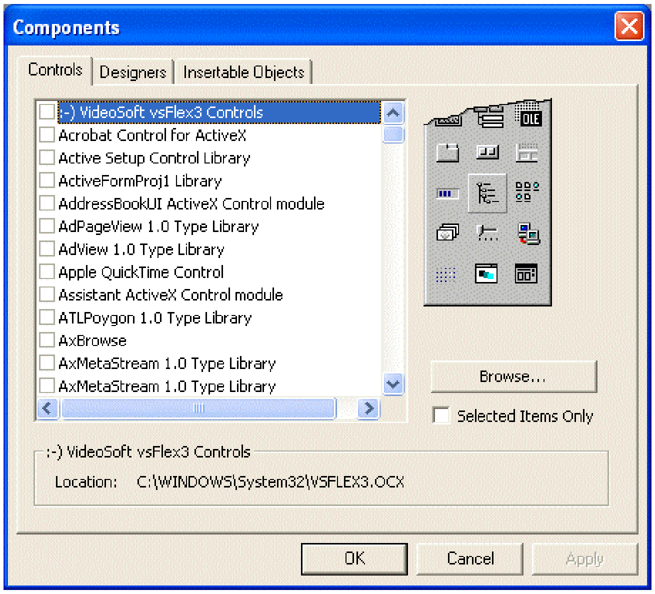
15.5.1 Property Sheet Buttons

There are two types of property sheets: modal and modeless. A modeless property sheet does not display buttons.

A modal property sheet is equipped with command buttons that allow the user to make a decision after selecting items on the pages or changing the values of the page's controls. By default, after creating a property sheet, it is equipped with the OK, the Cancel, and the Apply buttons.

By design, the **CPropertySheet** class, which is the implementer of the property sheet, has a member variable called **m_psh**. This member represents the **PROPSHEETHEADER** structure, which is the Win32 implementer of a property sheet. One way you can use the **m_psh** member variable is to hide the Apply button if you do not need it in your application. Otherwise, this button is available by default on a property sheet.

Here is an example from the Components dialog box of Microsoft Visual Basic:



- ☐ :-) VideoSoft vsFlex3 Controls
- ☐ Acrobat Control for ActiveX
- ☐ Active Setup Control Library
- ☐ ActiveFormProj1 Library
- ☐ AddressBookUI ActiveX Control module
- ☐ AdPageView 1.0 Type Library
- ☐ AdView 1.0 Type Library
- ☐ Apple QuickTime Control
- ☐ Assistant ActiveX Control module
- ☐ ATLPolygon 1.0 Type Library
- ☐ AxBrowse
- ☐ AxMetaStream 1.0 Type Library
- ☐ AxMetaStream 1.0 Type Library

:-) VideoSoft vsFlex3 Controls

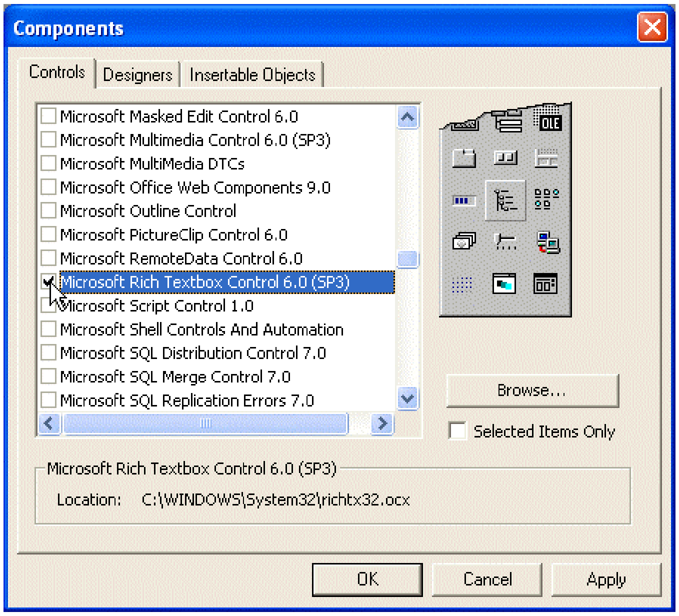
Location: C:\WINDOWS\System32\VSFLEX3.OCX

In this classic design, the functionality of the buttons is commonly standardized:

?? The OK button allows the user to validate any change(s) made on the control(s) of the property page(s) and close the dialog box. For example, if the user changes text from an edit box and clicks OK, the application that called the dialog will have the opportunity to acknowledge the change and act accordingly, and the property sheet would be closed

?? If the user clicks Cancel, the change(s) made on the property page's control(s) would be discarded, not taken into consideration, and the property sheet would be closed

?? When the property sheet comes up, the Apply button on the property page is disabled. If the user changes something on the controls of the property page(s), the Apply button would become enabled:



☐ Microsoft Masked Edit Control 6.0
☐ Microsoft Multimedia Control 6.0 (SP3)
☐ Microsoft MultiMedia DTCs
☐ Microsoft Office Web Components 9.0
☐ Microsoft Outline Control
☐ Microsoft PictureClip Control 6.0
☐ Microsoft RemoteData Control 6.0
☑ Microsoft Rich Textbox Control 6.0 (SP3)
☐ Microsoft Script Control 1.0
☐ Microsoft Shell Controls And Automation
☐ Microsoft SQL Distribution Control 7.0
☐ Microsoft SQL Merge Control 7.0
☐ Microsoft SQL Replication Errors 7.0

☐ Selected Items Only

Microsoft Rich Textbox Control 6.0 (SP3)
Location: C:\WINDOWS\System32\richtx32.ocx

Once the Apply button is enabled, the user can use it. If the user clicks the Apply button, 1) any change(s) made on the control(s) is(are) sent to the object that called the property sheet, 2) the Apply button becomes disabled again, 3) the dialog box remains opened.

This description is conform to the standards or suggestions of the Microsoft Windows operating system. In reality, you are completely free to do what you want with the buttons on the property sheet:

?? You can hide them

?? you can display them

?? you can completely delete (destroy) any unneeded button

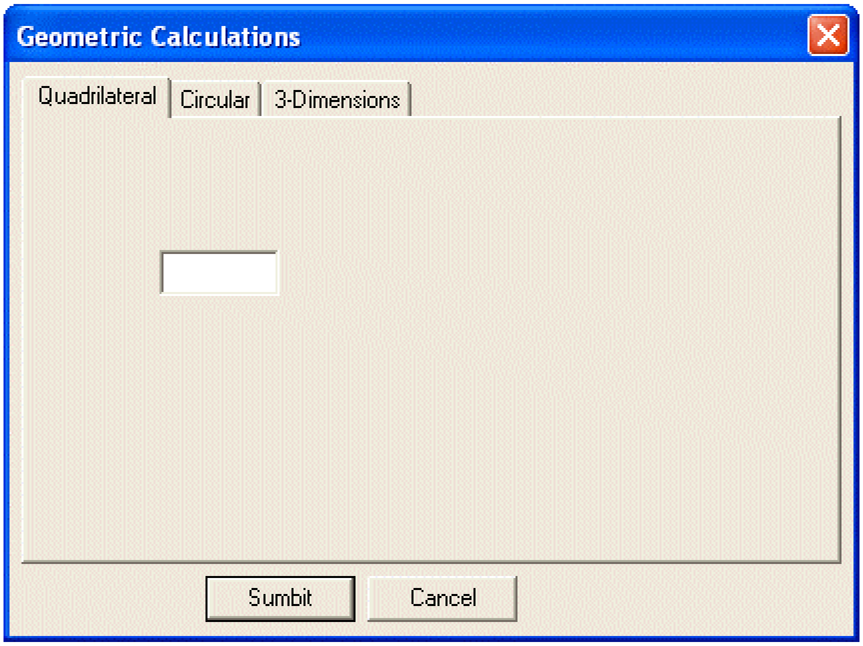
?? you can add as many buttons as you judge necessary and as the bottom area can afford

?? you can change the captions of the buttons

Some of these issues we already know how to do. We already know that each control of an MFC application has an identifier. The buttons automatically added to a property sheet are identified as **IDOK** for the OK button, **IDCANCEL** for the Cancel button, **ID_APPLY_NOW** for the Apply button, and **IDHELP** for the Help button. Therefore, to manipulate any of these buttons, you can call the **CWnd::GetDlgItem()** method to get a handle to the desired button and do what you want with it. Suppose you had created the Geometry application we dealt with in Chapter 13. Here is an example code you can use to change the caption of a button, hide another button, or simply destroy another:

```
BOOL CGeomeSheet::OnInitDialog()
```

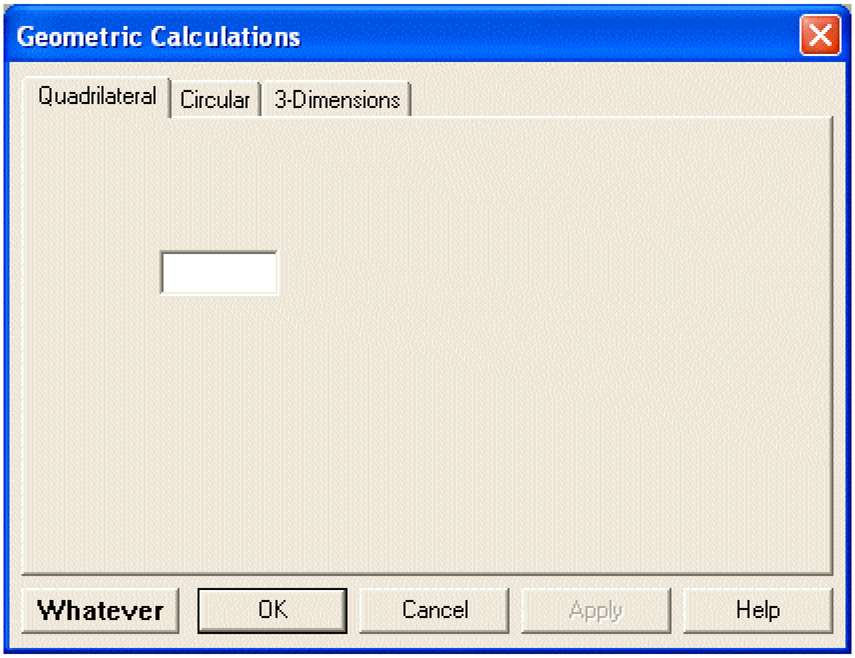
```
{
	BOOL bResult = CPropertySheet::OnInitDialog();

	// TODO: Add your specialized code here
	// Change the caption of the OK button
	CButton *btnOK;

	btnOK = reinterpret_cast<CButton *>(GetDlgItem(IDOK));
	btnOK->SetWindowText("Sumbit");

	// Hide the Apply button
	CButton *btnApply;

	btnApply = reinterpret_cast<CButton *>(GetDlgItem(ID_APPLY_NOW));
	btnApply->ShowWindow(FALSE);

	// Destroy the Help button
	CButton *btnHelp;

	btnHelp = reinterpret_cast<CButton *>(GetDlgItem(IDHELP));
	btnHelp->DestroyWindow();

	return bResult;
}
```



To add a button, declare a pointer to **CButton** and call its **Create()** method to initialize. We have seen various examples of how to dynamically create a control. If you decide to dynamically create a button, some of the issues you would have to deal with here are the location and probably the size of the new button, which have little to do with programming but with geometry. Here is an exa mple:

```
BOOL CGeomeSheet::OnInitDialog()
{
	BOOL bResult = CPropertySheet::OnInitDialog();

```

```
        // TODO:  Add your specialized code here
        // A pointer to button we will need
        CButton *btnApply;
        // We will need to location and dimensions of the Apply button
        CRect    RectAppl;

        // Get a handle to the Apply button
        btnApply = reinterpret_cast<CButton *>(GetDlgItem(ID_APPLY_NOW));
        // Get the location and the dimensions of the Apply button
        btnApply->GetWindowRect(&RectApply);

        // Convert the location and dimensions to screen coordinates
        ScreenToClient(&RectApply);

        CButton *Whatever = new CButton;

        Whatever->Create("&Whatever", WS_CHILD | WS_VISIBLE,
                         CRect(6, RectApply.top, 85,
                         RectApply.top+RectApply.Height()),
                         this, 0x188);
        return bResult;
}
```



Another issue you would deal with is each of the messages sent by your dynamic button.

As seen on the above picture, manipulating one button has no influence on the other(s). For example, if you destroy the Cancel button, the OK button does not move to the right. You would have to reposition any button as you judge it necessary.

We have already mentioned that, by standard and by design, the Apply button is disabled when the property sheet comes up. It is supposed to become enabled once the user gets any control “dirty”, that is, once a control, any control, is not the same as it was when the dialog box came up, the Apply button becomes available. To enable this control programmatically, once a control becomes dirty, call the **CPropertyPage::SetModified().** Its syntax is:

```
void SetModified(BOOL bChanged = TRUE);
```

This method is called by the control whose value you want to validate once the user has modified it.

When the user clicks the OK button, the **CPropertyPage::OnOK()** event fires. By design, the changes made on the controls are acknowledged. The controls receive the status of "clean". The property sheet closes.

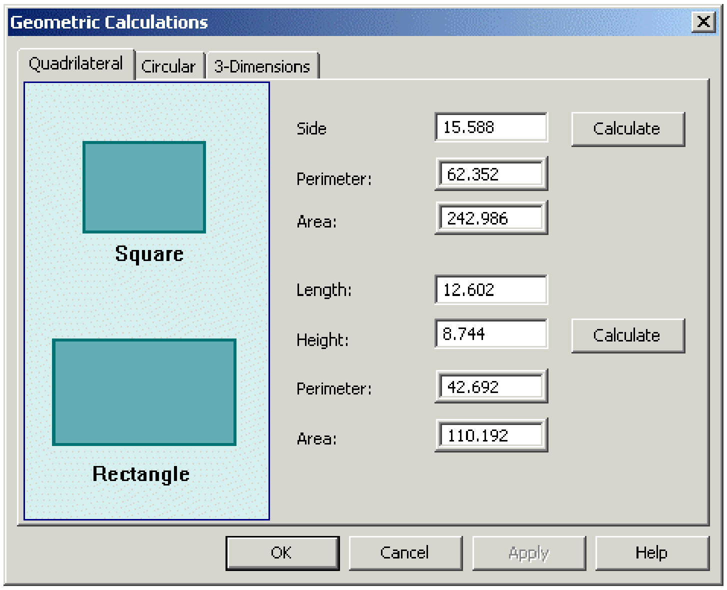
When the user clicks the Cancel button, the **CPropertyPage::OnCancel()** event fires. By design, the changes made on the controls are dismissed. The controls values are kept as they were when the property sheet displayed as long as the user did not previously click Apply since the property sheet was opened. The property sheet closes.

When the user clicks the Apply button, the **CPropertyPage::OnApply()** event fires. The changes that were made on the controls are acknowledged. The property sheet stays opened.

Once again, these behaviors are the default suggested by the standard but you can change them as you wish, although you should remain with these suggestions because your users may be more familiar with them.

Practical Learning: Manipulating Property Sheet Buttons

1. Open the Geometry application. If you do not have it, locate the exercises that accompany this book and open the Geometry4 application
2. Execute the application to review its interface:

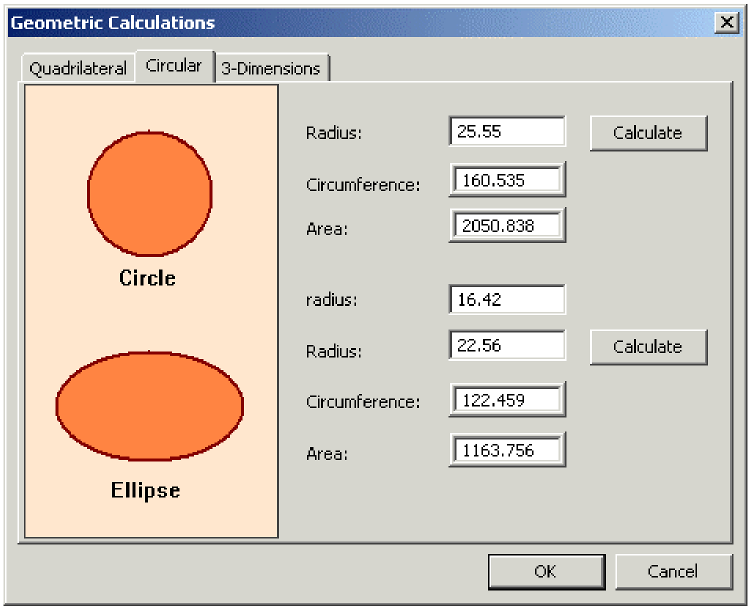


Side 15.588
Perimeter: 62.352
Area: 242.986
Length: 12.602
Height: 8.744
Perimeter: 42.692
Area: 110.192

3. Close it and return to MSVC
4. To delete the Apply and the Help buttons, call their **DestroyWindow()** method. Then move the OK and the Cancel buttons to the right where the other buttons were. To do this, implement the OnInitDialog event as follows:

```
BOOL CGeomeSheet::OnInitDialog()
{
	BOOL bResult = CPropertySheet::OnInitDialog();

	// TODO:  Add your specialized code here
	// A pointer to each button we will need
	CButton *btnOK, *btnCancel, *btnApply, *btnHelp;
	// We will need to location and dimensions of Apply and Help
	CRect	RectApply, RectHelp;

	// Get handles to the OK and Cancel buttons
	btnOK = reinterpret_cast<CButton *>(GetDlgItem(IDOK));
	btnCancel = reinterpret_cast<CButton *>(GetDlgItem(IDCANCEL));

	// Get a handle to the Apply button
	btnApply = reinterpret_cast<CButton *>(GetDlgItem(ID_APPLY_NOW));
	// Get the location and the dimensions of the Apply button
	btnApply->GetWindowRect(&RectApply);

	// Get a handle to the Help button
	btnHelp = reinterpret_cast<CButton *>(GetDlgItem(IDHELP));
	// Get the location and the dimensions of the Help button
	btnHelp->GetWindowRect(&RectHelp);

	// Dismiss the Apply and the Help buttons
	btnApply->DestroyWindow();
	btnHelp->DestroyWindow();

	// Convert  the location and dimensions to screen coordinates
	ScreenToClient(&RectApply);
	ScreenToClient(&RectHelp);

	// Put the OK button where the Apply button was
	btnOK->SetWindowPos(NULL, RectApply.left, RectApply.top, 0, 0,
				SWP_NOSIZE | SWP_NOZORDER | SWP_SHOWWINDOW);
	// Put the Cancel button where the Help button was
	btnCancel->SetWindowPos(NULL, RectHelp.left, RectHelp.top, 0, 0,
				SWP_NOSIZE | SWP_NOZORDER | SWP_SHOWWINDOW);

	return bResult;
}
```

5. Test the application



Radius: 25.55
Circumference: 160.535
Area: 2050.838
radius: 16.42
Radius: 22.56
Circumference: 122.459
Area: 1163.756

6. Close the application and return to MSVC
7. In order to enable the Apply button, we need to have it.
   To redisplay the Apply button, change the CGeomeSheet::OnInitDialog() event as follows:

```
BOOL CGeomeSheet::OnInitDialog()
{
	BOOL bResult = CPropertySheet::OnInitDialog();

	// TODO:  Add your specialized code here
	// A pointer to each button we will need
	CButton *btnOK, *btnCancel, *btnApply, *btnHelp;
	// We will need to location and dimensions of Apply and Help
	CRect    RectCancel, RectApply, RectHelp;

	// Get handles to all buttons
	btnOK = reinterpret_cast<CButton *>(GetDlgItem(IDOK));
	btnCancel = reinterpret_cast<CButton *>(GetDlgItem(IDCANCEL));
	btnApply = reinterpret_cast<CButton *>(GetDlgItem(ID_APPLY_NOW));
	btnHelp = reinterpret_cast<CButton *>(GetDlgItem(IDHELP));

	// Get the location and the dimensions of the buttons
	btnCancel->GetWindowRect(&RectCancel);
	btnApply->GetWindowRect(&RectApply);
	btnHelp->GetWindowRect(&RectHelp);

	// Destroy the Help button
	btnHelp->DestroyWindow();

	// Convert the location and dimensions to screen coordinates
	ScreenToClient(&RectCancel);
	ScreenToClient(&RectApply);
	ScreenToClient(&RectHelp);

	// Put the Apply button where the Help button was
	btnApply->SetWindowPos(NULL, RectHelp.left, RectHelp.top, 0, 0,
				SWP_NOSIZE | SWP_NOZORDER | SWP_SHOWWINDOW);
	// Put the Cancel button where the Apply button was
	btnCancel->SetWindowPos(NULL, RectApply.left, RectApply.top, 0, 0,
	                        SWP_NOSIZE | SWP_NOZORDER | SWP_SHOWWINDOW);
	// Put the OK button where the Cancel button was
	btnOK->SetWindowPos(NULL, RectCancel.left, RectCancel.top, 0, 0,
	                        SWP_NOSIZE | SWP_NOZORDER | SWP_SHOWWINDOW);

	return bResult;
}
```

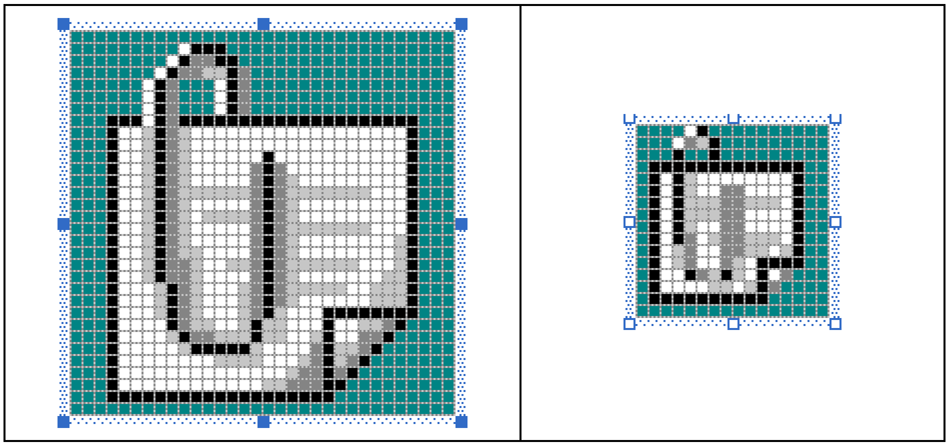
8. We will enable the Apply button when any control on the property pages changes value.
   Display the IDD_QUADRILATERAL dialog box and double-click the Side edit control. If you are using MSVC 6, accept the suggest Member Function Name and click OK. Implement the event as follows:

```
void CQuadrilateral::OnEnChangeEdtSside()
{
	// TODO:  Add your control notification handler code here
	SetModified();
}
```

9. Display the IDD_G3D dialog box and double-click the Side edit box. If you are using MSVC 6, accept the suggest Member Function Name and click OK

```
void CGeome3D:: OnEnChangeEdtUside()
{
	// TODO: Add your control notification handler code here
	SetModified();
}
```

10. Test the application and return to MSVC
11. Using either MFC AppWizard (exe)(MSVC 6) or MFC Application (MSVC 7), create a new application named **Associates**
12. Create it as a **Single Document** without Printing And Print Preview
13. Change the design of the IDR_MAINFRAME as follows:

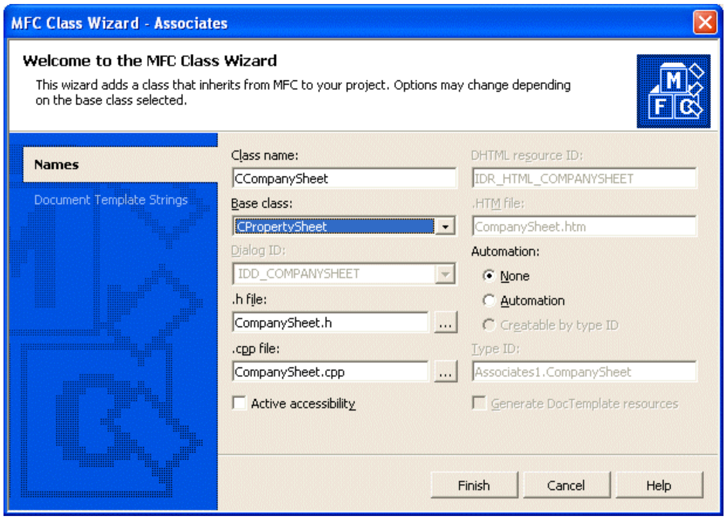


14. Access the **CMainFrame::PreCreateWindow()** event and, to remove the Untitled word from the title bar, type the following:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreateWindow(cs) )
		return FALSE;
	// TODO: Modify the Window class or styles here by modifying
```

```
    //  the CREATESTRUCT cs
    cs.style &= ~FWS_ADDTOTITLE;

    return TRUE;
}
```

15. Test the application. Then close it and return to MSVC
16. Open the IDR_MAINFRAME menu and change the Caption of the New menu item under File to **&New…\tCtrl+N**
17. Display the Add Resource dialog box and expand the Dialog node. Double-click IDD_PROPPAGE_MEDIUM
18. Delete its TODO line. Change its **Caption** to **Company Forms** and its **ID** to **IDD_COMPANY_FORMS**
19. Add A New Class for the new dialog box. Name it **CCompanyForms** and base it on **CPropertyPage**
20. Add another IDD_PROPPAGE_ MEDIUM dialog. Delete its TODO line. Change its **Caption** to **Projects** and its **ID** to **IDD_COMPANY_PROJECTS**
21. Add A New Class for the new dialog box. Name it **CCompanyProjects** and base it on **CPropertyPage**
22. Add a New MFC Class named CCompanySheet and based on CPropertySheet



Welcome to the MFC Class Wizard

This wizard adds a class that inherits from MFC to your project. Options may change depending on the base class selected.

Class name: CCompanySheet

Base class: CPropertySheet

Dialog ID: IDD_COMPANYSHEET

.h file: CompanySheet.h

.cpp file: CompanySheet.cpp

☐ Active accessibility

DHTML resource ID: IDR_HTML_COMPANYSHEET

.HTM file: CompanySheet.htm

Automation: ● None ○ Automation ○ Createable by type ID

Type ID: Associates1.CompanySheet

☐ Generate DocTemplate resources

23. In the header of the property sheet class, delete the constructor that takes a null-terminated string and add the previously created property page classes in the header of the:

```
#pragma once
#include "CompanyForms.h"
#include "CompanyProjects.h"

// CCompanySheet

class CCompanySheet : public CPropertySheet
{
    DECLARE_DYNAMIC(CCompanySheet)

public:
    CCompanySheet(UINT nIDCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
```

```
	//CCompanySheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
	virtual ~CCompanySheet();

protected:
	DECLARE_MESSAGE_MAP()

public:
	CCompanyForms Forms;
	CCompanyProjects Projects;
};
```

24. Use the constructor to add each page and remove the Apply button:

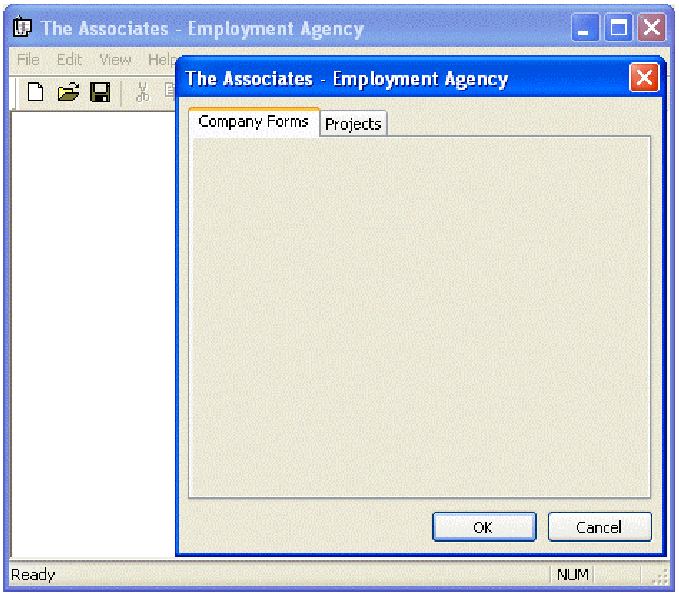
```
CCompanySheet::CCompanySheet(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
	:CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
	m_psh.dwFlags |= PSH_NOAPPLYNOW;
	AddPage(&Forms);
	AddPage(&Projects);
}
```

25. Access the String Table and change the caption of **AFX_IDS_APP_TITLE** to **The Associates - Employment Agency**

26. Add an COMMAND (MSVC 6) or an Event Handler (MSVC 7) event for the ID_FILE_NEW menu identifier associated with the CMainFrame class and implement it as follows:

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "Associates.h"

#include "MainFrm.h"
#include "CompanySheet.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

. . .

// CMainFrame message handlers

void CMainFrame::OnFileNew()
{
	// TODO: Add your command handler code here
	CCompanySheet CompSheet(AFX_IDS_APP_TITLE);

	CompSheet.DoModal();
}
```

27. Test the application


The Associates - Employment Agency
File Edit View Help
The Associates - Employment Agency
Company Forms
Projects
OK
Cancel
Ready
NUM

28. Close it and return to MSVC

29. Add an IDD_PROPPAGE_ LARGE dialog. Delete its TODO line. Change its **Caption** to **Personal Information** and its **ID** to **IDD_PERS_INFO**

30. Add A New Class for the new dialog box. Name it **CPersonalInfo** and base it on **CPropertyPage**

31. Add another IDD_PROPPAGE_ LARGE dialog. Delete its TODO line. Change its **Caption** to **Education and Experience** and its **ID** to **IDD_EDUC_EXP**

32. Add A New Class for the new dialog box. Name it **CEducExperience** and base it on **CPropertyPage**

33. Add another IDD_PROPPAGE_ LARGE dialog. Delete its TODO line. Change its **Caption** to **New Employee Survey** and its **ID** to **IDD_EMPL_SURVEY**

34. Add A New Class for the new dialog box. Name it **CEmplSurvey** and base it on **CPropertyPage**

35. Add an OnInitDialog event for each of the classes we have previously added to the application (MSVC 6: display the ClassWizard, select the class such as CCompanyForms in the Object IDs and, in the Messages list, double-click WM_INITDIALOG. MSVC 7: in the Class View, click the class such as CCompanyForms and, in the Properties window, click the Overrides button ; then, in the OnInitDialog combo box, select the only item in the list)

36. Add A New MFC Class named **CEmplAppSheet** and based on **CPropertySheet**

37. In the header of the new property sheet class, delete the constructor that takes a null-terminated string as argument and add the previously created property page:

```
#pragma once
#include "EducExperience.h"
#include "PersonalInfo.h"
#include "EmplSurvey.h"

// CEmplAppSheet

class CEmplAppSheet : public CPropertySheet
{
	DECLARE_DYNAMIC(CEmplAppSheet)
public:
    CEmplAppSheet(UINT nIDCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
//  CEmplAppSheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL, UINT
iSelectPage = 0);
    virtual ~CEmplAppSheet();

protected:
    DECLARE_MESSAGE_MAP()

public:
    CPersonalInfo    PersInfo;
    CEducExperience EducExp;
    CEmplSurvey      Survey;
};
```

38. Use the constructor to add each page:

```
CEmplAppSheet::CEmplAppSheet(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
    AddPage(&PersInfo);
    AddPage(&EducExp);
    AddPage(&Survey);
}
```

39. To start the wizard when the user clicks OK on the property sheet, we wil laccess the OnOK() event of the CCompanyForms class.
If you are using MSVC 6, display the ClassWizard. In the Class Name combo box and in the Object IDs list, select CCompanyForms. In the Messages list, double-click WM_OnOK
If you are using MSVC 7, in the Class View, click CCompanyForms. In the Properties window, click the Overrides button ◆. In the OnOK combo box, select the only item in the list

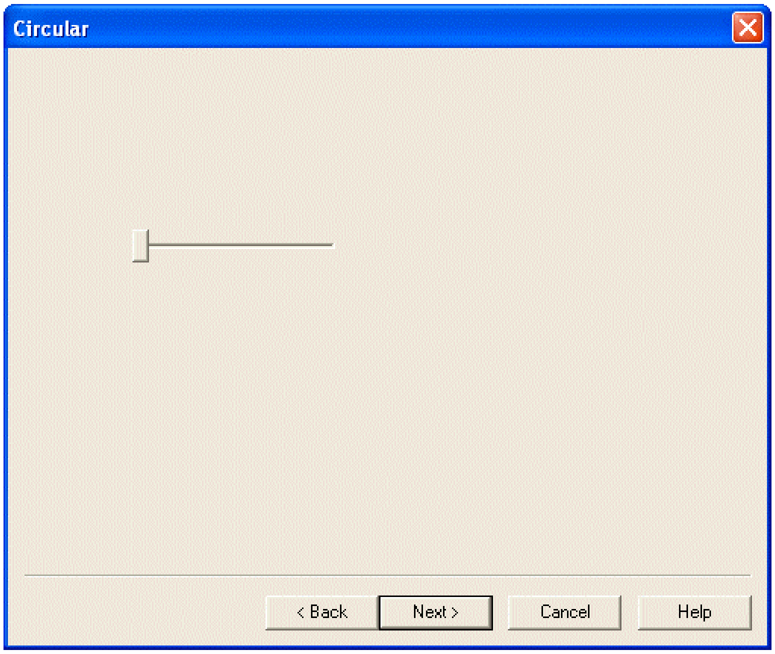
40. Implement it as follows (in reality, the wizard should be displayed based on an item the user had selected in a property page but we have not covered any of the necessary controls that would be used to apply such a behavior):

```
// CompanyForms.cpp : implementation file
//

#include "stdafx.h"
#include "Associates.h"
#include "CompanyForms.h"
#include "EmplAppSheet.h"

. . .

void CCompanyForms::OnOK()
{
    // TODO: Add your specialized code here and/or call the base class
    CEmplAppSheet EmplSheet(AFX_IDS_APP_TITLE);

    EmplSheet.SetWizardMode();
    EmplSheet.DoModal();

    CPropertyPage::OnOK();
}
```

41. Test the application

15.5.2 Wizard Buttons

Like a property sheet, a wizard can be made of one or more pages (it is possible but not practical to have a wizard with only one page). A wizard you have just created may have the Back, Next, Cancel, and Help buttons:



To exploit it, the user would click a Next or Back buttons. The Next button allows the user to access the page that succeeds the current one. The Back button implements the opposite, allowing the user to access the previous page. If the wizard is equipped with a Cancel button, the user can close and dismiss it at any time. This, by default allows you to ignore anything the user did on the wizard.

As with anything else in the world or programming, you should make your wizard intuitive and friendly. For example, on the first page, since there is no reason for the user to access the previous page, you should disable it. On the other hand, when the user gets to the last page, since there is no succeeding page, there is no need for a Next button. Consequently, you should replace it with a Finish button. Fortunately, the display of these buttons is not the most difficult thing to manage on a wizard.

In order to decide what button(s) should be available when a particular page is accessed, generate its **CPropertyPage::OnSetActive()** event. Its syntax is:

```
virtual BOOL OnSetActive( );
```

This event fires when a page of your choice is accessed, allowing you to do what you want on the wizard page. One of the actions you can take when a page displays is to decide what button(s) should be displayed on the wizard. The buttons of a wizard are managed through the **CPropertySheet::SetWizardButtons()** method. Its syntax is:

```
void SetWizardButtons(DWORD dwFlags);
```

The *dwFlags* argument is a constant or a combination of values that defines the buttons to display. The possible values are **PSWIZB_BACK** for the Back button, **PSWIZB_NEXT** for the Next button, **PSWIZB_FINISH** for the Finish button, and **PSWIZB_DISABLEDFINISH** to disable Finish button disabled.

On the first page, to display the Next button while disabling the Back button, you can pass only the **PSWIZB_NEXT** value. On the last page, you should display the Back and the Finish buttons. All pages between the first and the last should display the Back and the Next buttons, unless you have a reason to do otherwise.

To add your own button to the wizard, you can use the same approach we saw earlier. Here is an example:

```
CmyWizardSheet::OnInitDialog()
{
	BOOL bResult = CPropertySheet::OnInitDialog();

	CButton *btnWhatever = new CButton;

	BtnWhatever->Create(“Whatever”, WS_CHILD | WS_VISIBLE,
	                    Crect(10, 362, 126, 385), this, 0x122);

	Return bResult;
}
```

When the user clicks the Back button, the **CPropertyPage::OnWizardBack()** event fires, giving you the opportunity to do what you judge necessary. Its syntax is:

```
virtual LRESULT OnWizardBack();
```

You should use this event on a page that has the Back button and if this button is enabled. When the user clicks the Next button, the **CPropertyPage::OnWizardNext()** event fires, allowing you to take appropriate measures. Its syntax is:

```
virtual LRESULT OnWizardNext();
```

You should use this event on a page that has the Next button. When the user clicks the Finish button, the **CPropertyPage::OnWizardFinish()** event fires. Its syntax is:

```
virtual BOOL OnWizardFinish( );
```

This event should be implement when either the user is on the last page that has the Finish button or at any time if the wizard has a permanent Finish button available on all pages.

Practical Learning: Implementing Wizard Buttons

1. Access the events of the CPersonalInfo class and fire its OnSetActive.
2. Implement it as follows:

```
BOOL CPersonalInfo::OnSetActive()
{
    // TODO: Add your specialized code here and/or call the base class
    CPropertySheet *EmplSheet = reinterpret_cast<CPropertySheet *>(GetParent());
    EmplSheet->SetWizardButtons(PSWIZB_NEXT);

    return CPropertyPage::OnSetActive();
}
```

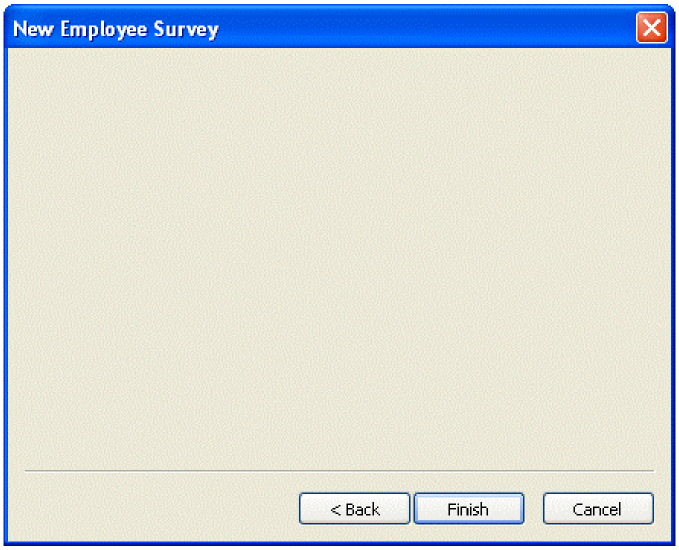
3. In the same way, fire the OnSetActive event of the CEducExperience class and implement it as follows:

```
BOOL CEducExperience::OnSetActive()
{
    // TODO: Add your specialized code here and/or call the base class
    CPropertySheet *EmplSheet = reinterpret_cast<CPropertySheet *>(GetParent());

    EmplSheet->SetWizardButtons(PSWIZB_BACK | PSWIZB_NEXT);

    return CPropertyPage::OnSetActive();
}
```

4. Once more fire the OnSetActive event of the CEmplSurveyclass and implement it as follows:

```
BOOL CEmplSurvey::OnSetActive()
{
    // TODO: Add your specialized code here and/or call the base class
    CPropertySheet *EmplSheet = reinterpret_cast<CPropertySheet *>(GetParent());

    EmplSheet->SetWizardButtons(PSWIZB_BACK | PSWIZB_FINISH);

    return CPropertyPage::OnSetActive();
}
```

5. Test the application



6. Close it and return to MSVC

15.6 Bitmap Buttons

15.6.1 Overview

A bitmap button is a button that display a picture or a picture and text on its face. This is usually intended to make the button a little explicit. There are two ways you can create a bitmap button: with or without an existing resource identifier.

A bitmap button is created using the **CBitmapButton** class, which is derived from **CButton**, which in turn is a descendent of the **CWnd** class.

Because the bitmap button is in fact a customized version of a button, you must first declare a variable or pointer of it:

```
CBitmapButton *btnMap = new CBitmapButton;
```

Using the variable or pointer, you can call the **Create()** method to formally create the button. Here is an example:

```
BOOL CDialog6Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	. . .

	// TODO: Add extra initialization here
	CBitmapButton *btnMap = new CBitmapButton;

	btnMap->Create(NULL, WS_CHILD | WS_VISIBLE,
		     CRect(10,10,100,100), this, IDC_BTN_NEW);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

15.6.2 Bitmap Button Implementation

Creating a bitmap button without an identifier is considered creating it from scratch. Creating a bitmap button using an existing identifier is more practical because you can easily use the messages of such a button and refer to it in code. Therefore, before creating a bitmap button, you can first add a button to a dialog box and specify its identifier. For a button to display a picture, it must have the BS_OWNERDRAW style. This means that you should check or set to True the Owner Draw option.

Visually, the most appealing aspect of a bitmap button is the picture it displays. The bitmap can be designed following the same tools and instructions we reviewed for the icons. To give a 3-D or realistic appearance to the button, you should mimic the focus effect when designing the bitmaps.

Therefore, you should create one to four bitmaps for the control. Here is how they would work:

?? If you create one bitmap, it would be used for all clicks of the button. Since the button would always appear the same, you should avoid using only one picture

?? If you create two bitmaps, the first would display as the regular picture for the button. That is, when the button is not being clicked. The second would be used when the button is clicked and the mouse is down on the button

?? If you create three bitmaps, the first would display when the button is not being accessed and another control has focus. The second picture would display when the user is clicking the button and as long as the mouse is down on the button. The third would be used when the button has focus but it is not being used

?? If you create four bitmaps, the first would display when the button is not being accessed and another control has focus. The second picture would display when the user is clicking the button and as long as the mouse is down on the button. The third would be used when the button has focus but it is not being used or pressed. The fourth would be used when the button is disabled

With these options, to use a bitmap button to its fullest, you should strive to provide four bitmaps. After creating the bitmaps, you can load them into the button. This is done using the **LoadBitmaps()** method. It comes in two versions as follows:

```
BOOL LoadBitmaps(LPCTSTR lpszBitmapResource,
                 LPCTSTR lpszBitmapResourceSel = NULL,
                 LPCTSTR lpszBitmapResourceFocus = NULL,
                 LPCTSTR lpszBitmapResourceDisabled = NULL );
BOOL LoadBitmaps(UINT nIDBitmapResource,
                 UINT nIDBitmapResourceSel = 0,
                 UINT nIDBitmapResourceFocus = 0,
                 UINT nIDBitmapResourceDisabled = 0 );
```

Each version takes four arguments. The first version uses the bitmaps as string resources while the second version uses resource identifiers.

As stated above, you can use one to four bitmaps. The first argument must specify the first or default bitmap and is required. The second argument is the name or identifier of the bitmap that would be used when the button is selected. The third is used when the button has focus. The last bitmap is used when the button is disabled. Here is an example:

```
BOOL CDialog6Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	. . .

	// TODO: Add extra initialization here
	CBitmapButton *btnMap = new CBitmapButton;

	btnMap->Create(NULL, WS_CHILD|WS_VISIBLE|BS_OWNERDRAW,
		     CRect(10,10,100,100), this, IDC_BTN_NEW);
	btnMap->LoadBitmaps(IDB_BMP_BTN1, IDB_BMP_BTN2, IDB_BMP_BTN3,
IDB_BMP_BTN4);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

It is very likely that you may not be able to design the button on one hand and the bitmaps on the other hand at exactly the same dimensions. To adjust these measures, the **CBitmapButton** class is equipped with the **SizeToContent()** method. Its syntax is:

```
void SizeToContent();
```

When this method is called on the button, it resizes it to the size of the bitmaps. If one bitmap is larger and/or taller than the others, and if you had loaded more than one, this method would resize the button to the larger or largest and taller or tallest bitmap. For this reason, always make sure that you design bitmaps that have the same dimension. Here is an example of using this method:

```
BOOL CDialog6Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	. . .

	// TODO: Add extra initialization here
	CBitmapButton *btnMap = new CBitmapButton;

	btnMap->Create(NULL, WS_CHILD|WS_VISIBLE|BS_OWNERDRAW,
	                  CRect(10,10,100,100), this, IDC_BTN_NEW);
	btnMap->LoadBitmaps(IDB_BMP_BTN1, IDB_BMP_BTN2,
	                          IDB_BMP_BTN3, IDB_BMP_BTN4);
	btnMap->SizeToContent();

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

As you may realize, if you create a bitmap button strictly using this approach, it may not work. The suggestion is to "subclass" your button class so the messages sent to the bitmap button would be applied effectively. The function used to do this is the **CWnd::SubclassDlgItem()** method and its syntax is:

```
BOOL SubclassDlgItem( UINT nID, CWnd* pParent );
```

The first argument is the resource ID of the existing button. The second argument is the control that is hosting the button; this is usually the dialog box but it can be another control container.

Practical Learning: Creating Bitmap Buttons

1. Continue from the Associates application.
   To add a help button, in the constructor of the CCompanyForms class, add the **PSP_HASHELP** flag as follows:

```
CCompanyForms::CCompanyForms()
	: CPropertyPage(CCompanyForms::IDD)
{
	m_psp.dwFlags |= PSP_HASHELP;
}
```

2. From the resources that accompany this book, import the following bitmaps and change their IDs as follows

File	ID	File	ID
canceldis .bmp	IDB_CANCEL_DIS	cancelfoc.bmp	IDB_CANCEL_FOC
cancelnrm.bmp	IDB_CANCEL_NRM	cancelsel.bmp	IDB_CANCEL_SEL
helpdis.bmp	IDB_HELP_DIS	helpfoc.bmp	IDB_HELP_FOC
helpnrm.bmp	IDB_HELP_NRM	helpsel.bmp	IDB_HELP_SEL
okdef.bmp	IDB_OK_DEF	okdis.bmp	IDB_OK_DIS
okfoc.bmp	IDB_OK_FOC	oksel.bmp	IDB_OK_SEL

3. In the header of the CCompanySheet class, declare three CBitmapButton variables as follows:

```
private:
    CBitmapButton btnBmpOK, btnBmpCancel, btnBmpHelp;
};
```

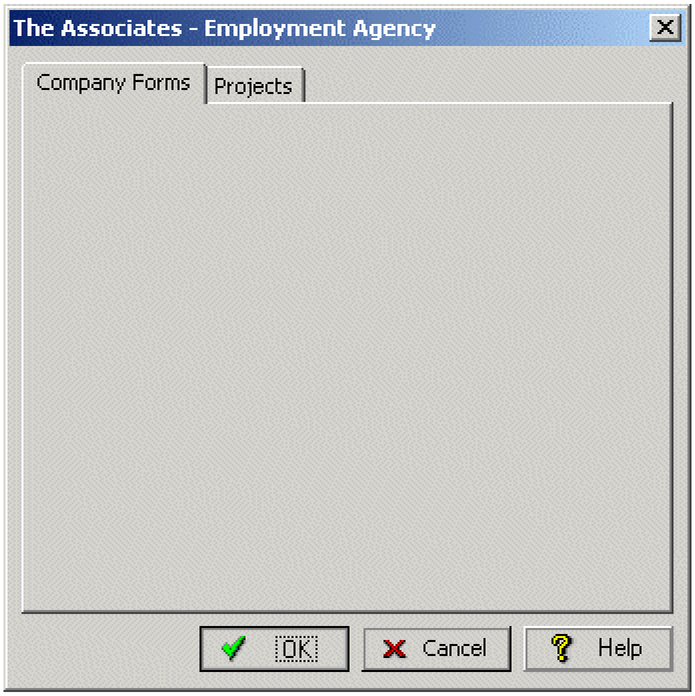
4. Use the constructor to load the bitmaps of each button:

```
CCompanySheet::CCompanySheet(UINT nIDCaption, CWnd* pParentWnd, UINT
iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
    m_psh.dwFlags |= PSH_NOAPPLYNOW;
    AddPage(&Forms);
    AddPage(&Projects);

    btnBmpOK.LoadBitmaps(IDB_OK_DEF, IDB_OK_SEL,
                    IDB_OK_FOC, IDB_OK_DIS);
    btnBmpCancel.LoadBitmaps(IDB_CANCEL_NRM, IDB_CANCEL_SEL,
                             IDB_CANCEL_FOC, IDB_CANCEL_DIS);
    btnBmpHelp.LoadBitmaps(IDB_HELP_NRM, IDB_HELP_SEL,
                           IDB_HELP_FOC, IDB_HELP_DIS);
}
```

5. To customize the existing buttons, implement the OnInitDialog event of the CCompanySheet class as follows (most of the code is used because the buttons were already created by default; this means that we have to manually change their style):

```
BOOL CCompanySheet::OnInitDialog()
{
    BOOL bResult = CPropertySheet::OnInitDialog();

    // TODO:  Add your specialized code here
    // We need a handle to each
    CButton *btnOK, *btnCancel, *btnHelp;

    // Get a handle to each of the existing buttons
    btnOK     = reinterpret_cast<CButton *>(GetDlgItem(IDOK));
    btnCancel = reinterpret_cast<CButton *>(GetDlgItem(IDCANCEL));
    btnHelp   = reinterpret_cast<CButton *>(GetDlgItem(IDHELP));

    // Get the style of the button(s)
    LONG GWLOK = GetWindowLong(btnOK->m_hWnd, GWL_STYLE);
    LONG GWLCancel = GetWindowLong(btnCancel->m_hWnd, GWL_STYLE);
    LONG GWLHelp = GetWindowLong(btnHelp->m_hWnd, GWL_STYLE);

    // Change the button's style to BS_OWNERDRAW
    SetWindowLong(btnOK->m_hWnd, GWL_STYLE, GWLOK | BS_OWNERDRAW);
    SetWindowLong(btnCancel->m_hWnd, GWL_STYLE, GWLCancel |
BS_OWNERDRAW);
    SetWindowLong(btnHelp->m_hWnd, GWL_STYLE, GWLHelp |
BS_OWNERDRAW);

    // Subclass each button
    btnBmpOK.SubclassDlgItem(IDOK, this);
    btnBmpCancel.SubclassDlgItem(IDCANCEL, this);
    btnBmpHelp.SubclassDlgItem(IDHELP, this);

    return bResult;
}
```

6. Test the application:



7. After using it, close it and return to MSVC

Chapter 16: Text-Based Controls

- ✍ Labels
- ✍ Edit Controls
- ✍ Rich Edit Controls

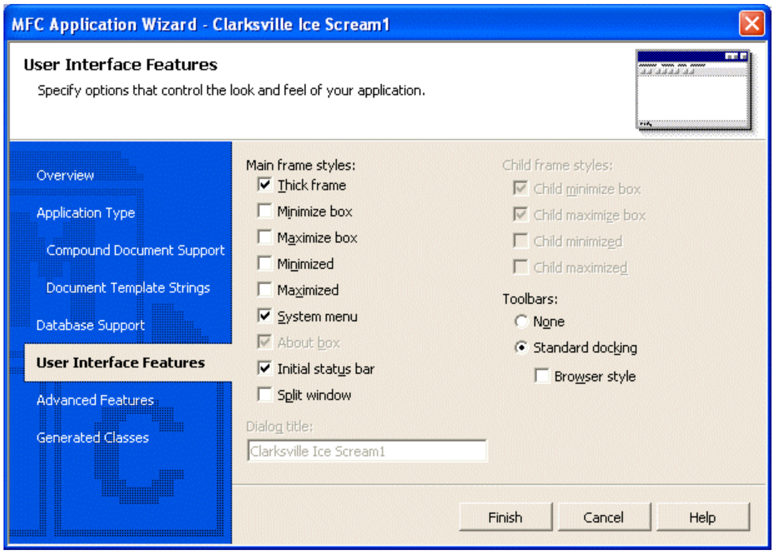
16.1 Labels

16.1.1 Overview

A label is a static control that serves as a guide to the user. It displays text that the user cannot change but can read to get information about a message from the programmer or an indication about another control on the dialog box or form. Most controls on the form or dialog box are not explicit at first glance and the user would not know what they are used for. Therefore, you can assign a label to the control as a help to the user.

Practical Learning: Creating a Label-Based Application

1. Start a new MFC Application named **Clarksville Ice Scream1**
2. Set the Application Type to Single Document
3. Remove the check mark of Printing and Print Preview
4. Remove the Minimize Box and the Maximize Box



User Interface Features

Specify options that control the look and feel of your application.

Main frame styles:

☑ Thick frame
□ Minimize box
□ Maximize box
□ Minimized
□ Maximized
☑ System menu
☑ About box
☑ Initial status bar
□ Split window

Dialog title: Clarksville Ice Scream1

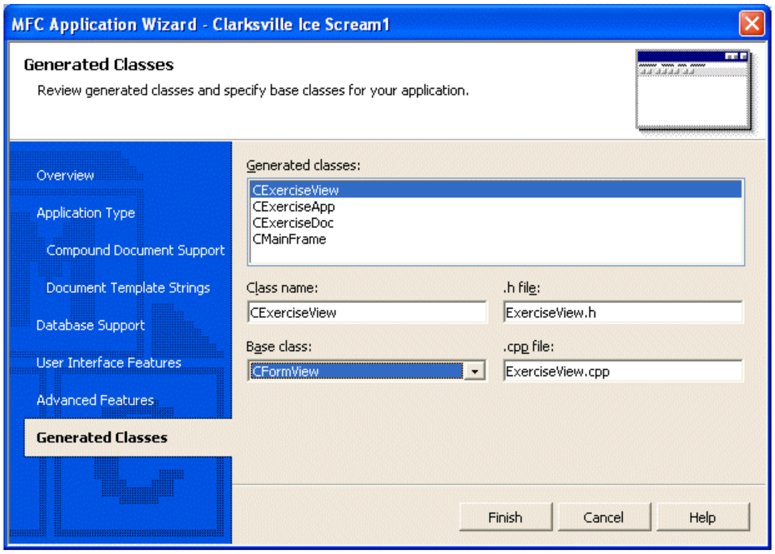
Child frame styles:

☑ Child minimize box
☑ Child maximize box
□ Child minimized
□ Child maximized

Toolbars:

○ None
● Standard docking
□ Browser style

5. Change the view Class Name to **CExerciseView**
6. Change its header file to **Exercise.h** and its source file to **Exercise.cpp**
7. Set the Base Class to **CFormView**



Generated Classes

Review generated classes and specify base classes for your application.

Generated classes:

CExerciseView
CExerciseApp
CExerciseDoc
CMainFrame

Class name: CExerciseView

.h file: ExerciseView.h

Base class: CFormView

.cpp file: ExerciseView.cpp

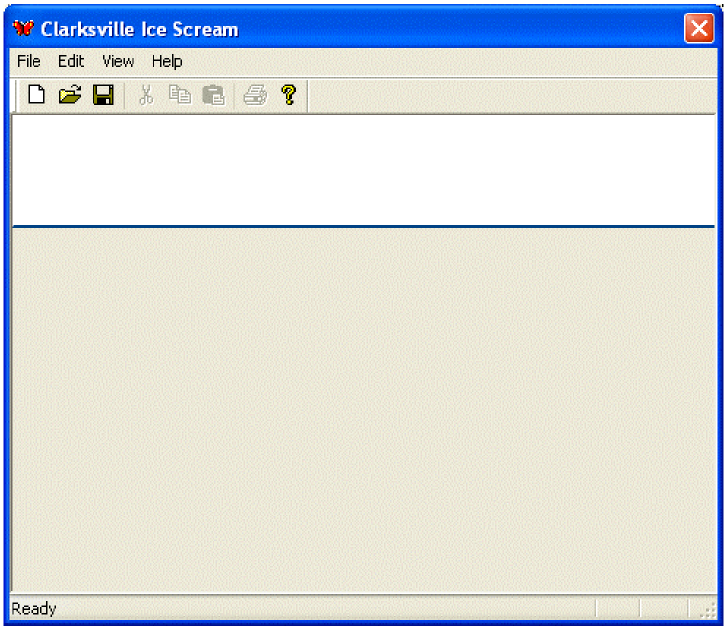
4. Click Finish.
5. If you are using MSVC 7, in the Resource View, expand the Dialog folder
   Click the TODO line and press Delete
6. In the Class View, click the view node and, in the Properties window, click the Overrides button
7. Click OnDraw to display its combo box. Click the arrow and select <Add> OnDraw
8. Implement it as follows:

```
 void CExerciseView::OnDraw(CDC* pDC)
{
	// TODO: Add your specialized code here and/or call the base class
	CBrush BrushWhite(RGB(255, 255, 255));
	CPen   PenWhite(PS_SOLID, 1, RGB(255, 255, 255));
	CPen   PenNavy(PS_SOLID, 2, RGB(0, 64, 128));
	CRect Recto;

	// We need the current width of the client area
	GetClientRect(&Recto);

	// Select a color for the brush and the pen
	CPen *pnOld = pDC->SelectObject(&PenWhite);
	CBrush *brOld = pDC->SelectObject(&BrushWhite);
	// Draw a rectangle
	pDC->Rectangle(0, 0, Recto.Width(), 68);

	// Draw a (heavy) line under the rectangle
	pnOld = pDC->SelectObject(&PenNavy);
	pDC->MoveTo(0, 68);
	pDC->LineTo(Recto.Width(), 68);

	// Restore the GDI tools
	pDC->SelectObject(pnOld);
	pDC->SelectObject(brOld);
}
```

9. Access the **CMainFrame::PreCreateWindow()** method and change it as follows to remove the unnecessary title:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	if( !CFrameWnd::PreCreat eWindow(cs) )
		return FALSE;
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	cs.style = WS_OVERLAPPED | WS_CAPTION | FWS_ADDTOTITLE
		| WS_THICKFRAME | WS_SYSMENU;
	cs.style &= ~FWS_ADDTOTITLE;

	return TRUE;
}
```

10. Test the application



11. Return to MSVC

16.1.2 Drawn Labels

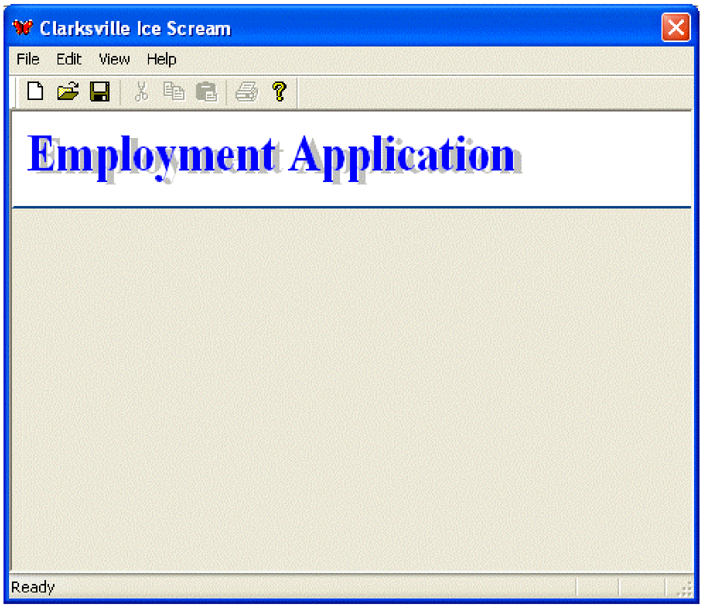
There are two main types of labels you can use in your application. One type of label you can add to a form or dialog box is to draw text. We have learned how to do this in Lesson 7.

Practical Learning: Drawing Labels

1. To draw text used as a label, change the OnDraw() method of the view class as follows:

```
void CExerciseView::OnDraw(CDC* pDC)
{
    // TODO: Add your specialized code here and/or call the base class
    CFont font;
    CBrush BrushWhite(RGB(255, 255, 255));
    CPen   PenWhite(PS_SOLID, 1, RGB(255, 255, 255));
    CPen   PenNavy(PS_SOLID, 2, RGB(0, 64, 128));
    CRect Recto;

    // Create a bold font
    font.CreateFont(42, 14, 0, 0,
                FW_HEAVY, FALSE, FALSE, FALSE, ANSI_CHARSET,
                OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                DEFAULT_QUALITY, DEFAULT_PITCH | FF_ROMAN,
                "Times New Roman");

    CFont *pFont = pDC->SelectObject(&font);
    // We need the current width of the client area
    GetClientRect(&Recto);

    // Create a string label and get its length
    const char *lblEmplApp = "Employment Application";
    int Len = strlen(lblEmplApp);

    GetClientRect(&Recto);

    // Select a color for the brush and the pen
    CPen *pnOld = pDC->SelectObject(&PenWhite);
    CBrush *brOld = pDC->SelectObject(&BrushWhite);
    // Draw a rectangle
    pDC->Rectangle(0, 0, Recto.Width(), 68);

    // Draw a (heavy) line under the rectangle
    pnOld = pDC->SelectObject(&PenNavy);
    pDC->MoveTo(0, 68);
    pDC->LineTo(Recto.Width(), 68);

    // We want the text to be transparent
    pDC->SetBkMode(TRANSPARENT);

    // Draw the label
    pDC->SetTextColor(RGB(192, 192, 192));
    pDC->TextOut(16, 10, lblEmplApp, Len);
    // Draw the label's shadow
    pDC->SetTextColor(RGB(0, 0, 255));
    pDC->TextOut(10, 8, lblEmplApp, Len);

    // Restore the GDI tools
    pDC->SelectObject(pFont);
    font.DeleteObject();
    pDC->SelectObject(pnOld);
    pDC->SelectObject(brOld);
}
```

2. Test the application



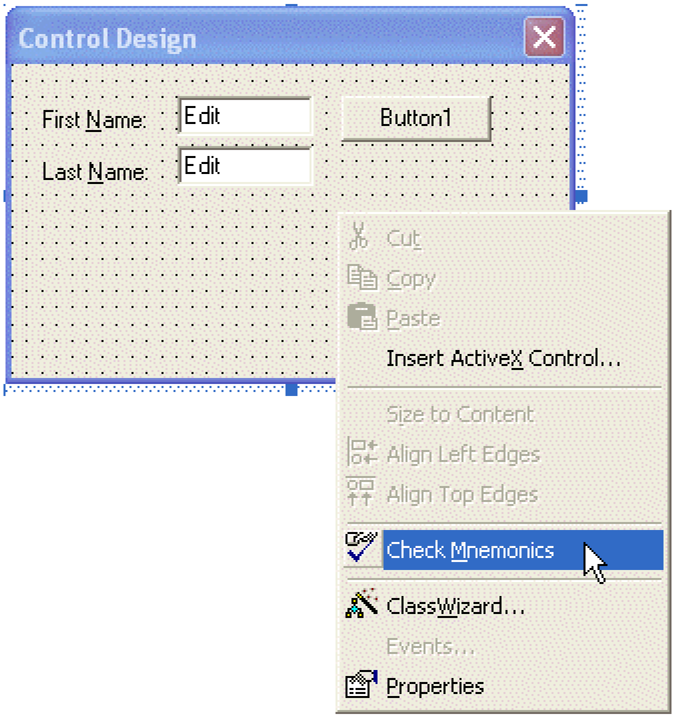
3. Return to MSVC

16.1.3 Static Labels

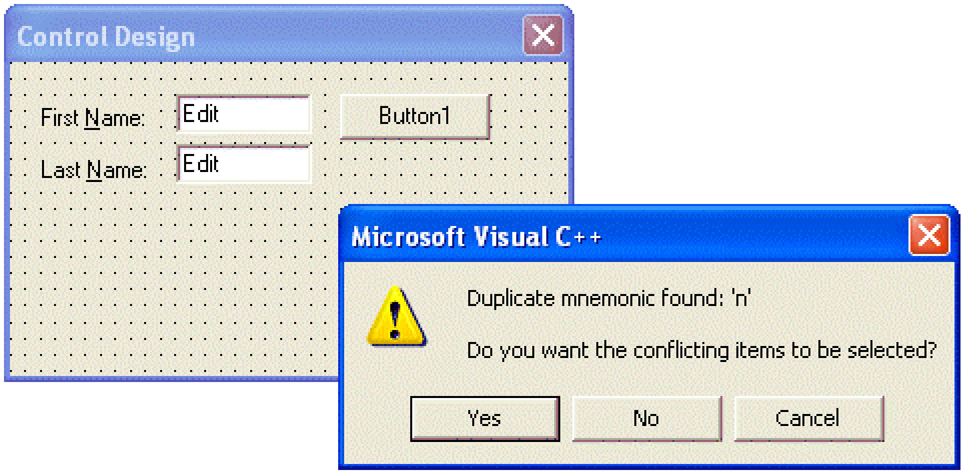
Another type of label you can add to a form or dialog box consis ts of clicking the Static Text button from the Controls toolbar and clicking on the host. The Static Text control is based on the CStatic class.

A user cannot directly change any aspect of a label. For this reason, you will usually not be concerned with a label's IDentifier. In fact, as we will see in this lesson, all controls that are static type have an identifier called IDC_STATIC. If, for any reason, you want to refer to a label in your code, you must first change its identifier from IDC_STATIC to something else.

As stated already, the most important aspect of a label is the text it displays. This is the **Caption** property. If the label is used to identify another control, you can help the user access that control using an indicator called an access key. An access key is a underlined letter on the label so that, when clicked, it gives access to the accompanying control. To create an access key, choose a letter on the label and precede it with the ampersand character "&". Form example L&etter would produce Letter. When there are many access keys on a form or dialog box, you should make sure that no two access keys are the same. That is, you should not use the same letter on various labels, although this is allowed. Because you can forget to follow this rule, after creating the access keys, you can ask Visual Studio to check for access key duplicates. To do this, right-click the form or dialog box and click Check Mnemonics:



If there are duplicate access keys, you would receive a message accordingly:



Duplicate mnemonic found: 'n'

Do you want the conflicting items to be selected?

If you possible, you should correct by changing one of the access keys without using one that exists already.

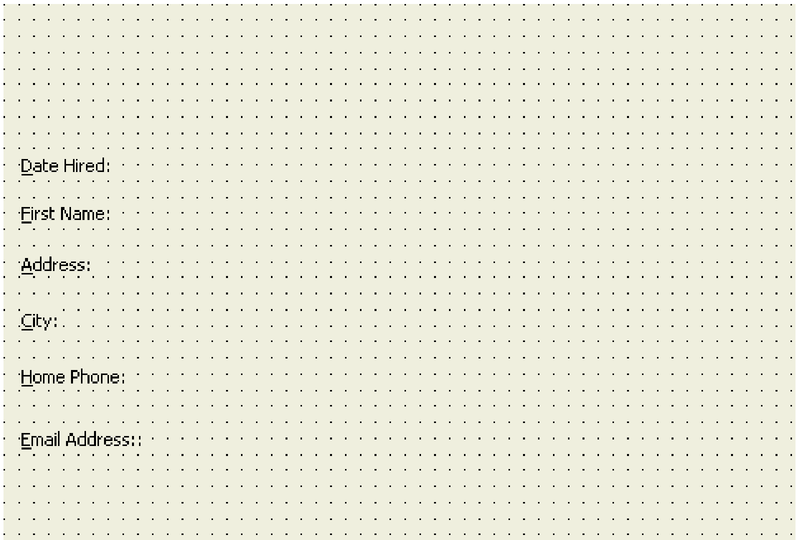
To use the access keys, the user presses Alt and the letter that corresponds to the access key.

The text of the caption follows a line alignment that can assume one of three values: **Left**, **Center**, or **Right**. By default, text of a label is aligned to the left. At design time, to change it, select the desired value using the **Align Text** combo box from the Styles property page of the Properties window.

To apply a fancy appearance to a label, you can use the Extended Styles of the Properties window.

Practical Learning: Using Static Labels

1. From the Dialog folder of the Resource View, double-click the IDD_CLARKSVILLEICESCREAM1_FORM form to display it
2. On the Controls toolbox, click Static Text and click on the form.
3. On the Properties window, click Caption and type &Date Hired:
4. Using the Static Text control, complete the form as follows:



Date Hired:

First Name:

Address:

City:

Home Phone:

Email Address::

5. Test the application and return to MSVC

16.2 Edit Controls

16.2.1 Introduction

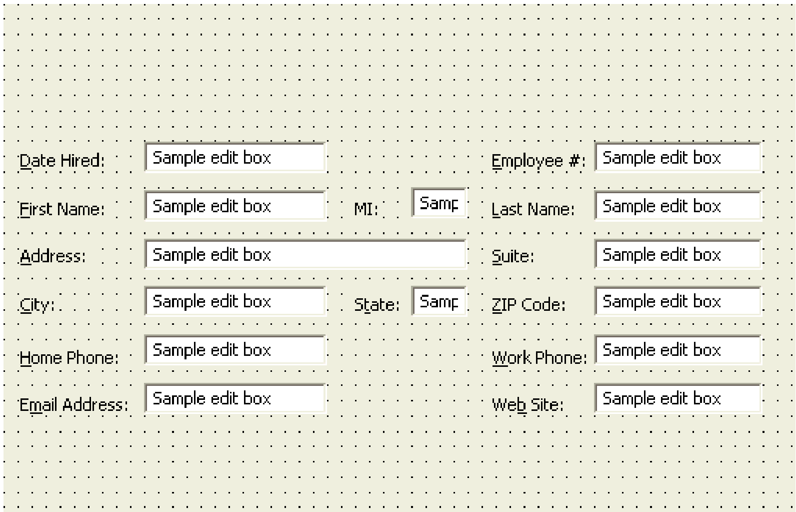
An edit box is a Windows control used to display text or get it from the user. To provide its functionality, the control displays a box, whose background is white by default, surrounded by a black line. If the box is empty, the user may be expected to enter some letters, numbers, or other characters into it. If the box contains some text, the user should be able to edit it. Another edit box may be used to present text to the user without his or her being able to change it. The text that displays in an edit box is referred to as its value.

Like most other controls, the role of an edit box is not obvious at first glance. That is why it should (always) be accompanied by a label that defines its purpose. From the user's standpoint, an edit box is named after the label closest to it. Such a label is usually positioned to the left or the top side of the edit box. From the programmer's point of view, an edit box is a place holder used for various things. For example, you can show or hide it as you see fit.

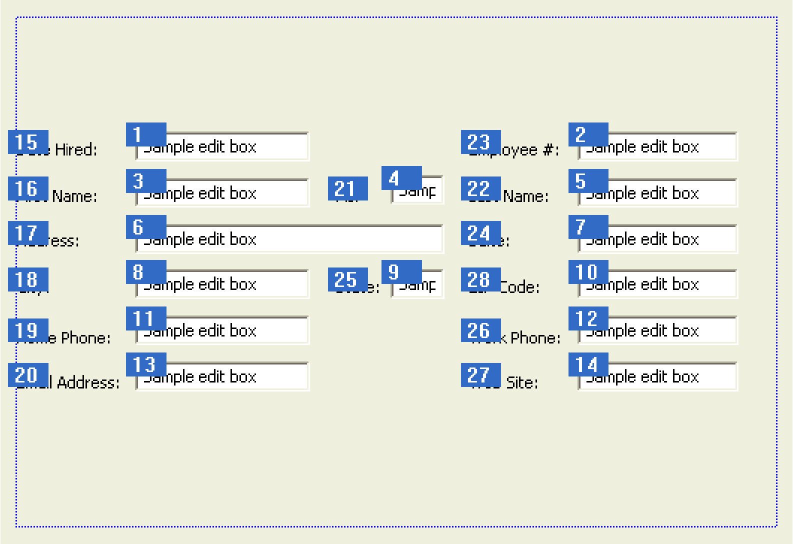
To create an edit box, click the Edit Box button from the Controls toolbox and click the desired area on a form or a dialog box.

Practical Learning: Creating Edit Boxes

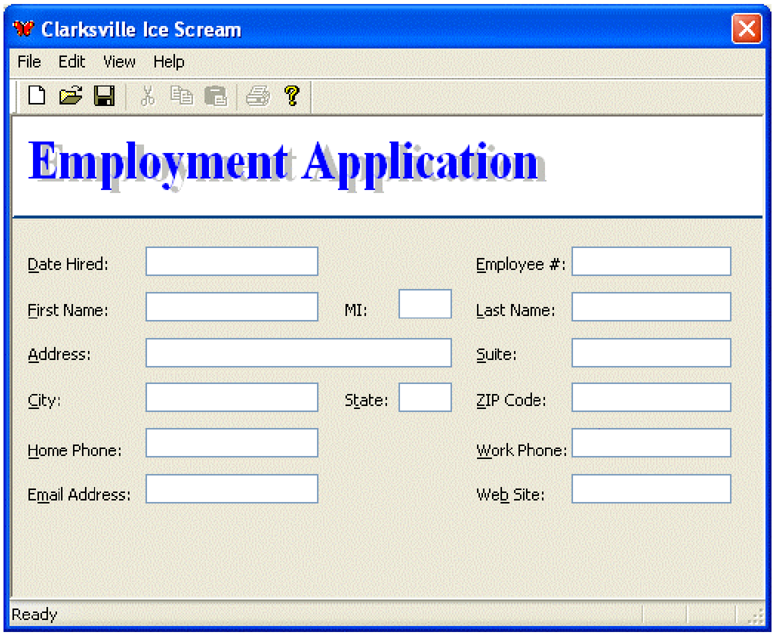
1. Display the form
2. On the Controls toolbox, click the Edit Control button and click on the right side of the Date Hired label
3. Using the Edit Control and additional labels, complete the form as follows:



4. On the main menu, click Layout or Format and click Tab Order
5. Click the Date Hired edit box to place the number 1 in it and arrange the rest of the tab sequence as follows:



6. Test the application



7. Return to MSVC

16.2.2 Edit Control Characteristics

An edit box is a control based on the **CEdit** class. Therefore, to programmatically create an edit box, declare a variable of **CEdit** type using its (default) constructor. To define the characteristics of this control, you can call the **CEdit::Create()** method. Its syntax is:

```
BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);
```

The content of an edit box, that is, its value, is a string. It could be considered as null-terminated constant (**LPCTSTR**) or a **CString** value.

Like every other control added during design, you should give some attention to the identifier of an edit box. The first edit box placed on a form or dialog box receives an identifier of IDC_EDIT1. The second would be identified as IDC_EDIT2, etc. It is highly recommended that you give a meaningful and friendly identifier to each control. For example, if an edit box is used to receive or display an address, you can set its identifier to IDC_ADDRESS. If an edit box is used to enter an email address, you can change its ID to IDC_EMAILADDRESS or IDC_EMAIL_ADDRESS. An identifier should have a maximum of 30 characters. Notice that the ID contains C, which stands for Control.

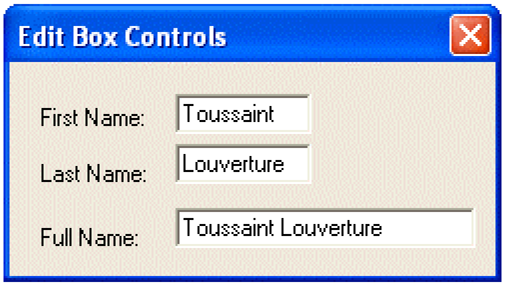
If you plan to access an edit box in your code, you should create a variable for it. The variable can be a **CEdit** or a **CString** object. The difference is that, if you want to access the control as an MFC class, you should create the variable as a **CEdit** object. If you are more interested with the value (as a string) of the control, you should declare the variable as **CString**. Fortunately, you can add two variables for the same control.

Probably the most important characteristic of an edit control for both the programmer and the user is the text it is displaying or that it can display. When you add an Edit control, Visual C++ 6 displays Edit and Visual C++ 7 displays Sample edit box. This text is not part of the control and does not show when the control comes up. It is only an indicator.

If the user wants to change the text of an edit box and if the control allows changes, he or she must first click in the control, which places a caret in the edit box. The caret is a blinking I beam that serves as a reminder that lets the user know what edit control would receive any change made. This means that if the user starts typing, the edit control in which the caret is positioned would display a change in its value. The edit box that has the caret is said to have focus. As mentioned already, the user gives focus to an edit box by clicking it. Remember that if a label that accompanies an edit box has an access key, the user can also give focus to the edit control by pressing the access key. Also remember that you can programmatically give focus to an edit control by calling the **SetFocus()** method.

At any time, you can find out what text is in an edit control and there are various techniques you can use. We saw already how to get a handle to a control by calling the **CWnd::GetDlgItem()** method. After calling this method, you can use the **CWnd::GetWindowText()** method to find out what text an edit box holds. Here is an example:

```
void CEditBoxDlg::CreateName()
{
	CEdit *edtFirstName, *edtLastName, *edtFullName;
	CString FirstName, LastName;
	char FullName[40];

	edtFirstName = reinterpret_cast<CEdit *>(GetDlgItem(IDC_FIRST_NAME));
	edtLastName  = reinterpret_cast<CEdit *>(GetDlgItem(IDC_LAST_NAME));
	edtFullName  = reinterpret_cast<CEdit *>(GetDlgItem(IDC_FULL_NAME));

	edtFirstName->GetWindowText(FirstName);
	edtLastName->GetWindowText(LastName);

	sprintf(FullName, "%s %s", FirstName, LastName);
	edtFullName->SetWindowText(FullName);
}
void CEditBoxDlg::OnLButtonDblClk(UINT nFlags, CPoint point)
{
	// TODO: Add your message handler code here and/or call default
	CreateName();

	CDialog::OnLButtonDblClk(nFlags, point);
}
```



First Name: Toussaint

Last Name: Louverture

Full Name: Toussaint Louverture

Another technique you can use to get the text of an edit control consists of calling the **CWnd::GetDlgItemText()** method. It is provided in two syntaxes as follows:

int GetDlgItemText(int *nID*, LPTSTR *lpStr*, int *nMaxCount*) const;
int GetDlgItemText(int *nID*, CString& *rString*) const;

The *nID* argument is the identifier of the control whose text you want to retrieve. The lpStr or the rString is the returned value of the text in the edit box. It must be provided as a string variable. If you are using a null-terminated variable, pass a third argument as nMaxCount that specifies the maximum length of the string. Here is an example:

```
void CFormView1View::OnButton1()
{
	// TODO: Add your control notification handler code here
	CString Edit1, Edit2, Result;

	GetDlgItemText(IDC_EDIT1, Edit1);
	GetDlgItemText(IDC_EDIT2, Edit2);
	Result = Edit1 + " " + Edit2;

	SetDlgItemText(IDC_EDIT3, Result);
}
```

As mentioned already, an edit box can be used to request information from the user. If you want to prevent the user from changing the value of an edit box, you can make it read-only. This is taken care of by setting the Read-Only property to True or checking its check box. To do this programmatically, call the **CEdit::SetReadOnly()** method.

If an edit box is not “read-only”, that is, if it allows the user to change its value, the user must first give it focus. When an edit box has focus, it displays a blinking caret. By default, the carret is an I beam. If you want to use a different caret, you have various options. You can change the caret from an I beam to a wider or taller gray caret by calling the **CWnd::CreateGrayCaret()** method. Its syntax is:

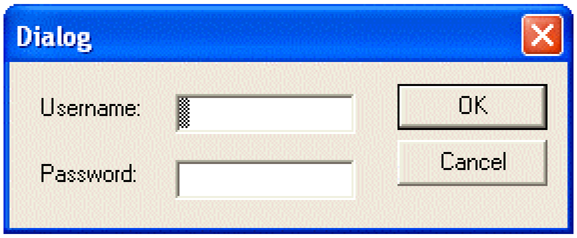
void CreateGrayCaret(int *nWidth*, int *nHeight*);

This method allows you to specify a width and a height for a gray blinking caret. After creating the caret, to display it, call the **CWnd::ShowCaret()** method. Its syntax is:

void ShowCaret( );

Here is an example:

```
BOOL CSolidCaretDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	m_Username.CreateGrayCaret(5, 15);
	m_Username.ShowCaret();

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



The above caret appears gray. If you want the caret to be completely black, you can call the **CWnd::CreateSolidCaret()** method. Its syntax is:

void CreateSolidCaret(int *nWidth*, int *nHeight*);

This method creates a rectangular caret of *nWidth* x *nHeight* dimensions. Here is an example:

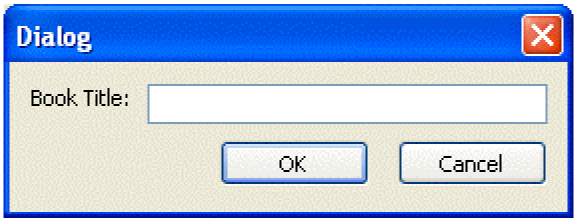
```
BOOL CSolidCaretDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        m_Username.CreateSolidCaret(5, 15);
        m_Username.ShowCaret();

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

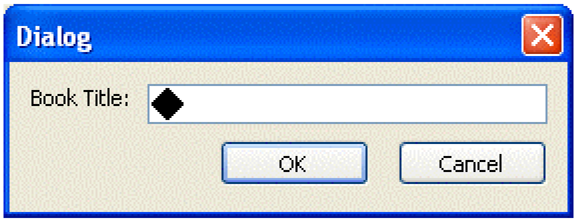
To provide a better designed caret, you can call the **CWnd::CreateCaret()** method. Its syntax is:

void CreateCaret(CBitmap* pBitmap);

Before calling this method, you can first design a bitmap, load it, and then pass it as the *pBitmap* argument. After initializing and loading the bitmap, to display it in the edit box, call the CWnd::ShowCaret() method. Here is an example:

```
BOOL CDialogCaret::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO:  Add extra initialization here
        CEdit *edtBookTitle;
        CBitmap *pBitmap = new CBitmap;

        edtBookTitle = reinterpret_cast<CEdit *>(GetDlgItem(IDC_BOOK_TITLE));

        pBitmap->LoadBitmap(IDB_BMP_CARET);
        edtBookTitle->CreateCaret(pBitmap);
        edtBookTitle->ShowCaret();

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





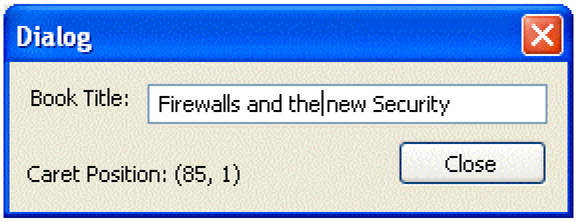
If an edit box is editable and the user starts typing in it, the new characters would display. As the user is typing, the caret moves from left to right (for US English). The user can also move the caret back and forth using the Backspace, Delete, Home, End, or the arrow keys. At any time you can find out the current position of the caret by calling the CWnd::GetCaretPos() method. Its syntax is:

```
static CPoint PASCAL GetCaretPos( );
```

This method retrieves the left (x) and bottom (y) coordinates of the caret in the control that called it. These coordinates represent the member variables of a CPoint that this method returns. The measures are relative to the control and not the control's parent.

Here is an example:

```
void CDialogCaret::CaretPosition(void)
{
	CEdit *edtBookTitle;
	CStatic *stcCaretPos;
	CPoint CrtPos;
	char Msg[40];

	edtBookTitle = reinterpret_cast<CEdit *>(GetDlgItem(IDC_BOOK_TITLE));
	stcCaretPos  = reinterpret_cast<CStatic *>(GetDlgItem(IDC_CARET_POS));

	CrtPos = edtBookTitle->GetCaretPos();
	sprintf(Msg, "Caret Position: (%d, %d)", CrtPos.x, CrtPos.y);

	stcCaretPos->SetWindowText(Msg);
}

void CDialogCaret::OnLButtonDown(UINT nFlags, CPoint point)
{
	// TODO: Add your message handler code here and/or call default
	CaretPosition();

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



If want to hide the characters that display in an edit box, you can set the Password property to True. To do this programmatically, add the **ES_PASSWORD** style to the edit control. This style makes the edit control displays each character, or changes each one of its characters into an asterisk. If you prefer another symbol for the password style, call the **CEdit::SetPassword()** method. Its syntax is:

```
void SetPasswordChar(TCHAR ch);
```

This method takes as argument the character that you want to use.

If an edit box is configured to display an asterisk character and you want to find out what that character is, call the CEdit::GetPasswordChar(). Its syntax is:

```
TCHAR GetPasswordChar() const;
```

This method takes no argument but returns the symbol used to mask the characters of a password-configured edit box.

By default, an edit box is configure to display or receive text of any alphabetic and non-alphabetic character. The alphabetical letters are received by their case and kept like that, uppercase and lowercase. If you want to convert an edit box' characters to uppercase, set the Uppercase property to True or programmatically add the **ES_UPPERCASE** style. In the same way, to convert the characters to lowercase, either at design time set the Lowercase property to True or programmatically add the **ES_LOWERCASE** style. The non-alphabetical characters are not treated by their case.

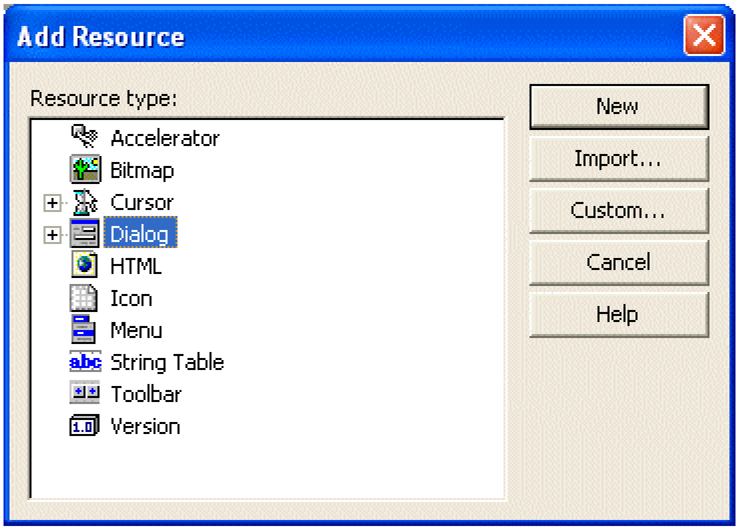
If you want, you can configure the edit box to allow only numeric characters. This is done by setting the Number property to True.

By default, the characters entered in an edit box start their positioning to the left, which is the default for regular text. You can align its value to the center or the right by selecting the desired value from the Align Text combo box. Any of these three alignment modes can also be set programmatically by adding either the **ES_LEFT**, the **ES_CENTER**, or the **ES_RIGHT** style.

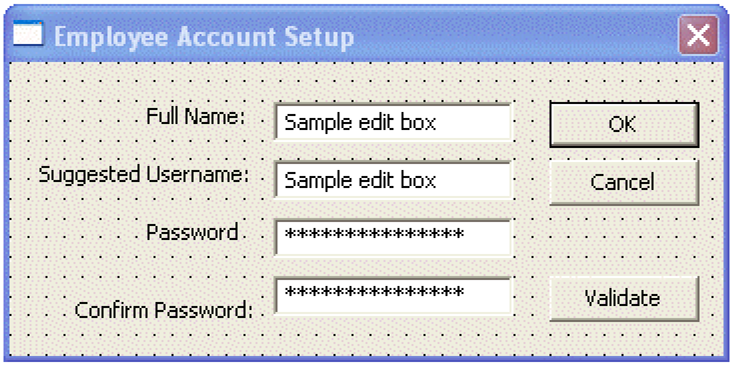
As mentioned above, the value of an edit box is of high interest to you and your users. If you want to retrieve the value of an edit box, call the **CWnd::GetWindowText()** method. If you want to display or change the text of an edit box, call the **CWnd::SetWindowText()** method. As we have seen in the past, SetWindowText() takes a constant pointer to null-terminated string (LPCTSTR) and displays its value in the edit. This method is very convenient if you had add a CEdit variable to your edit control. If you have added the variable as a **CString**, you can use the CString::Format() method to change or format the value to display in an edit box.

Practical Learning: Configuring Edit Boxes

1. The Clarksville Ice Scream1 application should still be opened
   Using the Add Resource dialog box, add a new dialog:

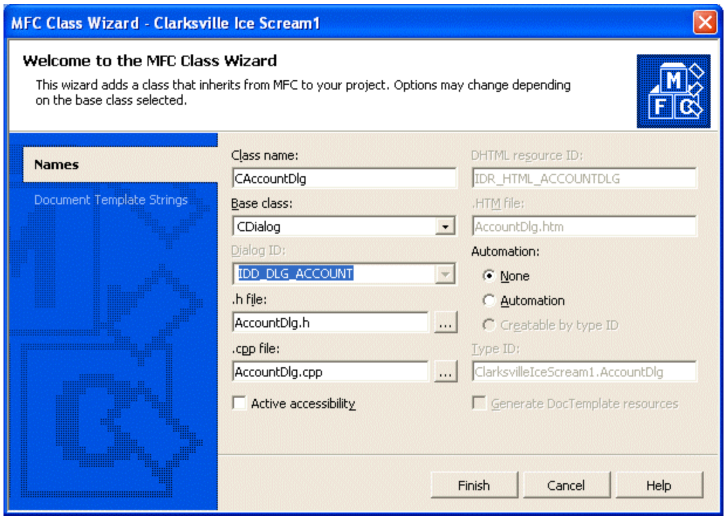


2. Design the dialog box as follows:



Control	ID	Caption	Other Properties
Dialog Box	IDD_DLG_ACCOUNT	Employee Account Setup	
Static Text		&Full Name	
Edit Box	IDC_FULL_NAME		Read-Only = True
Static Text		&Suggested Username:	
Edit Box	IDC_USERNAME		Lowercase = True
Static Text		&Password	
Edit Box	IDC_PASSWORD		Password = True
Static Text		&Confirm Password	
Edit Box	IDC_CONF_PASSWORD		Password = True
Button	IDC_BTN_VALIDATE	&Validate	

3. Create a class for the dialog box and name it **CAccountDlg**



MFC Class Wizard - Clarksville Ice Scream1

Welcome to the MFC Class Wizard

This wizard adds a class that inherits from MFC to your project. Options may change depending on the base class selected.

Class name: CAccountDlg

Base class: CDialog

Dialog ID: IDD_DLG_ACCOUNT

.h file: AccountDlg.h

.cpp file: AccountDlg.cpp

☐ Active accessibility

DHTML resource ID: IDR_HTML_ACCOUNTDLG

.HTM file: AccountDlg.htm

Automation:

● None

○ Automation

○ Creatable by type ID

Type ID: ClarksvilleIceScream1.AccountDlg

☐ Generate DocTemplate resources

4. Display the new dialog box, right-click the Suggested Username edit box and click Add Variable

5. Set the Variable Name to **m_StrUsername**

6. Set its **Category** to **Value**

7. Set the **Maximum** Characters to **5**

8. Add control and value variables to the controls as follows:

Control	ID	Control Variable	Value Variable	Max Chars
Edit Box	IDC_FULL_NAME		m_StrFullName	
Edit Box	IDC_USERNAME	m_Username	m_StrUsername	5
Edit Box	IDC_PASSWORD	m_Password	m_StrPassword	14
Edit Box	IDC_CONF_PASSWORD		m_StrConfPassword	

9. Display the main form.

10. Add a button under the Web Site edit box and its **Caption** to **Account Setu&p**

11. Change the Identifiers of the edit boxes and the button from left to right and from top to bottom as follows: IDC_DATE_HIRED, IDC_EMPLOYEE_NBR, IDC_FIRST_NAME, IDC_MI, IDC_LAST_NAME, IDC_ADDRESS, IDC_SUITE, IDC_CITY, IDC_STATE, IDC_ZIP_CODE, IDC_HOME_PHONE, IDC_WORK_PHONE, IDC_EMAIL_ADDRESS, IDC_WEB_SITE, IDC_BTN_ACCOUNT

12. Set the **Disabled** property of the Account Setup button to True

13. Set the **Uppercase** of the MI edit box to True

14. Add some CString value and a CButton Control variables to the controls as follows:

Control	Identifier	Value Variable	Control Variable	Max Chars
Edit Box	IDC_FIRST_NAME	m_StrFirstName		
Edit Box	IDC_MI	m_StrMI		1
Edit Box	IDC_LAST_NAME	m_StrLastName		
Button	IDC_BTN_ACCOUNT		m_BtnAccount	

15. Add a **BN_CLICKED** Event Handler to the button associated with the view class and implement it as follows:

```
void CExerciseView::OnBnClickedBtnAccount()
{
    // TODO: Add your control notification handler code here
    // Take over from the application
    UpdateData();

    // Get the full name by appending the last name to the first
    CString FullName = m_StrFirstName + " " + m_StrLastName;
    // Create a temporary username made of the
    // first letter of the first name + the last name
    CString TempUsername = m_StrFirstName.Left(1) + m_StrLastName;
    // Make an attempt to create a username made of 5 letters
    CString SuggestedUsername = TempUsername.Left(5);

    // What if the attempted username is less than 5 characters
    if( TempUsername.GetLength() < 5 )
    {
            // Since the username is < 5 characters,
            // add some digits to make it valid
            SuggestedUsername = TempUsername + "1234";
            // now that we have a username with at least 5 letters,
            // Retrieve only the first 5 to create a new username
            SuggestedUsername = SuggestedUsername.Left(5);
    }

    // It is time to access the Employee Account Setup dialog box
    CAccountDlg Dlg;

    // Transfer the the full name to the Employee Account Setup dialog box
    Dlg.m_StrFullName.Format("%s", FullName);
    Dlg.m_StrUsername.Format("%s", SuggestedUsername);

    // Display the Employee Account Setup dialog box
    Dlg.DoModal();

    // Hand the operations back to the application
    UpdateData(FALSE);
}
```

16. On top of the current source file, include the AccountDlg.h header file

```
#include "stdafx.h"
#include "Clarksville Ice Scream1.h"

#include "ExerciseDoc.h"
#include "ExerciseView.h"
#include "AccountDlg.h"
```

17. Test the application and return to MSVC

18. Open the Geometry application. If you do not have it, open one the Geometry4 application that accompanies this book.

19. Open each IDD_CIRCULAR dialog

20. Click the top Radius edit box. Press and hold Ctrl. Then click each one of the other boxes to select them. Then release Ctrl

21. On the Properties window, set the Align Text property to Right

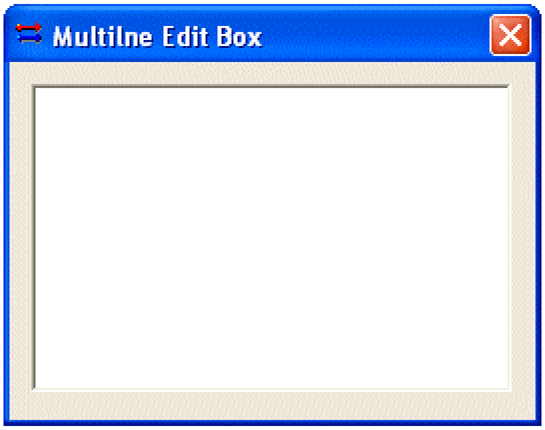
22. In the same way, set the Align Text property of the edit boxes of the IDD_ G3D and the IDD_QUADRILATERAL dialog boxes to Right

23. Test the application and return to MSVC

16.2.3 Multiline Edit Boxes

By default, an edit box is used to display a single line of text. An edit box is referred to as multiline when it can display its text on more than one line.

To create a multiline edit box, use the same Edit control as the above single line Edit box. To make this a multiline edit box, set its Multiline property to True. To do this at run time, add the **ES_MULTILINE** style to the CEdit. At design time, to make it obvious that the control can display more than one line of text, heighten it:

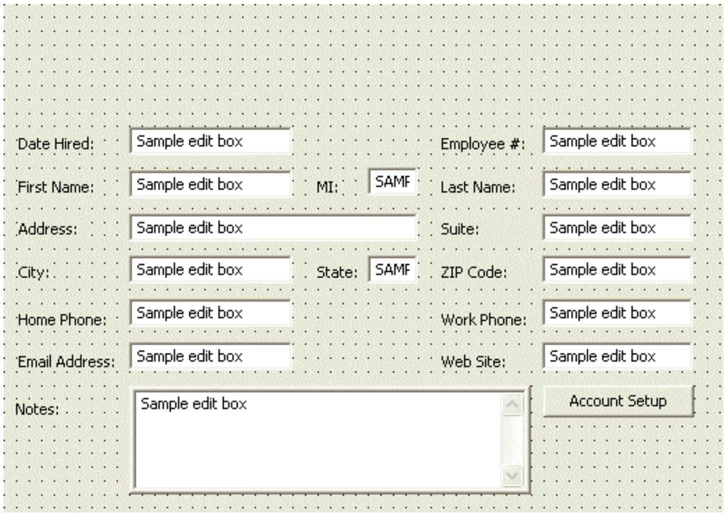


If the user is typing text in an edit control and press Enter, the control, usually a button, that is the default would be activated. This feature is valuable for a single line edit box. If you are creating a multiline edit box, you should allow the user to press Enter while entering text into the control. This would prevent the default button from being activated and would transfer the caret to the next line. To provide this functionality, add the **ES_WANTRETURN** style or, at design time, set the Want Return property to True.

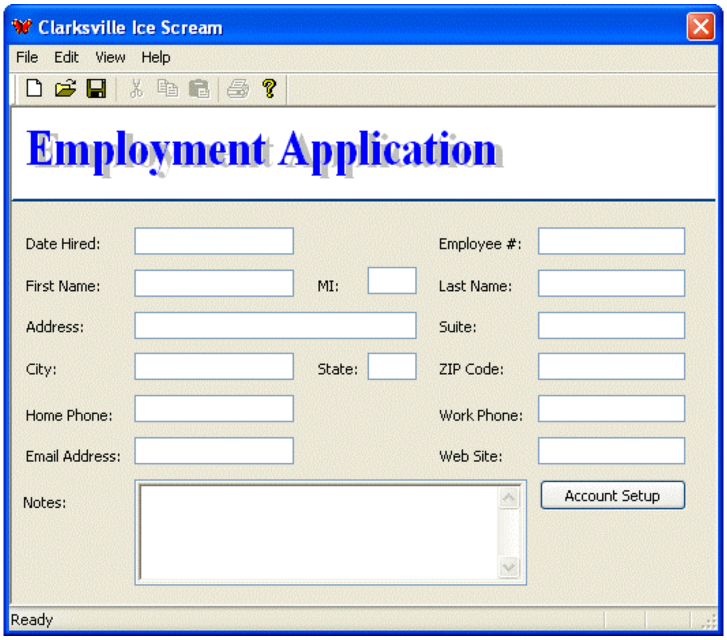
If the text of the edit control is longer than the edit control can display at one time, you should equip it with scroll bars. To do this, at design time, set the Horizontal and/or the Vertical Scroll properties to True. At run time, add the **WS_HSCROLL** and/or the **WS_VSCROLL** properties.

Practical Learning: Designing Edit Boxes

1. Reopen the Clarksville Ice Scream1 application and display the form
2. Add an Edit Control at the bottom section of the form
3. On the Properties window, check the Multiline check box or set its value to True
4. Check the Want Return check box or set its value to True
5. Check the Vertical Scroll check box or set its value to True
6. Check the Modal Frame check box or set its value to True:



7. Test the application



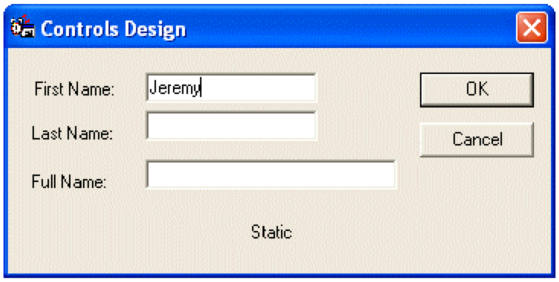
8. Close the application and return to MSVC

16.2.4 Edit Control Messages

To manage its role on a form or dialog box, an edit control is equipped with some messages known as notifications. These notifications messages are:

ON_EN_SETFOCUS: The event name for this notification is called OnSetFocus. It is sent when an edit box receives focus. This happens when the user clicks the edit box or after previously pressing Tab, to give focus to the control:

```
BEGIN_MESSAGE_MAP(CDialog3aDlg, CDialog)
	//{{AFX_MSG_MAP(CDialog3aDlg)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_EN_SETFOCUS(IDC_LAST_NAME, OnSetFocusLastName)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

…

void CDialog3aDlg::OnSetFocusLastName()
{
	// TODO: Add your control notification handler code here
	m_Message.SetWindowText("The First Name has focus");
}
```

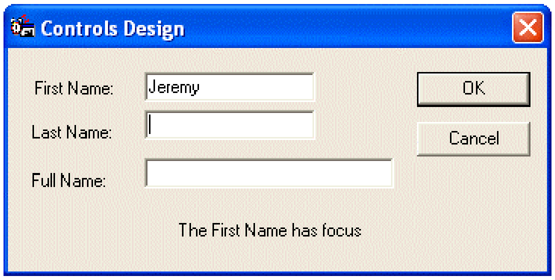


First Name: Jeremy

Last Name:

Full Name:

Static



First Name: Jeremy

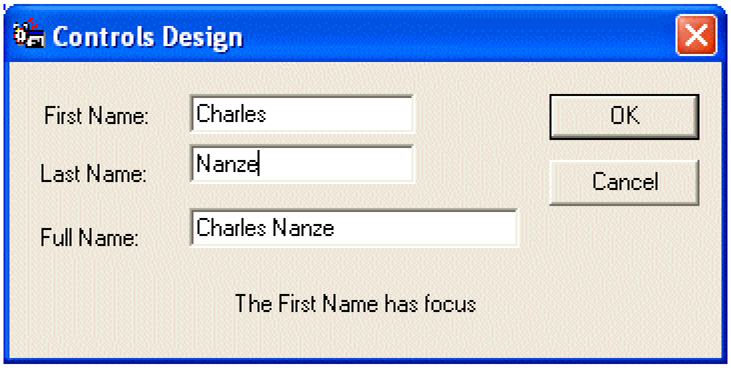
Last Name:

Full Name:

The First Name has focus

ON_EN_CHANGE: The OnChange event of this notification occurs as the user is typing text in the edit control. This happens as the user is changing the content of an edit control, which sends a message that the content of the edit box has changed. You can use this event to check, live, what the user is doing in the edit box. For example, if you create a dialog box or a form with a first and last names edit boxes, you can use another edit box to display the full name. You can implement the OnChange events of the edit boxes as follows:

```
BEGIN_MESSAGE_MAP(CDialog3aDlg, CDialog)
	//{{AFX_MSG_MAP(CDialog3aDlg)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_EN_SETFOCUS(IDC_LAST_NAME, OnSetFocusLastName)
	ON_EN_CHANGE(IDC_FIRST_NAME, OnChangeFirstName)
	ON_EN_CHANGE(IDC_LAST_NAME, OnChangeLastName)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

…

void CDialog3aDlg::OnChangeFirstName()
{
	// TODO: Add your control notification handler code here
	m_FirstName.GetWindowText(m_strFirstName);
	m_LastName.GetWindowText(m_strLastName);
	CString FullName  = m_strFirstName + " " + m_strLastName;

	m_FullName.SetWindowText(FullName);
}

void CDialog3aDlg::OnChangeLastName()
{
	// TODO: Add your control notification handler code here
	m_FirstName.GetWindowText(m_strFirstName);
	m_LastName.GetWindowText(m_strLastName);
	CString FullName  = m_strFirstName + " " + m_strLastName;

	m_FullName.SetWindowText(FullName);
}
```



First Name: Charles

Last Name: Nanze

Full Name: Charles Nanze

The First Name has focus

ON_EN_UPDATE: The **OnUpdate()** event is sent after the content of an edit box has changed but before the text is formally displayed.

ON_EN_MAXTTEXT: When adding a variable for the edit control, you can specify the maximum allowable number of characters the user can enter. If the user attempts exceeding this maximum, an **OnMaxtext()** event is sent. You can catch this event to let the user know about the problem. If you set up the Maximum Characters fine, you do not need to perform any "if" or "while" checking. The edit control would do it itself.

ON_EN_ERRSPACE: The **OnErrSpace()** event is event is sent if the compiler encountered a problem when attempting to allocate memory space to the control.

ON_EN_KILLFOCUS: The **OnExit()** event occurs when the control loses focus. In the case of an edit box, this could happen if the control has focus and the user presses Tab; the edit box would lose focus.

Practical Learning: Using Edit Box Notifications

1. The Clarksville Ice Scream1 application should still be opened
   Display the Employee Account Setup dialog box
2. If you are using MSVC 6, display the Message Maps property page of the ClassWizard dialog box. Then click IDC_USERNAME
   If you are using MSVC 7, right-click the Suggested Username edit box and click Add Event Handler
3. Add an event handler for the **EN_MAXTEXT** message and click either Edit Code or Finish
4. Implement the event as follows:

```
void CAccountDlg::OnEnMaxtextUsername()
{
	// TODO: Add your control notification handler code here
	MessageBox("A username must have a maximum of 5 characters");
}
```

5. Add an event handler to the Validate button and implement it as follows:

```
void CAccountDlg::OnBnClickedBtnValidate()
{
    // TODO: Add your control notification handler code here
    UpdateData();

    // In case the user decided to change the username,
    // make sure the username is = 5 characters
    if( m_StrUsername.GetLength() == 5 )
    {
            // Since the username is 5 characters, check the password
            if( m_StrPassword.GetLength() == 0 )
            {
                    MessageBox("Blank passwords are not allowed");
                    m_Password.SetFocus();
            }
            else if( m_StrPassword == m_StrConfPassword )
                    MessageBox("The account is ready\n"
                             "To create the account, click OK.\n"
                           "To stop the account processing, click Cancel.");
            else // if( m_StrPassword != m_StrConfPassword )
            {
                    MessageBox("The passwords do not match");
                    m_Password.SetWindowText("");
                    m_Password.SetFocus();
            }
    }
    // Since the username specified less than 3 characters, display a message
accordingly
    // Set the focus to the Username edit box
    else
    {
            MessageBox("A username must have 5 characters");
            m_Username.SetFocus();
    }
    UpdateData(FALSE);
}
```

6. Display the main form

7. Set the Disabled property of the Account Setup button to True

8. As done for the **EN_MAXTEXT**, add an EN_UPDATE event handler to the First Name edit box

9. Also add an **EN_UPDATE** event handler to the Last Name edit box

10. Implement both events as follows:

```
void CExerciseView::OnUpdateFirstName()
{
    // TODO:  Add your control notification handler code here
    // Let the edit boxes take over
    UpdateData();

    // Make sure both the First Name and the Last Name edit boxes are not empty
    // If one of them is, the account cannot be created.
    // In which case, disable the the Account Setup button
    if( (m_FirstName.GetLength() == 0) || (m_LastName.GetLength() == 0) )
            m_BtnAccount.EnableWindow(FALSE);
```

```
	else
		m_BtnAccount.EnableWindow(TRUE);

	// The dialog box can take over now
	UpdateData(FALSE);
}

void CExerciseView::OnEnUpdateLastName()
{
	// TODO:  Add your control notification handler code here
	UpdateData();

	if( (m_FirstName.GetLength() == 0) || (m_LastName.GetLength() == 0) )
		m_BtnAccount.EnableWindow(FALSE);
	else
		m_BtnAccount.EnableWindow(TRUE);

	UpdateData(FALSE);
}
```

11. Test the application
12. Return to MSVC

16.3 The Rich Edit Control

16.3.1 Overview

A rich edit control is a Windows object that resembles an edit box but can handle text that is formatted. This mean that it can display text with various characters formats and can show paragraphs with different alignments. A rich edit control can also allow a user to change the formatting on characters and control the alignment of paragraphs.

16.3.2 A Rich Edit Control

To create a rich edit control, you ca use the Rich Edit button from the Controls toolbox. You can also programmatically create this control using the CRichEditCtrl class. To do this, use its (default) constructor:

```
CRichEditCtrl RichEditor;
```

After declaring this variable, you can define the characteristics of the control using the CRichEditCtrl::Create() method. Its syntax is:

```
BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID);
```

The dwStyle argument specifies the window style to apply to the control. Since the rich edit control is not a container, it is usually positioned on another control such as a form or dialog box. Therefore, the primary style you must apply is WS_CHILD. To display the control to the user, also add the WS_VISIBLE style. This primary style can be defined as follows:

```
DWORD RichStyle = WS_CHILD | WS_VISIBLE;
```

Although a rich edit control can be used as a single-line text object, to make it more efficient, you should make it use various lines. This can be taken care of by setting the Multiline property to True or adding the ES_MULTILINE style. An exa mple would be:

```
DWORD RichStyle = WS_CHILD | WS_VISIBLE | ES_MULTILNE;
```

Once a rich edit can display multiple lines of text, if the text is longer than the control can display, you should equip it with scroll bars. The vertical scroll bar is made available by adding checking the Vertical Scroll check box or setting it to True. This can be done programmatically by adding the WS_VSCROLL window style. Here is an example:

```
DWORD RichStyle = WS_CHILD | WS_VISIBLE | WS_VSCROLL | ES_MULTILNE;
```

The horizontal scroll bar is made possible by setting the Horizontal Scroll property to True. To do this programmatically, add the WS_HSCROLL window style.

If you do not want the user to change the text in the rich edit control, you can set the Read-Only property to True. This can also be done by adding the ES_READONLY style.

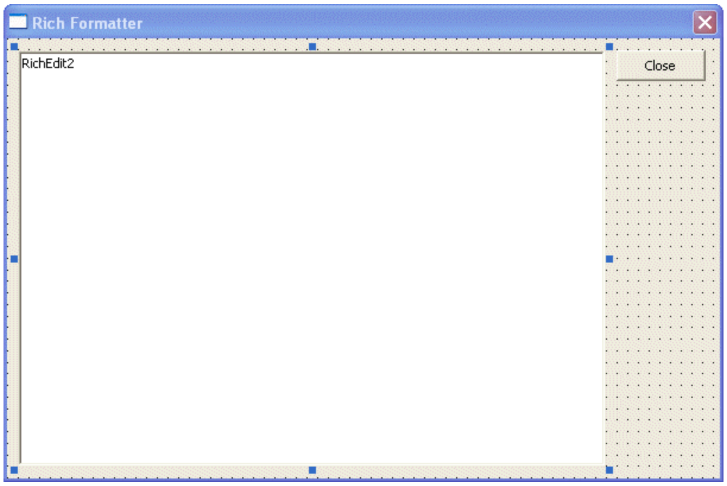
The rect argument specifies the location and dimensions of the control.

The pParentWnd argument is the control that is hosting the rich edit control. It is usually a form or a dialog box.

The nID is an identifier for the rich edit control.

Practical Learning: Creating a Rich Edit Application

1. Create a Dialog Based Application Type project named Richer and based on CformView
2. Delete the TODO line and the OK button
3. Change the caption of the Cancel button to Close
4. On the Control toolbox, click the Rich Edit button and draw a rectangle from the left border to the left of the Cancel



5. On the Properties window, change its ID to IDC_RICHER
6. Set the following properties to True: Multiline, Want Return, Vertical Scroll

7. Add a Control variable (of type CRichEditCtrl) to the rich edit control and name it m_Richer

16.3.3 Rich Edit Properties

At first glance, a rich edit appears like a regular edit control. Its ability to format text and paragraph sets them apart. To change the appearance of a letter, a word or a paragraph, you can change its size, height, or weight. This can be done by calling the CRichEditCtrl::SetSelectionCharFormat() method. Its syntax is:

```
BOOL SetSelectionCharFormat(CHARFORMAT& cf);
```

This simply means that the rich edit control relies on the Win32 API's CHARFORMAT structure to format text. This structure is defined as follows:

```
typedef struct _charformat {
        UINT      cbSize;
        DWORD     dwMask;
        DWORD     dwEffects;
        LONG      yHeight;
        LONG      yOffset;
        COLORREF crTextColor;
        BYTE      bCharSet;
        BYTE      bPitchAndFamily;
        TCHAR     szFaceName[LF_FACESIZE];
} CHARFORMAT;
```

To format the characters, declare a variable of this structure and take its size. Then initialize the necessary me mber variables, ignoring those you do not need. To start, initialize dwMask with the type of formatting you want to apply or the type of operation you want to perform. The possible values are:

Value	Used to
CFM_BOLD	Make the character(s) bold
CFM_ITA LIC	Italicize the character(s)
CFM_UNDERLINE	Underline the character(s)
CFM_STRIKEOUT	Strike out the character(s)
CFM_SIZE	Change the size the character(s)
CFM_CHARSET	Access character set
CFM_COLOR	Change the color of the text
CFM_FACE	Set the font name
CFM_OFFSET	Offset the character(s)
CFM_PROTECTED	Protect the character(s)

You can apply the actual text formatting using the dwEffects member variable. Its possible values are: **CFE_AUTOCOLOR**, **CFE_BOLD**, **CFE_ITALIC**, **CFE_STRIKEOUT**, **CFE_UNDERLINE**, and **CFE_PROTECTED**. These effects can be combined as needed using the bitwise OR operator. For example, you can combine **CFE_BOLD** and **CFE_ITALIC** as **CFE_BOLD | CFE_ITALIC** to have text that is both in bold and italic.

The *yHeight* variable is used to set the new height of the text.

The *yOffset* variable is used to create a superscript or subscript effect.

The *crTextColor* variable is used to set the color for the text.

The *bCharSet* variable is used to the character set value as defined for the Win32's LOGFONT structure.

The *bPitchAndFamily* member variable is the same as set for the LOGFONT structure.

The *szFaceName* variable is used to specify the name of the font to apply to the text.

If you want to find out what formatting is applied on a character or text, call the **CRichEditCtrl::GetSelectionCharFormat()** method. Its syntax is:

```
DWORD GetSelectionCharFormat(CHARFORMAT& cf) const;
```

This method returns the type of dwMask of the **CHARFORMAT** structure applied on the character or text.

To control the alignment of a paragraph on a right edit control, you can call the **CRichEditCtrl::SetParaFormat()** method. Its syntax is:

```
BOOL SetParaFormat(PARAFORMAT& pf);
```

The actual paragraph formatting is set using the Win32 API's **PARAFORMAT** structure. It is created as follows:

```
typedef struct _paraformat {
	UINT cbSize;
	DWORD dwMask;
	WORD wNumbering;
	WORD wReserved;
	LONG dxStartIndent;
	LONG dxRightIndent;
	LONG dxOffset;
	WORD wAlignment;
	SHORT cTabCount;
	LONG rgxTabs[MAX_TAB_STOPS];
} PARAFORMAT;
```

To define the necessary values for this structure, first declare a variable from it and retrieve its size. This is done with the *cbSize* member variable. Secondly, use the *dwMask* member variable to specify what formatting you want to perform. For example, to control paragraph alignment, initialize *dwMask* with **PFM_ALIGNMENT**. On the other hand, if you want to set or remove a bullet on a paragraph, initialize the *dwMask* variable with **PFM_NUMBERING**.

The *wNumbering* member variable is used to apply or remove the bullet from a paragraph.

You will need to use *wReserved*. Therefore, you can either ignore it or set its value to 0.

The *dxStartIndent*, the *dxRightIndent*, and the *dxOffset* member variables are used to indent text.

If you had initialized *dwMask* with **PFM_ALIGNMENT**, you can use *wAlignment* to specify the alignment of the paragraph. The possible values are:

Value	Description
PFA_LEFT	The paragraph will be aligned to the left
PFA_CENTER	The paragraph will be aligned to the center
PFA_RIGHT	The paragraph will be aligned to the right

The cTabCount and the rgxTabs member variables are used to control tab separation.

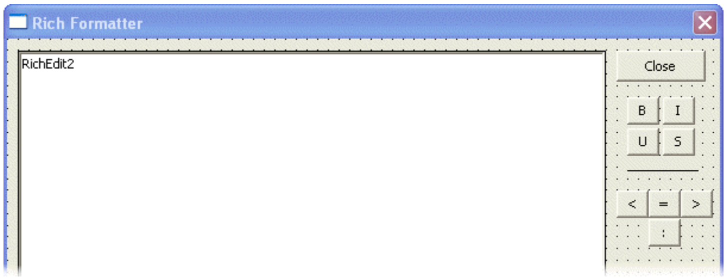
To retrieve the paragraph formatting applied on a paragraph, you can call the **CRichEditCtrl::GetParaFormat()** method. Its syntax is:

```
DWORD GetParaFormat(PARAFORMAT& pf) const;
```

This method returns formatting applied on the selected paragraph.

Practical Learning: Using Rich Edit Properties

1. Add the following 8 buttons:



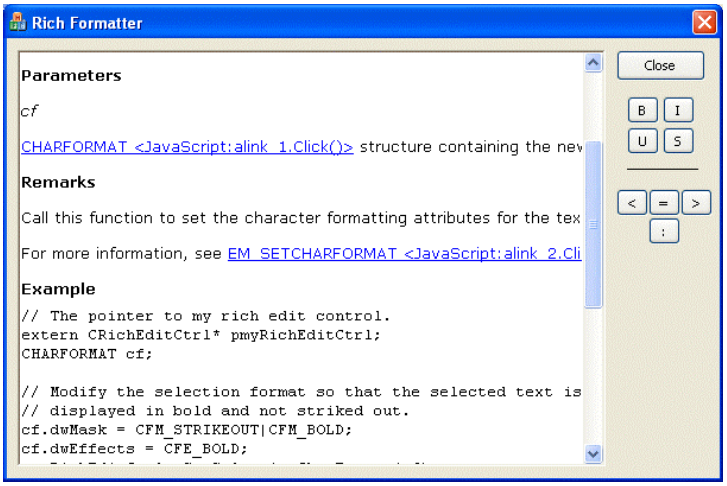
ID	Caption
IDC_BTN_BOLD	B
IDC_BTN_ITALIC	I
IDC_BTN_UNDERLINE	U
IDC_BTN_STRIKEOUT	S
IDC_BTN_LEFT	<
IDC_BTN_CENTER	=
IDC_BTN_RIGHT	>
IDC_BTN_BULLET	:

2. Add a BN_CLICKED event handler for each button and implement the events as follows:

```
void CRicher1Dlg::OnBnClickedBtnBold()
{
	// TODO: Add your control notification handler code here
	CHARFORMAT Cfm;

	m_Richer.GetSelectionCharFormat(Cfm);

	Cfm.cbSize = sizeof(CHARFORMAT);
	Cfm.dwMask = CFM_BOLD;
	Cfm.dwEffects ^= CFE_BOLD;

	m_Richer.SetSelectionCharFormat(Cfm);
	m_Richer.SetFocus();
}

void CRicher1Dlg::OnBnClickedBtnItalic()
{
    // TODO: Add your control notification handler code here
    CHARFORMAT Cfm;

    m_Richer.GetSelectionCharFormat(Cfm);

    Cfm.cbSize = sizeof(CHARFORMAT);
    Cfm.dwMask = CFM_ITALIC;
    Cfm.dwEffects ^= CFE_ITALIC;

    m_Richer.SetSelectionCharFormat(Cfm);
    m_Richer.SetFocus();
}

void CRicher1Dlg::OnBnClickedBtnUnderline()
{
    // TODO: Add your control notification handler code here
    CHARFORMAT Cfm;

    m_Richer.GetSelectionCharFormat(Cfm);

    Cfm.cbSize = sizeof(CHARFORMAT);
    Cfm.dwMask = CFM_UNDERLINE;
    Cfm.dwEffects ^= CFE_UNDERLINE;

    m_Richer.SetSelectionCharFormat(Cfm);
    m_Richer.SetFocus();
}

void CRicher1Dlg::OnBnClickedBtnStrikeout()
{
    // TODO: Add your control notification handler code here
    CHARFORMAT Cfm;

    m_Richer.GetSelectionCharFormat(Cfm);

    Cfm.cbSize = sizeof(CHARFORMAT);
    Cfm.dwMask = CFM_STRIKEOUT;
    Cfm.dwEffects ^= CFE_STRIKEOUT;

    m_Richer.SetSelectionCharFormat(Cfm);
    m_Richer.SetFocus();
}

void CRicher1Dlg::OnBnClickedBtnLeft()
{
    // TODO: Add your control notification handler code here
    PARAFORMAT Pfm;

    Pfm.cbSize = sizeof(PARAFORMAT);
    Pfm.dwMask = PFM_ALIGNMENT;
    Pfm.wAlignment = PFA_LEFT;

    m_Richer.SetParaFormat(Pfm);
    m_Richer.SetFocus();
}
```

```
void CRicher1Dlg::OnBnClickedBtnCenter()
{
        // TODO: Add your control notification handler code here
        PARAFORMAT Pfm;

        Pfm.cbSize = sizeof(PARAFORMAT);
        Pfm.dwMask = PFM_ALIGNMENT;
        Pfm.wAlignment = PFA_CENTER;

        m_Richer.SetParaFormat(Pfm);
        m_Richer.SetFocus();
}

void CRicher1Dlg::OnBnClickedBtnRight()
{
        // TODO: Add your control notification handler code here
        PARAFORMAT Pfm;

        Pfm.cbSize = sizeof(PARAFORMAT);
        Pfm.dwMask = PFM_ALIGNMENT;
        Pfm.wAlignment = PFA_RIGHT;

        m_Richer.SetParaFormat(Pfm);
        m_Richer.SetFocus();
}

void CRicher1Dlg::OnBnClickedBtnBullet()
{
        // TODO: Add your control notification handler code here
        PARAFORMAT Pfm;

        m_Richer.GetParaFormat(Pfm);
        Pfm.cbSize = sizeof(PARAFORMAT);
        Pfm.dwMask = PFM_NUMBERING;

        Pfm.wNumbering ^= PFN_BULLET;

        m_Richer.SetParaFormat(Pfm);
        m_Richer.SetFocus();
}
```

3. Test the application



Parameters

cf

CHARFORMAT <JavaScript:alink_1.Click()> structure containing the nev

Remarks

Call this function to set the character formatting attributes for the tex

For more information, see EM_SETCHARFORMAT <JavaScript:alink_2.Cli

Example

```
// The pointer to my rich edit control.
extern CRichEditCtrl* pmyRichEditCtrl;
CHARFORMAT cf;

// Modify the selection format so that the selected text is
// displayed in bold and not striked out.
cf.dwMask = CFM_STRIKEOUT|CFM_BOLD;
cf.dwEffects = CFE_BOLD;
```

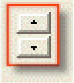
4. Return to MSVC

Chapter 17: Track-Based Controls

- ✍ Spin Buttons
- ✍ Updown Controls
- ✍ Slider Controls

17.1 Spin Button

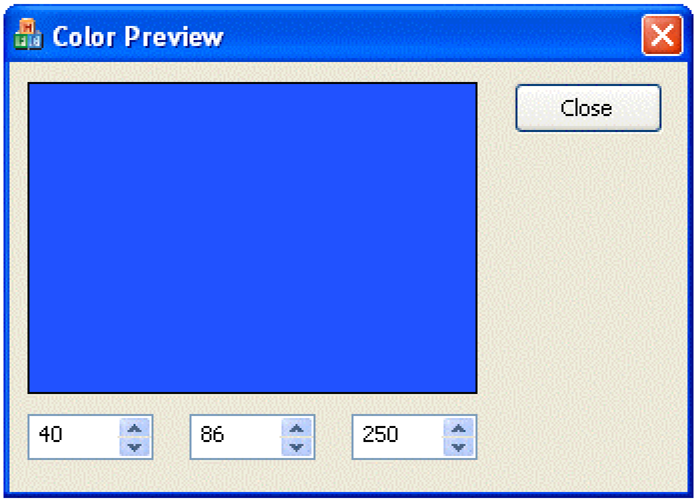
17.1.1 Overview

A spin button is a Windows control equipped with two opposite arrows . The user clicks one of the arrows at one time to increase or decrease the current values of the control. The value held by the control is also called its position. The values of a spin button range from a minimum to a maximum. When the up arrow is clicked the value of the control increases. If the user clicks and holds the mouse on the up pointing arrow. The value of the control keeps increasing until it reaches its maximum and stops. The opposite behavior applies when the user clicks or holds the mouse on the down-pointing arrow.

Because a spin button is only equipped with arrows, it does not inherently show its current value. Therefore, this control is usually accompanied by another, text-based, control, usually an edit box, that indicates its position

.

We are going the create a dialog box equipped with a large static control and three spin buttons. The background color will change when the spin buttons change values:

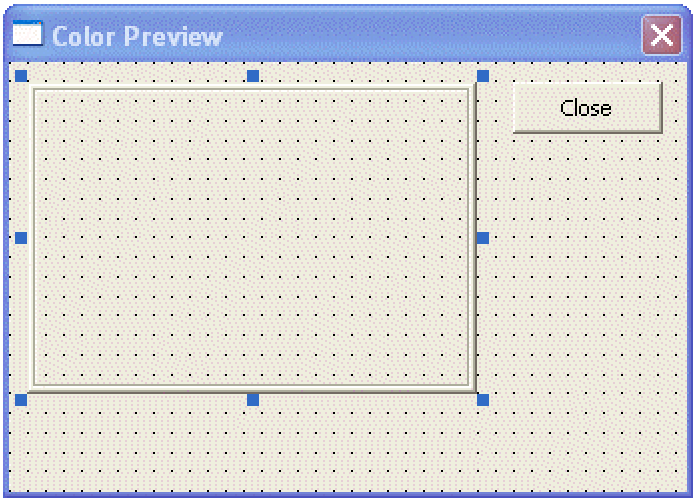


Practical Learning: Introducing a Spin Button

1. Create a new Dialog-Based application with no About Box and name it ColorPreview
2. Set the Dialog Title to Color Preview and click Finish
3. Delete the TODO line and the OK button
4. Change the Caption of the Cancel button to Close

5. On the Controls toolbox, click the Picture button and draw a rectangular object on the dialog box

6. Change its ID to **IDC_PREVIEW**



7. Add a CStatic Control variable for the picture object and name it **m_Preview**

8. In the header file of the CColorPreviewDlg class, declare a private **COLORREF** variable named **PreviewColor** and a member function named **UpdatePreview** of type **void**

```
private:
	COLORREF PreviewColor;
	void UpdatePreview();
};
```

9. In the OnInitDialog event of the dialog, initialize the PreviewColor variable with a gray color as follows:

```
	PreviewColor = RGB(192, 192, 192);
	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

10. In the source file of the dialog box, implement the UpdatePreview() method as follows:

```
void CColorPreviewDlg::UpdatePreview()
{
	CClientDC dc(this); // device context for painting

	// TODO: Add your message handler code here
	CBrush BrushBG(PreviewColor);
	CRect RectPreview;

	m_Preview.GetWindowRect(&RectPreview);
	CBrush *pOldBrush = dc.SelectObject(&BrushBG);

	ScreenToClient(&RectPreview);
	dc.Rectangle(RectPreview);

	dc.SelectObject(pOldBrush);
}
```

11. Save All

12. To set the starting color of the preview picture, access the OnPain() event of the CcolorPreviewDlg class and change it as follows:

```
void CColorPreviewDlg::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    CRect RectPreview;
    CBrush BrushGray(PreviewColor);

    m_Preview.GetWindowRect(&RectPreview);
    CBrush *pOldBrush = dc.SelectObject(&BrushGray);

    ScreenToClient(&RectPreview);
    dc.Rectangle(RectPreview);

    dc.SelectObject(pOldBrush);

    if (IsIconic())
    {
            SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

            // Center icon in client rectangle
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;

            // Draw the icon
            dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
            CDialog::OnPaint();
    }
}
```

13. Test the application and return to MSVC

17.1.2 Creating a Spin Button

To create a spin button, at design time, you can click the Spin button ⬍ on the Controls toolbox and click the desired area on the intended host.

A spin button control is based on the **CSpinButtonCtrl** class. Therefore, to programmatically create a spin button, declare a pointer to **CSpinButtonCtrl** then call its **Create()** method to initialize it. The syntax of the **CSpinButtonCtrl::Create()** method is:

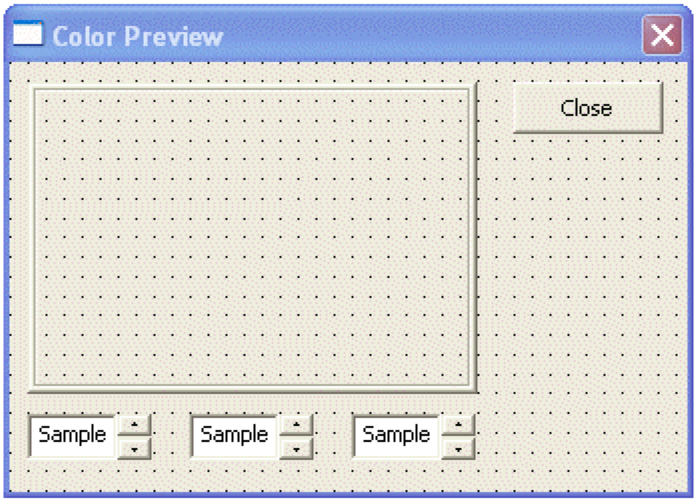
```
BOOL Create( DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
```

Because a spin button cannot be a parent, the minimum style you must apply is **WS_CHILD**. If you want the control to be visible upon creating, add the **WS_VISIBLE** style. Here is an example:

```
BOOL CSpinDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSpinButtonCtrl *SpinCtrl = new CSpinButtonCtrl;

	SpinCtrl->Create(WS_CHILD | WS_VISIBLE, CRect(60, 10, 80, 35), this, 0x128);

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

Practical Learning: Creating Spin Buttons

Here is the dialog box as we will design it:



1. On the Controls toolbox, click the Edit Box button ab| and click under the picture control and on the left side of the dialog box
2. On the Control toolbox, click the Spin button ⬍ and click to the right side of the IDC_EDIT1 control on the dialog box
3. While the spin button is still selected, press and hold Ctrl. Click the previously added edit box and release Ctrl
4. Press Ctrl + C to copy and press Ctrl + V to paste
5. Move the newly two pasted controls to the right
6. Press Ctrl + V to paste again and move the new controls to the right side of the others
7. Using the properties window, change the identifiers of the controls, from left to right, to **IDC_EDIT_RED**, **IDC_SPIN_RED**, **IDC_EDIT_GREEN**, **IDC_SPIN_GREEN**, **IDC_EDIT_BLUE**, and **IDC_SPIN_BLUE**

17.1.3 The Spin Button Properties

By default, a spin button appears with an up and a down pointing arrows. If you want the arrows to be horizontally directed, change the value of the Orientation property from Vertical (the default) to Horizontal. If you are programmatically creating the control, add the UDS_HORZ style to it. Here is an example:

```
SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_HORZ,
                 CRect(60, 10, 80, 35), this, 0x128);
```

We mentioned already that the value of the spin changes as the user clicks or holds the mouse on one arrow of the control. In the same way, if you want the user to be able to increment or decrement the value of the control using the up and down arrow keys of the keyboard, set the Arrow Keys property to True at design time or add the **UDS_ARROWKEYS** style.

As stated already, a spin button cannot display its value to the user. If you want to inform the user about the value of the control as it is being incremented or decremented, you can add another, usually text-based, control, such as an edit box. This control is called the buddy window of the spin button. The control should be positioned on one horizontal side of the spin button. After adding that new control, you should let the spin button know on what side the accompanying control is positioned, left or right. To do this, select the appropriate value in the **Alignment** property.

If you are programmatically creating the control, to position the spin button to the left edge of the buddy window, add the **UDS_ALIGNLEFT** style. On the other hand, if you want the spin button on the right side of the buddy window, apply the **UDS_ALIGNRIGHT** style instead. Just like at design time you cannot apply both styles, at run time, do not add both the **UDS_ALIGNLEFT** and the **UDS_ALIGNRIGHT** styles.

If you want the spin button to use an already existing and previously added control as its buddy window, set the Auto Buddy property to True or apply the **UDS_AUTOBUDDY** style. In this case, the control that was just previously added to the host, before adding the spin button, would be used as its buddy.

After specifying what control would be the buddy window of the spin button, when the user clicks the arrows of the button, the buddy would display its current value. If you want to make sure that the buddy window display only integral values, whether decimal or hexadecimal, change the Set Buddy Integer property to True. If you are programmatically creating the control, you can add the **UDS_SETBUDDYINT** style:

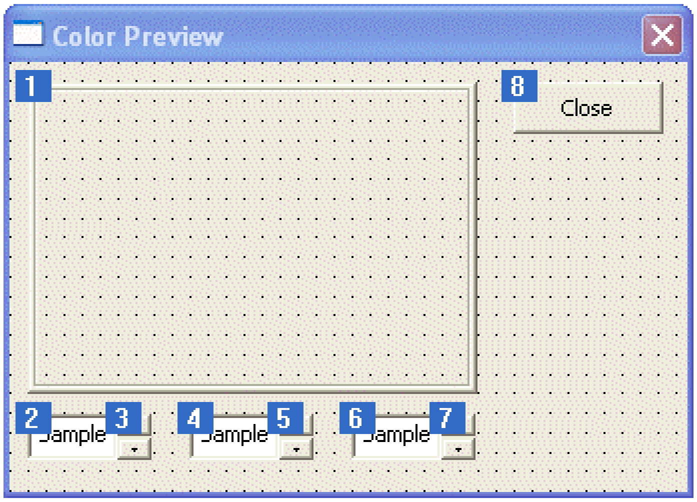
```
SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_SETBUDDYNT,
                 CRect(60, 10, 80, 35), this, 0x128);
```

When the buddy window displays the values of the spin button and when the value gets over 999, if you want the number to be separated by a comma, set the **No Thousands** property to True. To apply this property with code, add the **UDS_NOTHOUSANDS** style. If you do not want the thousand sections to be separated by comma, set the **No Thousands** property to False or omit the **UDS_NOTHOUSANDS** style.

Imagine you create a spin button and specify its range of values from 3 to 22 with an incremental of 5. When the control is incremented to 20 and the user clicks the up arrow button or presses the up arrow key, you can decide that the incrementing should stop at 20 or wrap to 22, although 22 is not a valid incremental value for this scenario. The ability to control this behavior is set by changing the **Wrap** property to True or by adding the **UDS_WRAP** style. This behavior applies also if the value has been fully decremented.

Practical Learning: Designing an UpDown control

1. Arrange the tab sequence as follows:



2. Click one of the spin buttons to select it. Then press and hold Ctrl. Click the other two spin buttons to select them, and release Ctrl
3. On the Properties window, click the arrow of the **Alignment** combo box and select **Right**
4. Check the **Auto Buddy** check box or set its value to True
5. Check the **Set Buddy Integer** check box or set its value to True
6. Test the application
7. Close the dialog box and return to MSVC

17.1.4 Methods of Managing an UpDown Control

We have already mentioned that a spin button is based on the **CSpinButtonCtrl** class. This class is equipped with a default constructor used to declare a **CSpinButtonCtrl** variable or pointer and the **Create()** method used to initialize the control. Here is an example:

```
BOOL CSpinDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSpinButtonCtrl *SpinCtrl = new CSpinButtonCtrl;

	SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_SETBUDDYINT,
			                 CRect(60, 10, 80, 35), this, 0x128);

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

After creating the control, to indicate what control would display the value of the spin button, we saw that you can use the **Alignment** or the **Auto Buddy** properties. If you did not do this at design time, and if you want to explicitly specify the name of the control that would act as the buddy window, you can call the **CSpinButtonCtrl::SetBuddy()** method. Its syntax is:

```
CWnd* SetBuddy(CWnd* pWndBuddy);
```

The *pWndBuddy* argument is the new control that would serve as buddy. Here is an example that sets an existing label on the dialog box (the label was created as a Static Text control and identified as IDC_SPIN_BUDDY) as the spin button's buddy window:

```
BOOL CSpinDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSpinButtonCtrl *SpinCtrl = new CSpinButtonCtrl;
	CStatic *SpinBuddy;

	SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_SETBUDDYINT,
	                                        CRect(60, 10, 80, 35), this, 0x128);

	SpinBuddy = reinterpret_cast<CStatic *>(GetDlgItem(IDC_SPIN_BUDDY));
	SpinCtrl->SetBuddy(SpinBuddy);

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

If the buddy window has already been set and you want to find what control performs that role, you can call the **CSpinButtonCtrl::GetBuddy()** method. Its syntax is:

```
CWnd* GetBuddy( ) const;
```

This method returns a handle to the control that acts as the buddy window of the spin button that called it.

One of the most important actions you should perform after creating a spin button is to specify its lowest and its highest values. The default range is 100 (lowest) to 0 (highest). This causes the spin button to count in decrement. If you do not want this (bizarre) behavior, you must explicitly set the lower and higher values.

To set the minimum and maximum values of a spin button, call either the **CSpinButtonCtrl::SetRange()** or the **CSpinButtonCtrl::SetRange32()** methods. Their syntaxes are:

```
void SetRange( int nLower, int nUpper );
void SetRange32( int nLower, int nUpper );
```

In both cases the *nLower* argument holds the minimum value and the *nUpper* argument specifies the maximum value. Here is an example:

```
BOOL CSpinDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSpinButtonCtrl *SpinCtrl = new CSpinButtonCtrl;

	SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_SETBUDDYINT,
			CRect(60, 10, 80, 35), this, 0x128);

	SpinCtrl->SetRange(-12, 1244);

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

If the control exists already and you want to get its minimum and maximum values, call either the **CSpinButtonCtrl::GetRange()** or the **CSpinButtonCtrl::GetRange32()** methods. The possible syntaxes used are:

```
DWORD GetRange() const;
void GetRange(int &lower, int& upper) const;
void GetRange32(int &lower, int &upper) const;
```

Based on the range of values that a spin button can handle, the user can increment and decrement the control's value. By default, the values are are incremented by adding 1 and decrement by adding –1 to the current value. If you want to increment and decrement by a different value, you have two main options. You can write a routine to take care of this, or call the **CSpinButtonCtrl::SetAccel()** method. Its syntax is:

```
BOOL SetAccel(int nAccel, UDACCEL* pAccel );
```

The **SetAccel()** method takes a UDACCEL value and its size as arguments. The UDACCEL class is defined as follows:

```
typedef struct {
   UINT nSec;
   UINT nInc;
}UDACCEL, FAR *LPUDACCEL;
```

The *nSec* member variable is a semi-timer that ticks at a specified rate of seconds.
The nInc member variable of this structure defines the incremental value to apply when the *nSec* value has passed.

The *nAccel* argument of the **SetAccel()** method is the size of the **UDACCEL** class.

In the following example, a spin button was added to a dialog box and a control variable named m_Spin was added for it. When the user clicks the arrow buttons or presses the arrow keys, the value of the spin button is incremented by 5:

```
BOOL CDlgSpin::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	UDACCEL udAccel;

	int SizeOfAccel = sizeof(UDACCEL);
	udAccel.nSec = 100;
```

```
        udAccel.nInc = 5;

        m_Spin.SetAccel(SizeOfAccel, &udAccel);
        m_Spin.SetRange(12, 168);

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

If a spin button has been created already, to find its incremental value, you can call the **CSpinButtonCtrl::GetAccel()** method. Its syntax is:

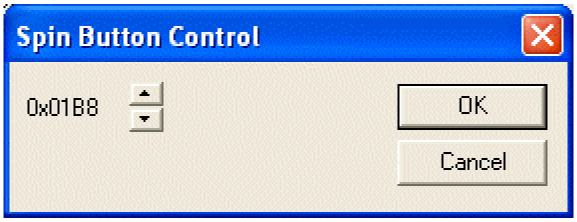
UINT GetAccel(int *nAccel*, UDACCEL* *pAccel*) const;

Normally, the values of a spin button are decimal integers. Alternatively, if you prefer the values to be given in hexadecimal format, set the range accordingly and call the **CSpinButtonCtrl::SetBase()** method. Its syntax is:

int SetBase (int *nBase*);

Using this method, if you want the value to be decimal, pass the nBase argument as 10. If you want hexadecimal values, pass the argument as 16. Here is an example:

```
BOOL CSpinDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        CSpinButtonCtrl *SpinCtrl = new CSpinButtonCtrl;
        CStatic *SpinBuddy;

        SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_SETBUDDYINT,
                                          CRect(60, 10, 80, 35), this, 0x128);
        SpinCtrl->SetRange(0x04, 0xFF06);
        SpinCtrl->SetBase(16);

        SpinBuddy = reinterpret_cast<CStatic *>(GetDlgItem(IDC_SPIN_BUDDY));
        SpinCtrl->SetBuddy(SpinBuddy);

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



To find out the current base, decimal or hexadecimal, that a spin button is using for its values, call the **CSpinButtonCtrl::GetBase()** method. Its syntax:

UINT GetBase() const;

Once a spin button is ready to hold values, its default value is the lowest specified with the **SetRange()** or the **SetRange32()** method. For example, if you create a spin button that can hold values from 15 to 62, when the control displays at startup, it would assume a value of 15. The value held by a spin button is referred to as its position. If you want the control to have a value different than the lowest, that is, to hold a different position, call the **CSpinButtonCtrl::SetPos()** method. Its syntax:

```
int SetPos(int nPos);
```

This method takes as argument the new value of the spin button. The value must be in the range set with the **SetRange()** or the **SetRange32()** methods. Here is an example:

```
BOOL CSpinDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSpinButtonCtrl *SpinCtrl = new CSpinButtonCtrl;
	CStatic *SpinBuddy;

	SpinCtrl->Create(WS_CHILD | WS_VISIBLE | UDS_SETBUDDYINT,
				CRect(60, 10, 80, 35), this, 0x128);

	SpinCtrl->SetRange(4, 64);
	SpinCtrl->SetBase(10);

	SpinCtrl->SetPos(36);

	SpinBuddy = reinterpret_cast<CStatic *>(GetDlgItem(IDC_SPIN_BUDDY));
	SpinCtrl->SetBuddy(SpinBuddy);
	SpinBuddy->SetWindowText("4");

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

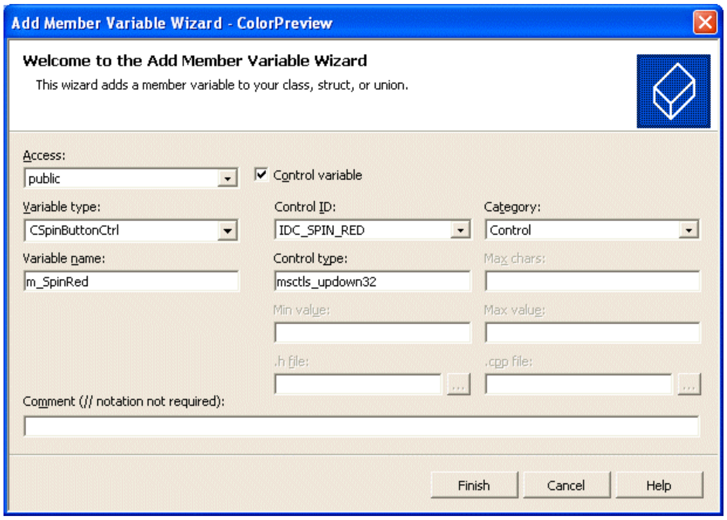
To explore the spin button, the user click one of its arrow buttons or presses the arrow keys to increase or decrease the control’s value. The only actual thing the spin button provides is the value it holds. It is up to you to decide what to do with such a value. This means that, on a regular basis, you will need to retrieve the current value of the control and do whatever you want with it.. To get the position of a spin button, call the **CSpinButtonCtrl::GetPos()** method. Its syntax is:

```
int GetPos() const;
```

This method returns the value of the spin button at the time the method is called.

Practical Learning: Using a Spin Button

1. If you are using MSVC 6, press Ctrl + W to access the Class Wizard. Click the Member Variables property page. Click IDC_SPIN_RED and click Add Variable… If you are using MSVC 7, on the dialog box, right-click the most left spin button and click Add Variable…
2. Set the name of the variable to **m_SpinRed**



Add Member Variable Wizard - ColorPreview

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: public

☑ Control variable

Variable type: CSpinButtonCtrl

Control ID: IDC_SPIN_RED

Category: Control

Variable name: m_SpinRed

Control type: msctls_updown32

Max chars:

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required):

3. Click OK or Finish
4. In the same way, add a Control variable for the IDC_SPIN_GREEN control named **m_SpinGreen** and a Control variable for the IDC_SPIN_BLUE control named **m_SpinBlue**
5. For MSVC 6, click the Message Maps property page. In the Member Functions list box, click OnInitDialog and click Edit Code

   In the OnInitDialog event of the CcolorPreviewDlg class, set the range of the spin buttons to (0, 255) and set its initial value to 192

```
BOOL CColorPreviewDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_SpinRed.SetRange(0, 255);
	m_SpinGreen.SetRange(0, 255);
	m_SpinBlue.SetRange(0, 255);

	m_SpinRed.SetPos(192);
	m_SpinGreen.SetPos(192);
	m_SpinBlue.SetPos(192);

	PreviewColor = RGB(192, 192, 192);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

6. Execute the application. Test the spin buttons and make sure they are working fine. Then close the dialog box and return to MSVC

17.1.5 The Spin Button Events

When the user clicks one of the arrow buttons of a spin button or presses an up or a down arrow key when the control has focus, the operaing sends a **UDN_DELTAPOS** message to the parent window of the spin button, notifying this parent that the position (the value) of the control is about to be changed. The syntax of the event fired by the **UDN_DELTAPOS** message is:

```
OnDeltaPosSpin(NMHDR* pNMHDR, LRESULT* pResult)
```

Because this event is fired before the value of the spin button is changed, you can use it to check, validate, allow or deny the change. The first argument, *pNMHDR*, is an **NMHDR** structure value. The NMHDR structure is sometimes used to carry information about a message. It is defined as follows:

```
typedef struct tagNMHDR {
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

The *hwndFrom* member variable is a handle to the window that is sending the message. The idFrom is the identifier of the control that is sending the message. The code member is the actual notification code.

When implementing the event of the **UDN_DELTAPOS** message, instead of using the value of the NMHDR argument, Visual C++ takes the liberty of casting the *pNMHDR* pointer into a pointer to NM_UPDOWN. Therefore, the event provided to you appears as follows:

```
void CDlgSpin::OnDeltaPosSpinNew(NMHDR* pNMHDR, LRESULT* pResult)
{
	NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
	// TODO: Add your control notification handler code here

	*pResult = 0;
}
```

The **NM_UPDOWN** structure is defined as follows:

```
typedef struct _NM_UPDOWN {
    NMHDR hdr;
    int   iPos;
    int   iDelta;
} NMUPDOWN, FAR *LPNMUPDOWN;
```

This structure was specifically created to carry notification information for a spin button. The first member variable of the **NM_UPDOWN** structure, *hdr*, is an **NMHDR**. The hdr itself carries additional information about the message being sent, as mentioned above. The *iPos* member variable is the value of the current position of the spin button. The *iDelta* member is the intended change that would be performed on the spin button.

Practical Learning: Using the Spin Button Events

1. If you are using MSVC 6, display the ClassWizard and the Message Maps property page. Click IDC_SPIN_RED. In the Messages list, double-click **UDN_DELTAPOS**.

Accept the suggested name of the function and click OK. Then click Edit Code…
If you are using MSVC 7, click the most left spin button on the dialog box and, on the Properties window, click the Controls Events button ⚡. Click the arrow of the UDN_DELTAPOS combo box and select the only item in the list

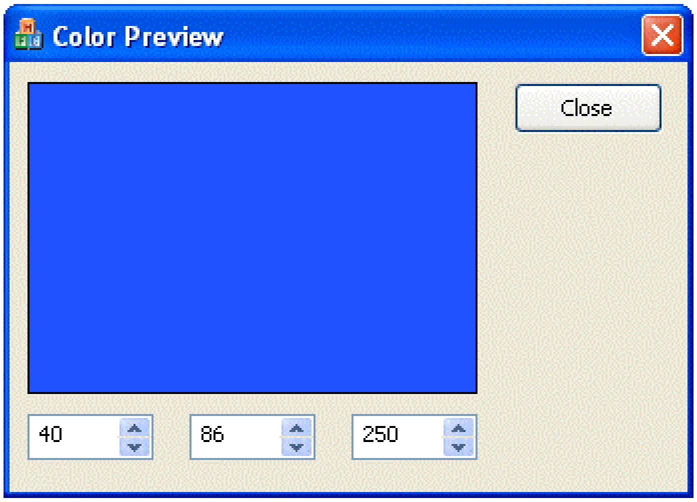
2. Implement the event as follows:

```
void CColorPreviewDlg::OnDeltaposSpinRed(NMHDR *pNMHDR, LRESULT *pResult)
{
	LPNMUPDOWN pNMUpDown = reinterpret_cast<LPNMUPDOWN>(pNMHDR);
	// TODO: Add your control notification handler code here
	int RColor = m_SpinRed.GetPos();
	int GColor = m_SpinGreen.GetPos();
	int BColor = m_SpinBlue.GetPos();

	PreviewColor = RGB(RColor, GColor, BColor);
	UpdatePreview();

	*pResult = 0;
}
```

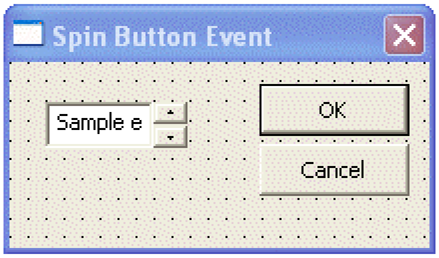
3. In the same way, add the UDN_DELTAPOS event of the IDC_SPIN_GREEN and the IDC_SPIN_BLUE controls and implement them as follows:

```
void CColorPreviewDlg::OnDeltaposSpinGreen(NMHDR *pNMHDR, LRESULT *pResult)
{
	LPNMUPDOWN pNMUpDown = reinterpret_cast<LPNMUPDOWN>(pNMHDR);
	// TODO: Add your control notification handler code here
	int RColor = m_SpinRed.GetPos();
	int GColor = m_SpinGreen.GetPos();
	int BColor = m_SpinBlue.GetPos();

	PreviewColor = RGB(RColor, GColor, BColor);
	UpdatePreview();

	*pResult = 0;
}

void CColorPreviewDlg::OnDeltaposSpinBlue(NMHDR *pNMHDR, LRESULT *pResult)
{
	LPNMUPDOWN pNMUpDown = reinterpret_cast<LPNMUPDOWN>(pNMHDR);
	// TODO: Add your control notification handler code here
	int RColor = m_SpinRed.GetPos();
	int GColor = m_SpinGreen.GetPos();
	int BColor = m_SpinBlue.GetPos();

	PreviewColor = RGB(RColor, GColor, BColor);
	UpdatePreview();

	*pResult = 0;
}
```

4. Test the application:



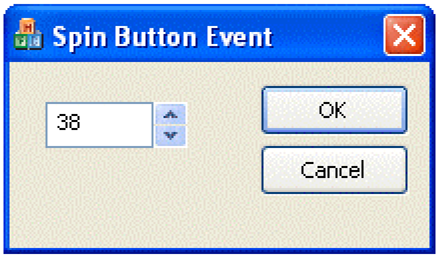
5. After testing the spin buttons, close the dialog box and return to MSVC
6. Now we will implement an exercise that exploits the *pNMHDR* argument of the **UDN_DELTAPOS** message.
   Create a new Dialog-based application named SpinEvent without the About Box
7. Delete the TODO line
8. Change the caption of the Cancel button to Close
9. Add an Edit Box control ab| to the top-left section of the dialog box and change its ID to IDC_EDIT
10. Add a Spin button ⬍ to the right side of the edit box and change only its ID to IDC_SPIN



11. Add a Control variable for the edit box and name it **m_Edit**
12. Add a Control variable for the spin button and name it **m_Spin**
13. As done in the previous exercise, fire the **UDN_DELTAPOS** event for the spin button.
14. Change the OnInitDialog event of the dialog box and implement the event of the **UDN_DELTAPOS** message as follows:

```
BOOL CSpinEventDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_Spin.SetRange(12, 185);
	m_Edit.SetWindowText("26");

	return TRUE;  // return TRUE  unless you set the focus to a control
}

. . .

void CSpinEventDlg::OnDeltaposSpin(NMHDR *pNMHDR, LRESULT *pResult)
{
	LPNMUPDOWN pNMUpDown = reinterpret_cast<LPNMUPDOWN>(pNMHDR);
	// TODO: Add your control notification handler code here

	// Declare a pointer to a CSpinButtonCtrl;
	CSpinButtonCtrl *Spinner;

	// Get a pointer to our spin button
	Spinner = reinterpret_cast<CSpinButtonCtrl *>(GetDlgItem(IDC_SPIN));

	// Found out if it is our spin button that sent the message
	// This conditional statement appears useless but so what?
	if( pNMHDR->hwndFrom == Spinner->m_hWnd )
	{
		// Get the current value of the spin button
		int CurPos = pNMUpDown->iPos;

		// Convert the value to a string
		char StrPos[20];

		sprintf(StrPos, "%d", CurPos);

		// Display the value into the accompanying edit box
		m_Edit.SetWindowText(StrPos);
	}

	*pResult = 0;
}
```

15. Test the application


Spin Button Event

38

16. Close it and return to MSVC

17.2 The UpDown Control

17.2.1 Overview

Besides the spin button, Visual C++ ships with an ActiveX control that provides the same functionality. To use this object, you can add it from the Insert ActiveX Control dialog box and it is named Microsoft UpDown Control. If there are two versions provided, you should select the latest, which usually has a higher version number.

The Microsoft UpDown control is simply a spin button but with true visual accessibility and rapid application development (RAD) concept. Like the spin button, it is equipped with two buttons. Each button has a small picture (a bitmap) that displays an arrow.

The application we are about to develop is for a CD publishing small business. This company manufactures compact discs for self-promoting musicians and small business that want to sell their own CDs. When taking an order of a new CD, the company charges:

?? $20/CD if the customer is ordering less than 20 units

?? $15/CD if the customer is ordering up to 50 units

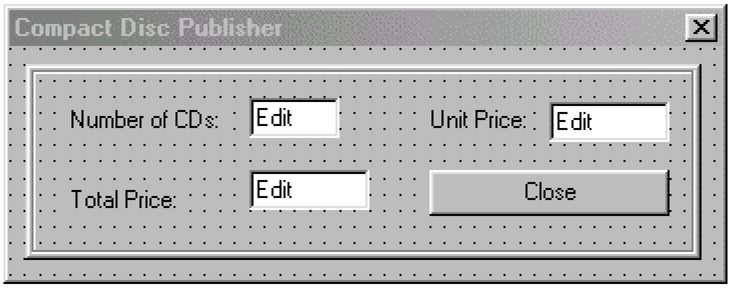
?? $12/CD if the customer is ordering up to 100 units

?? $8/CD if the customer is ordering up to 500 units

?? $5/CD for any order over 500 units

Practical Learning: Using UpDown Controls

1. Create a new Dialog-based application named **CDPublisher** and set its title to **Compact Disc Publisher**
2. Delete the TODO line and the OK button
3. Change the Caption of the Cancel button to **Close**
4. Add a Picture control to the dialog box. Set its Color property to Etched and check its Modal Frame check box
5. Design the dialog box as follows:



The names of the edit boxes from left to right are **IDC_EDIT_QTY**, **IDC_UNITPRICE**, and **IDC_TOTALPRICE**

6. Add a **CString** variable for the IDC_EDIT_QTY control and name it **m_EditQuantity**
7. Add a CString variable for the IDC_ UNITPRICE control and name it **m_UnitPrice**
8. Add a CString variable for the IDC_ TOTALPRICE control and name it **m_TotalPrice**
9. Access the source code of the CCDPublisherDlg class and, in its constructor, initialize the edit boxes to 1 or $20.00

```
CCDPublisherDlg::CCDPublisherDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CCDPublisherDlg::IDD, pParent)
{
        //{{AFX_DATA_INIT(CCDPublisherDlg)
        m_EditQuantity = _T("1");
        m_UnitPrice = _T("$20.00");
        m_TotalPrice = _T("$20.00");
        //}}AFX_DATA_INIT
        // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

10. Save All

17.2.2 Using an UpDown Control

To provide an UpDown control to your application, display the Insert ActiveX Control dialog box, select Microsoft UpDown Control 6.0 and click OK. If you plan to refer to the control in your code, you can add a variable for it. When you do this, Visual C++ would create and even implement its class, giving all of its code (its header and source files). You can then find out that the UpDown control is based on the **CUpDown** class which itself is based on **CWnd**, making it convenient to use **CWnd** properties and methods.

Like the spin button, the UpDown control cannot display its value to the user. If you want an accompanying control to play that role, you should place the new UpDown control next to a text-based control such as an edit box. You have the option of positioning the UpDown control to the left or the right side of its buddy control.

After placing the UpDown control on the parent window, by default, its arrow buttons point up and down. If you want the arrows to point left and right, change the value of the **Orientation** property.

To make the UpDown control easier to configure, you can set its range value visually. The minimum value is set using the **Min** edit box the maximum value is set on the **Max** edit box. To programmatically change the minimum value, call the **SetMin()** method. In the same way, the maximum value can be changed by calling the **SetMax()** method. Here is an example:

```
void CDlgSpin::OnConfigureUpDown()
{
        // TODO: Add your control notification handler code here
        m_UpDown.SetMin(12);
        m_UpDown.SetMax(250);
}
```

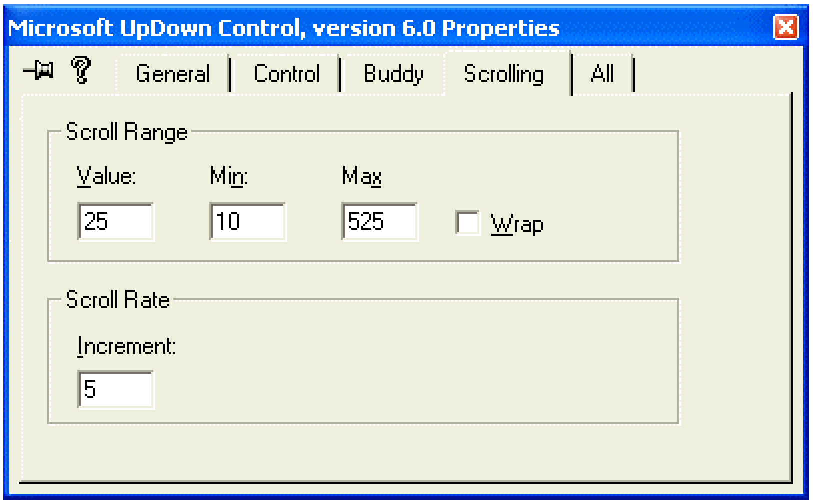
To get the minimum value of an UpDown control, call the **GetMin()** method. To get the maximum value of an UpDown control, call the **GetMax()** method.

After setting the minimum and the maximum values, you can specify the initial value the UpDown control would hold when the application comes up. This value is set using the **Value** edit box and it must be in the range (**Min**, **Max**). To change the value with code, call the **SetValue()** method and pass it the desired but valid value. Here is an example:

```
void CDlgSpin:: OnConfigureUpDown()
{
        // TODO: Add your control notification handler code here
        m_UpDown.SetMin(12);
        m_UpDown.SetMax(250);
        m_UpDown.SetValue(88);
}
```

To get the value of an UpDown control, call its **GetValue()** method.

When using the UpDown control by clicking one of its buttons, its value increases or decreases by one. If you want a different incremental value, specify it using the **Increment** edit box.



Scroll Range

Value: 25

Min: 10

Max: 525

□ Wrap

Scroll Rate

Increment: 5

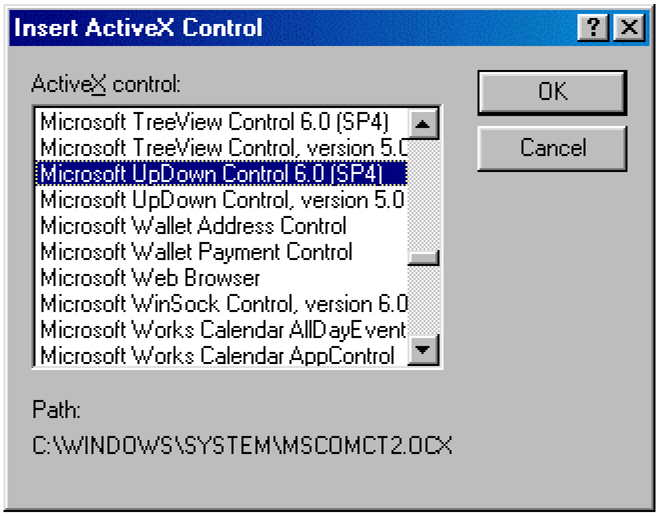
To programmatically set an incremental value, call the **SetIncrement()** method. Here is an example:

```
void CDlgSpin:: OnConfigureUpDown()
{
        // TODO: Add your control notification handler code here
        m_UpDown.SetMin(12);
        m_UpDown.SetMax(250);

        m_UpDown.SetValue(88);

        m_UpDown.SetIncrement(5);
}
```

To know what increment value an UpDown control is using, call its **GetIncrement()** method.

Practical Learning: Using UpDown Controls

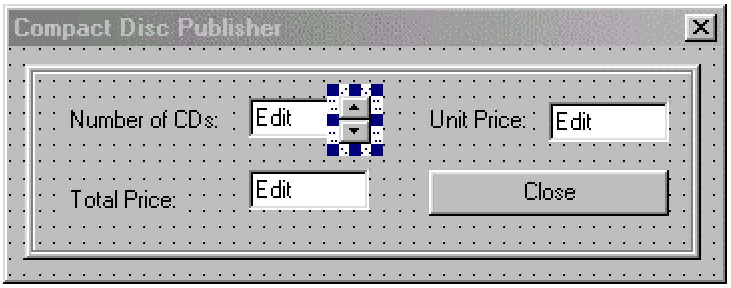
1. Right-click in the middle of the dialog box and click Insert ActiveX Control


Insert ActiveX Control
ActiveX control:
Microsoft TreeView Control 6.0 (SP4)
Microsoft TreeView Control, version 5.0
Microsoft UpDown Control 6.0 (SP4)
Microsoft UpDown Control, version 5.0
Microsoft Wallet Address Control
Microsoft Wallet Payment Control
Microsoft Web Browser
Microsoft WinSock Control, version 6.0
Microsoft Works Calendar AllDayEvent
Microsoft Works Calendar AppControl
OK
Cancel

Path:
C:\WINDOWS\SYSTEM\MSCOMCT2.OCX

2. Click OK

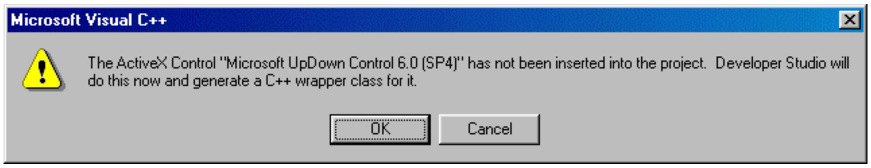
3. Position the new control to the right side of the Number of CDs edit box and resize it:



4. Using the Properties window, change its ID to **IDC_QUANTITY**

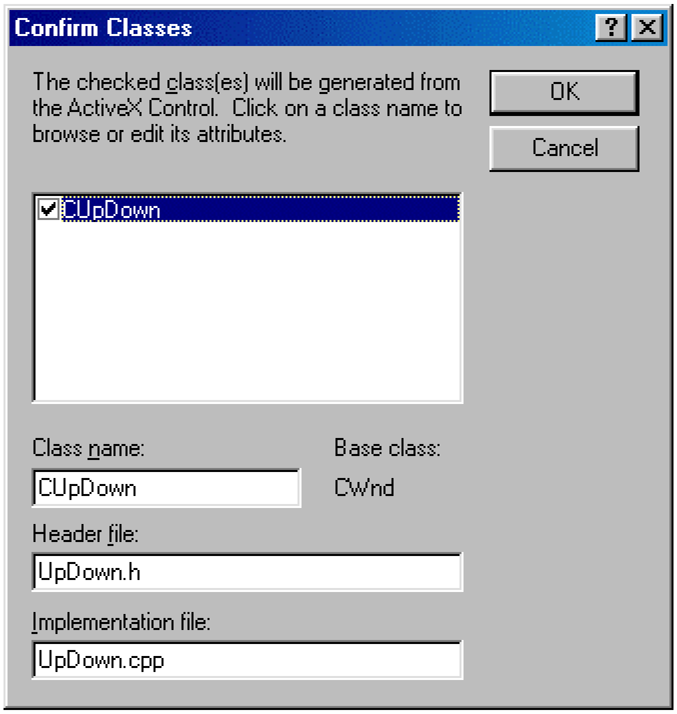
5. Set the values of the Min edit box **1** and the Max edit box to **5000**

6. Add a variable to the UpDown control. You may receive a message box informing you that the control needs to be inserted into your project:



The ActiveX Control "Microsoft UpDown Control 6.0 (SP4)" has not been inserted into the project. Developer Studio will do this now and generate a C++ wrapper class for it.

Click OK.



The checked class(es) will be generated from the ActiveX Control. Click on a class name to browse or edit its attributes.

☑ CUpDown

Class name: CUpDown

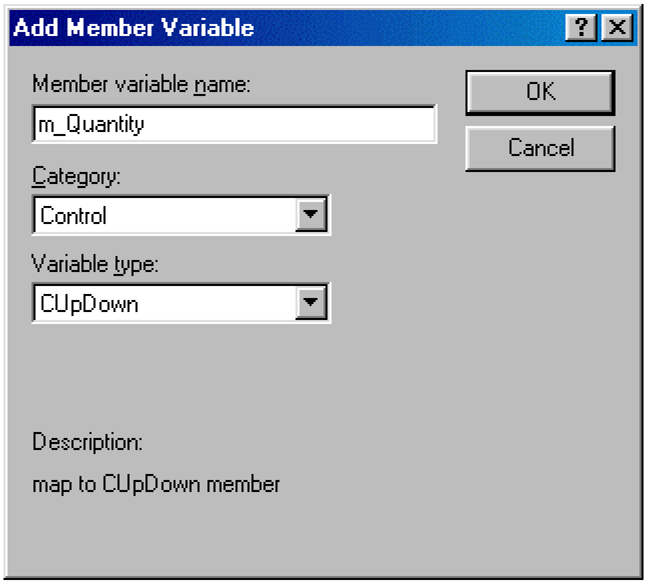
Base class: CWnd

Header file: UpDown.h

Implementation file: UpDown.cpp

Click OK again

Set the name of the variable to **m_Quantity**



Member variable name: m_Quantity

Category: Control

Variable type: CUpDown

Description:

map to CUpDown member

7. Click OK twice
8. In Class View, expand **CUpDown** and double-click it to display its source code

17.2.3 The UpDown Control Events

The UpDown control makes an object distinction between its button components. For example, when the user clicks the up pointing arrow button, the control fires the

UpClick() event. On the other hand, when the user clicks the down pointing arrow button, the control sends a **DownClick()** event. These allow you to treat each event separately if you want.

If you are more interested in the value of the UpDown control, when the user clicks either of its buttons, the value of the control changes. Once the value of the control has been modified, it fires a **Change()** event.

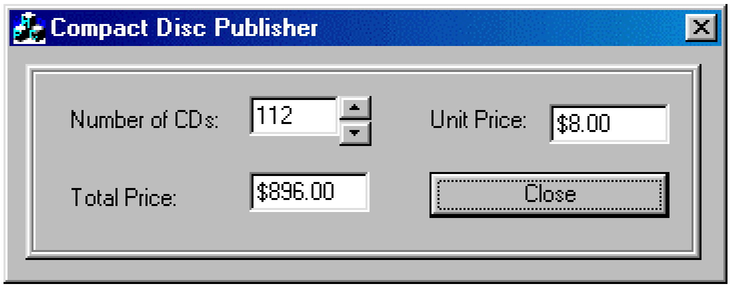
The UpDown control provides bonus events related to the mouse. When the mouse arrives to passes over this control, the **MouseMove()** event is fired. If the user presses a mouse button on the control. It fires the **MouseDown()** event. When the user releases the mouse button, the **MouseUp()** event is fired.

Practical Learning: Configuring an UpDown Control

1. If you are using MSVC 6, display the ClassWizard and the Message Maps property page. Click the IDC_QUANTITY. In the Messages section, double-click Change, click OK and click Edit Code
   If you are using MSVC 7, display the dialog box and click the UpDown control. On the Property window, click the Control Events button ⚡. Click the arrow of the Change combo box and select the only item in the list
2. Implement the event as follows:

```
void CCDPublisherDlg::OnChangeQuantity()
{
	// TODO: Add your control notification handler code here
	int Quantity;
	double UnitPrice, TotalPrice;

	Quantity = m_Quantity.GetValue();

	if( Quantity < 20 )
		UnitPrice = 20;
	else if( Quantity < 50 )
		UnitPrice = 15;
	else if( Quantity < 100 )
		UnitPrice = 12;
	else if( Quantity < 500 )
		UnitPrice = 8;
	else
		UnitPrice = 5;

	TotalPrice = Quantity * UnitPrice;

	m_EditQuantity.Format("%d", Quantity);
	m_UnitPrice.Format("$%.2f", UnitPrice);
	m_TotalPrice.Format("$%.2f", TotalPrice);

	UpdateData(FALSE);
}
```

3. Test the application



Number of CDs: 112

Unit Price: $8.00

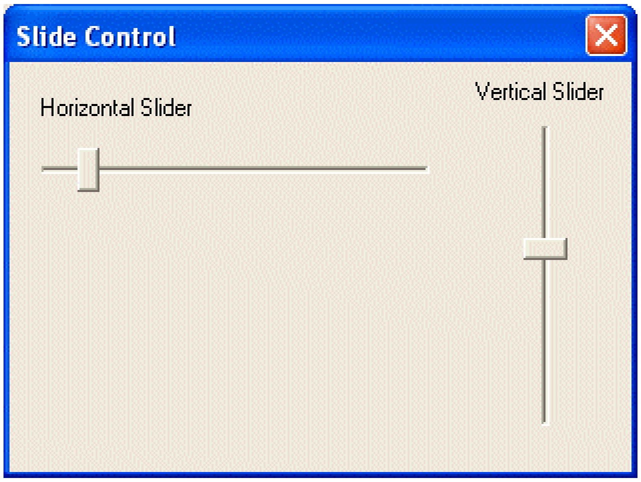
Total Price: $896.00

4. Close it and return to MSVC

17.3 Slider Controls

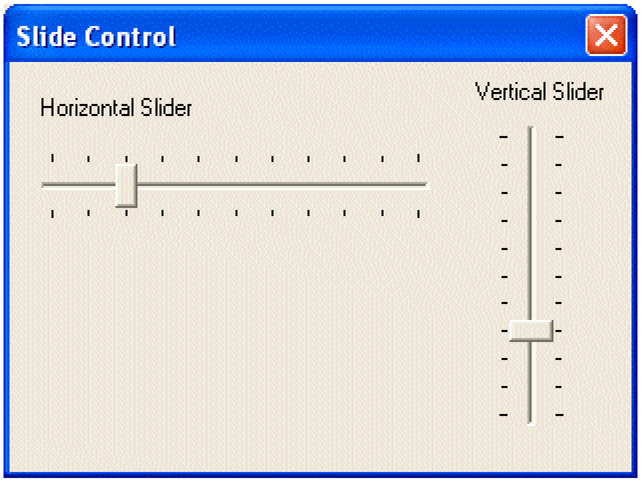
17.3.1 Overview

A slider is a Windows control equipped with a small bar, also called a thumb, which slides along a visible line. There are two types of sliders: horizontal and vertical:



To use the slider control, the user can drag the thumb in one of two directions. If the slider is horizontal, the user can drag the thumb left or right. The thumb of a vertical slider can be dragged up or down. This changes the position of the thumb. The user can also click the desired position along the line to place the thumb at the desired location. Alternatively, when the slider has focus, the user can also use the arrow keys of the keyboard to move the thumb.

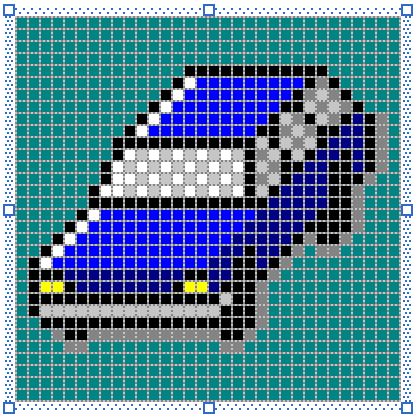
A slider is configured with a set of values from a minimum to a maximum. Therefore, the user can make a selection included in that range. Optionally, a slider can be equipped with small indicators called ticks:

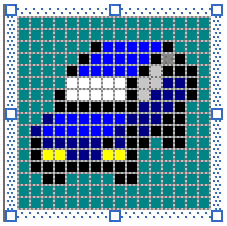


The ticks can visually guide the user for the available positions of the thumb mark. A slider can be used to let the user specify a value that conforms to a range. When equipped with ticks, a slider can be used to control exact values that the user can select in a range, preventing the user from setting just any desired value.

Practical Learning: Introducing Slides

1. Start a new Dialog-based application named **CarInventory1** with no About Box and set the Dialog Title to Car Inventory
2. Redesign the IDR_MAINFRAME icons as follows:





3. Delete the TODO line and the OK button
4. Change the caption of the Cancel button to Close
5. Move the Close button to the bottom-right section of the dialog box
6. On the main menu, click Insert -> Resource or Project -> Add Resource…
7. On the Add Resource dialog box, click Import…
8. From the resources that accompany this book, locate the Cars folder, click Civic and click Open
9. Change the ID of the new bitmap to IDB_CIVIC
10. In the same way, import the Elantra, Escape, Escort, Focus, GdMarquis, Mystique, Navigator, Sentra, and Sephia

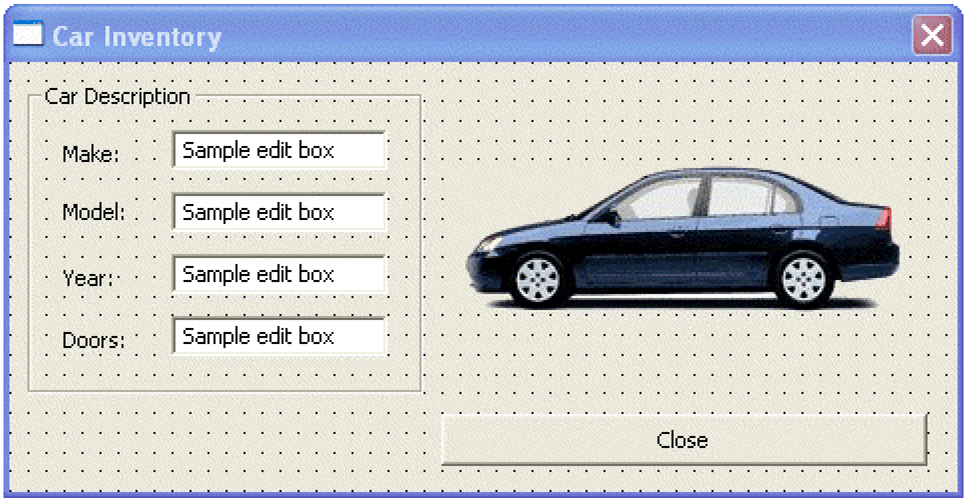
11. Change their IDs to IDB_ELANTRA, IDB_ESCAPE, IDB_ESCORT, IDB_FOCUS, IDB_MARQUIS, IDB_MYSTIQUE, IDB_NAVIGATOR, IDB_SENTRA, and IDB_SEPHIA respectively

12. Add a Picture control to the middle side of the dialog box and change its ID to IDC_PREVIEW

13. Set its **Type** to **Bitmap** and set its Bitmap to IDB_CIVIC

14. Add a Control variable for the picture control and name it **m_Picture**

15. Complete the dialog design as follows:



16. Set the IDs of the edit boxes from top to bottom to **IDC_MAKE**, **IDC_MODEL**, **IDC_YEAR**, and **IDC_DOORS**

17. Add a Value control to each edit box and name them **m_Make**, **m_Model**, **m_Year**, and **m_Doors** respectively

18. To store the values for the controls on the dialog box, in the header file of the dialog box, create a structure named LisOfCars and declare a private array for it in the dialog's class as follows:

```
// CarInventory1Dlg.h : header file
//

#pragma once
// A class for each car
struct CListOfCars
{
	CString Make;
	CString Model;
	UINT	CarYear;
	UINT	Doors;
	UINT	CarPicture;
};
// CCarInventory1Dlg dialog
class CCarInventory1Dlg : public CDialog
{
// Construction
public:
	CCarInventory1Dlg(CWnd* pParent = NULL);	// standard constructor

// Dialog Data
```

```
    enum { IDD = IDD_CARINVENTORY1_DIALOG };

    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support


// Implementation
protected:
    HICON m_hIcon;

    // Generated message map functions
    virtual BOOL OnInitDialog();
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    DECLARE_MESSAGE_MAP()

private:
    CListOfCars Car[10];
};
```

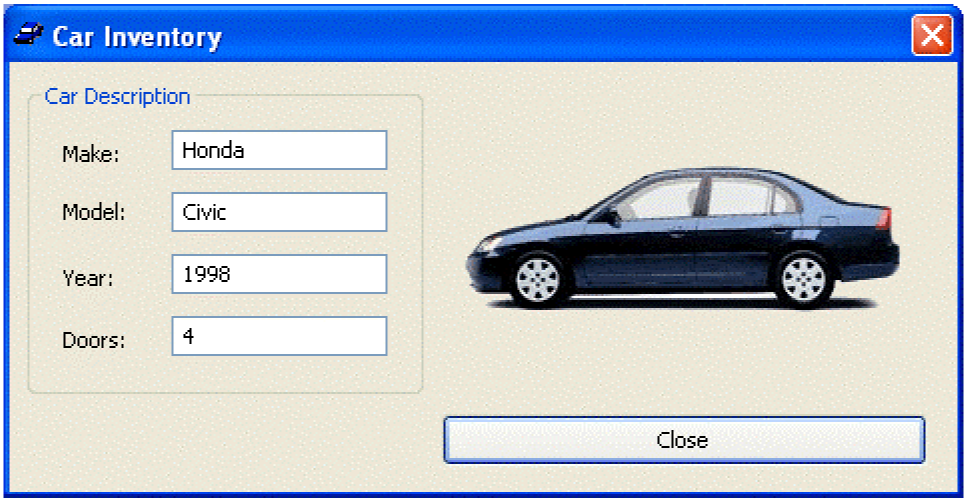
19. To initialize the array, type the following in the OnInitDialog event:

```
BOOL CCarInventory1Dlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);                         // Set big icon
    SetIcon(m_hIcon, FALSE);            // Set small icon

    // TODO: Add extra initialization here
    Car[0].Make = "Honda";
    Car[0].Model = "Civic";
    Car[0].CarYear = 1998;
    Car[0].Doors = 4;
    Car[0].CarPicture = IDB_CIVIC;

    Car[1].Make = "Hyundai";
    Car[1].Model = "Elantra";
    Car[1].CarYear = 1996;
    Car[1].Doors = 4;
    Car[1].CarPicture = IDB_ELANTRA;

    Car[2].Make = "Ford";
    Car[2].Model = "Escape";
    Car[2].CarYear = 2003;
    Car[2].Doors = 5;
    Car[2].CarPicture = IDB_ESCAPE;

    Car[3].Make = "Ford";
    Car[3].Model = "Escort";
    Car[3].CarYear = 1997;
    Car[3].Doors = 2;
    Car[3].CarPicture = IDB_ESCORT;

    Car[4].Make = "Mercury";
    Car[4].Model = "Grand Marquis";
    Car[4].CarYear = 2001;
    Car[4].Doors = 4;
    Car[4].CarPicture = IDB_MARQUIS;

    Car[5].Make = "Mercury";
    Car[5].Model = "Mystique";
    Car[5].CarYear = 2000;
    Car[5].Doors = 4;
    Car[5].CarPicture = IDB_MYSTIQUE;

    Car[6].Make = "Lincoln";
    Car[6].Model = "Navigator";
    Car[6].CarYear = 2003;
    Car[6].Doors = 5;
    Car[6].CarPicture = IDB_NAVIGATOR;

    Car[7].Make = "Nissan";
    Car[7].Model = "Sentra";
    Car[7].CarYear = 1997;
    Car[7].Doors = 2;
    Car[7].CarPicture = IDB_SENTRA;

    Car[8].Make = "Ford";
    Car[8].Model = "Focus";
    Car[8].CarYear = 2002;
    Car[8].Doors = 4;
    Car[8].CarPicture = IDB_FOCUS;

    Car[9].Make = "Kia";
    Car[9].Model = "Sephia";
    Car[9].CarYear = 2003;
    Car[9].Doors = 4;
    Car[9].CarPicture = IDB_SEPHIA;

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

20. To display default values in the dialog box, on top of its OnPaint() event, type the following:

```
void CCarInventory1Dlg::OnPaint()
{
    CBitmap Bmp;

    Bmp.LoadBitmap(Car[0].CarPicture);

    m_Make.Format("%s", Car[0].Make);
    m_Model.Format("%s", Car[0].Model);
    m_Year.Format("%d", Car[0].CarYear);
    m_Doors.Format("%d", Car[0].Doors);
    m_Picture.SetBitmap(Bmp);
    UpdateData(FALSE);

    if (IsIconic())
    {
            . . .
    }
}
```

21. Test the application:



Car Description

Make: Honda

Model: Civic

Year: 1998

Doors: 4

22. Close it and return to MSVC

17.3.2 Slider Creation

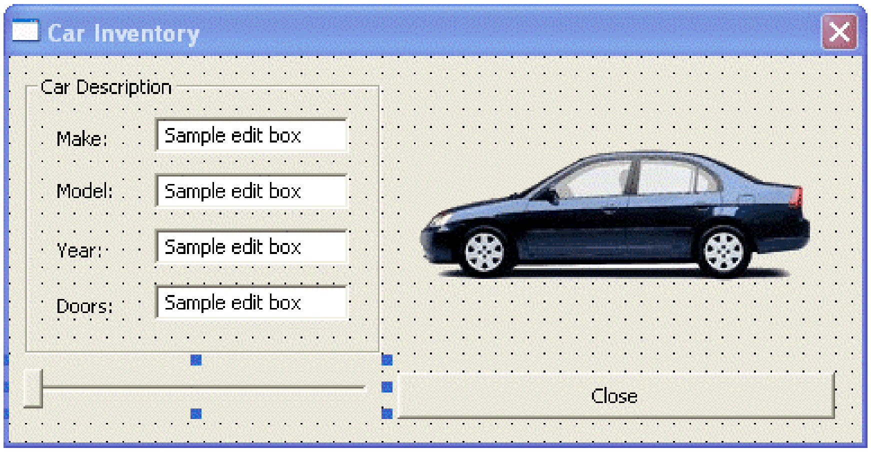
To provide a slider to an application, from the Controls toolbox, click the Slider button and click the desired area on the dialog box or the form.

To programmatically create a slider, declare a pointer to CSliderCtrl using the new operator. To initialize the control, call its **Create()** method. Here is an example:

```
BOOL CDlgSlide::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSliderCtrl *TrackBar = new CSliderCtrl;

	TrackBar->Create(WS_CHILD | WS_VISIBLE,
				CRect(15, 20, 222, 50), this, 0x14);

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

Practical Learning: Creating a Slider

1. On the Controls toolbox, click the Slider button and click in lower-left section of the dialog box



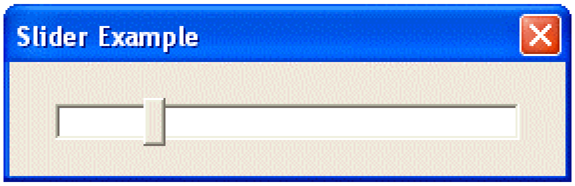
2. Save All

17.3.3 Slider Properties

After placing a slider control on a form or other host, by default, it assumes a horizontal position. If you want a vertical slider, change the value of the **Orientation** property. If you were dynamically creating the control, its default orientation would be horizontal whose style is **TBS_HORZ**. If you want a vertical slider, apply the **TBS_VERT** style:

```
BOOL CDlgSlide::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CSliderCtrl *Taskbar = new CSliderCtrl;

	Taskbar->Create(WS_CHILD | WS_VISIBLE | TBS_VERT,
			CRect(20, 20, 50, 250), this, 0x14);

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

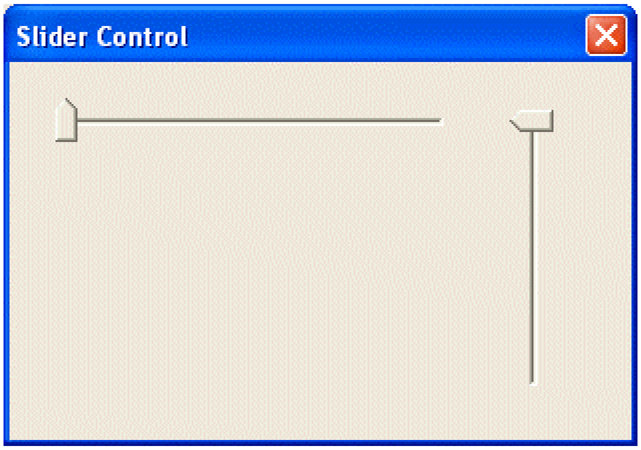
The new slider appears as one line, horizontal or vertical, that guides the user with the area to slide the thumb. When sliding the thumb along the line, the user can set only the value where the thumb is positioned. Alternatively, if you want the user to be able to select a range of values instead of just a value, at design time, you can set the **Enable Selection** property to **True**. This is equivalent to adding the **TBS_ENABLESELECTION** style. A slider equipped with this style displays a 3-D "whole" in the body of the slider:



The selection area allows the user to select a range of values.

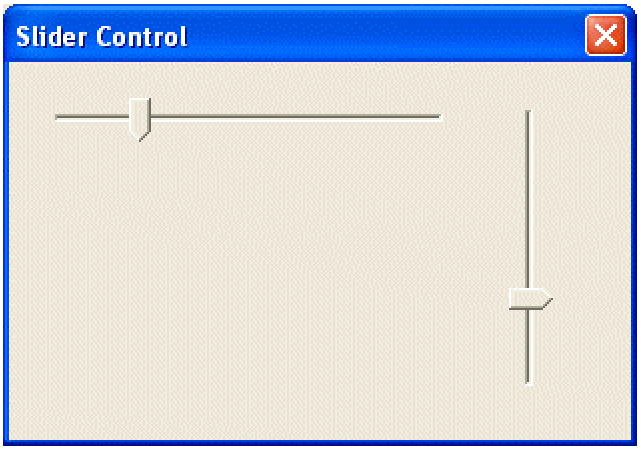
The thumb of a slider can assume one of three shapes. By default, it appears as a rectangular box. Alternatively, you can convert one of its shorter borders to appear as an arrow. The shape of the thumb is controlled at design time by the **Point** property. Its default value is **Both**, which gives it a rectangular shape. You can get this same shape by omitting or adding the **TBS_BOTH** value.

For a horizontal slider, you can make the thumb 's arrow point to the left by changing the Point property to **Top/Left**. If the slider were horizontal, this **Point** value would orient the thumb arrow to the top:



To make the thumb of a dynamically created horizontal slider point up, add the **TBS_TOP**. If the slider is vertical, to point its thumb to the left, add the **TBS_LEFT** style to it.

If you want the thumb to point down for a horizontal slider, set the **Point** property to **Bottom/Right**. This same value would make the thumb of a vertical slider point n the right direction:

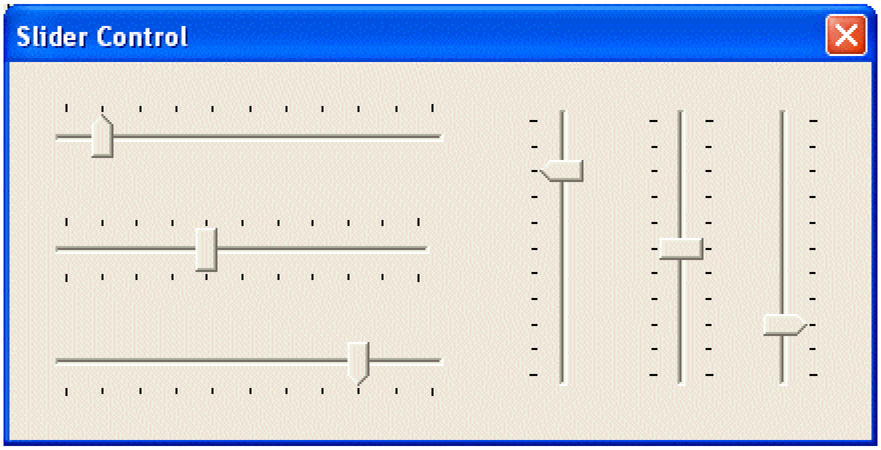


To point the thumb up for a horizontal slider you are programmatically creating, add the **TBS_BOTTOM**. For the thumb of a vertical slider to point right, add the **TBS_RIGHT** to it.

If you want to guide the user with some ticks on the control, at design time, set the **Tick Marks** property to **True**. If you are dynamically creating the slider and you want it to display ticks, simply adding the either the **TBS_VERT** or the **TBS_HORZ** style equips the slider with ticks. If you do not want to display the ticks at all, at design time, clear the **Tick Marks** property or set its value to False.

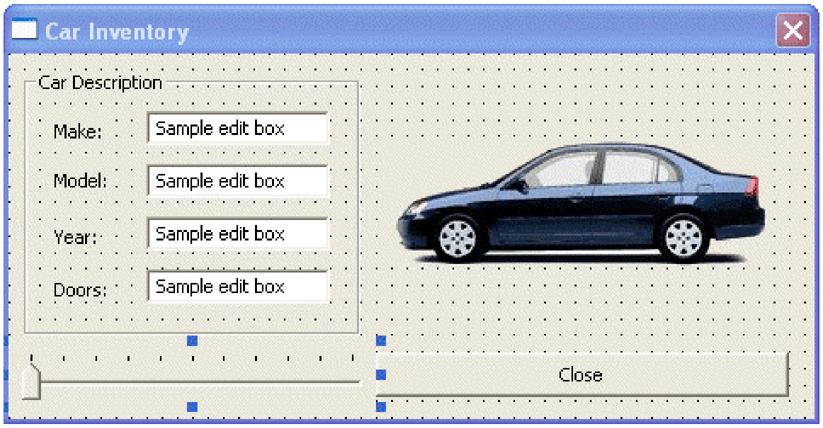
The ticks are positioned on the side the thumb is pointing. If the slider is created with the **Both** value for the **Point** property or the **TBS_BOTH** style, the ticks would appear on both sides the thumb.

The thumb of a slider is used to scroll from one minimum value to another. The range of these extreme values can be divided in regular increments that can further guide the user with value selection. To display where these increments are set, at design time, set the Auto Ticks property to True or add the **TBS_AUTOTICKS** style:



Practical Learning: Designing a Track Bar

1. While the slider control is selected on the dialog box, on the Properties window, change its ID to **IDC_SLIDER**
2. Check its **Auto Ticks** check box or set it to True
3. Check its **Tick Marks** check box or set it to True
4. Set its **Point** property to **Top/Left**



5. Set to True the **Tooltips** property of the slider control
6. Add a Control variable for the slider and name it **m_CarSlider**
7. Test the application and return to MSVC

17.3.4 Slider Methods

As seen earlier, the slider control is based on the **CSliderCtrl** class. Therefore, we saw that we can dynamically create a slider by declaring pointer to **CSliderCtrl** and call its **Create()** method to initialize it. Once a slider has been designed and received the desired style, you can programmatically use it and provide the necessary feedback to the user.

A slider is a control that provides a range of values between which the user can navigate using the control's thumb or by clicking on the slider's line. Usually, the first aspect you may need to configure on your control is to specify its limit values. To specify the minimum value of a slider control, you can call the **CSliderCtrl::SetRangeMin()** method. Its syntax is:

```
void SetRangeMin(int nMin, BOOL bRedraw = FALSE);
```

The *nMin* value is the new minimum value that the slider can assume. The control is typically redrawn once the new value has been set. If you do not want the control to be redrawn, pass a second argument with the FALSE value. If the lowest value of the control has already been set and you want to find out what that value is, you can call the **CSliderCtrl::GetRangeMin()** method. Its syntax is:

```
int GetRangeMin() const;
```

This method simply returns the lowest value that the slider can assume when the user has dragged the thumb to the extreme left or bottom.

To set the highest value of a slider control, you can call the **SliderCtrl::SetRangeMax()** method. Its syntax is:

```
void SetRangeMax(int nMax, BOOL bRedraw = FALSE);
```

The *nMax* argument holds the new maximum value for the control. Here is an example:

```
BOOL CControlsDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	m_Slider.SetRangeMin(0);
	m_Slider.SetRangeMax(50);

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

If the maximum value of the control had previously been set, you can find it out by calling the **SliderCtrl::GetRangeMax()** method. Its syntax is:

```
int GetRangeMax() const;
```

This method returns the highest value that the slider control can have.

To set both the minimum and the maximum values of the slider with one line of code, you can call the **CSliderCtrl::SetRange()** method. Its syntax is:

```
void SetRange(int nMin, int nMax, BOOL bRedraw = FALSE);
```

The *nMin* and the *nMax* arguments hold the lowest and the highest respective values of the control. Here is an example:

```
BOOL CControlsDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        m_Slider.SetRange(0, 50);

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

If the control is already functioning and you want to know its limit values, you can call the **CSliderCtrl::SetRange()** method whose syntax is:

```
void GetRange(int& nMin, int& nMax) const;
```

This method returns two values, namely the lowest value, as *nMin*, and the highest value, as *nMax*.

Once the minimum and maximum values have been set, the user can slide the thumb to select a value or a range. This value is what mostly interests you. While sliding the thumb, the value of the slider is called its position. At startup or at any time, you can set a specific position for the thumb. This can be done by calling the **CSliderCtrl::SetPos()** method. Its syntax is:

```
void SetPos(int nPos);
```

The nPos argument holds the new position of the slider. Here is an example:

```
BOOL CControlsDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        m_Slider.SetRange(0, 50);
        m_Slider.SetPos(32);

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

The value specified using the **SetPos()** method should be in the range *nMax – nMin* of the **SetRange()** method. If there is a possibility that this value is outside the valid range, you can call the **CSliderCtrl::VerifyPos()** method to check it. Its syntax is:

```
void VerifyPos( );
```

When the position of the thumb has change and you want to find out what it is, call the **CSliderCtrl::GetPos()** method whose syntax is:

```
int GetPos() const;
```

If the slider control was specified to let the user select a range, you can define your own selected range at any time by calling the **CSliderCtrl::SetSelection() method**. Its syntax is:

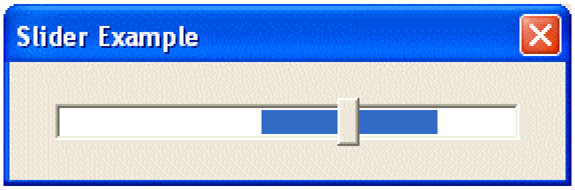
```
void SetSelection(int nMin, int nMax);
```

When calling this method, make sure you specify the nMin and the *nMax* values so that this nMin is greater than the minimum value of the slider and this *nMax* is less than the highest possible value of the slider. Furthermore, the value of this nMin must be less than that of *nMax*. This relationship can be illustrated as follows:

Minimum <= nMin < nMax <= Maximum

Here is an example:

```
BOOL CControlsDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	m_Slider.SetRange(0, 50);
	m_Slider.SetPos(32);
	m_Slider.SetSelection(22, 42);

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



If a selected range has been performed on the slider, if you want to get the minimum and the maximum values of the selection, you can call the **CSliderCtrl::GetSelection()** method whose syntax is:

```
void GetSelection(int& nMin, int& nMax) const;
```

This method returns two values, the minimum as *nMin* and the maximum as *nMax*.

If the slider control is configured to display ticks, you can specify their frequency with a call to the **CSliderCtrl::SetTicFreq()** method. Its syntax is:

```
void SetTicFreq(int nFreq);
```

Practical Learning: Using Track Bar Events

1. In the OnInitDialog event of the dialog class, set the range of values of the slider to 0 to 9 and the frequency of its ticks to:

```
m_CarSlider.SetRange(1, 10);
m_CarSlider.SetTicFreq(1);
```

```
    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. Test the application and return to MSVC

17.3.5 Slider Events

On its own, the slider controls can send three notification messages:

?? The **NM_OUTOFMEMORY** message is sent when the slider has run out of memory and could not complete a task

?? The **NM_RELEASECAPTURE** message is sent when the user releases the mouse on the slider

?? The **NM_CUSTOMDRAW** message is used if you want to draw something on the slider or you want to customize the appearance of the slider beyond what Visual C++ proposes

For its functionality, the slider highly relies on its parent. When the user clicks the thumb or any part of the slider, which causes it to slide, a scroll event is fired. If the slider is horizontal, the **CWnd::OnHScroll()** event is sent. If the slider is vertical, the **CWnd::OnVScroll()** event is sent.

Practical Learning: Scrolling a Slider Control

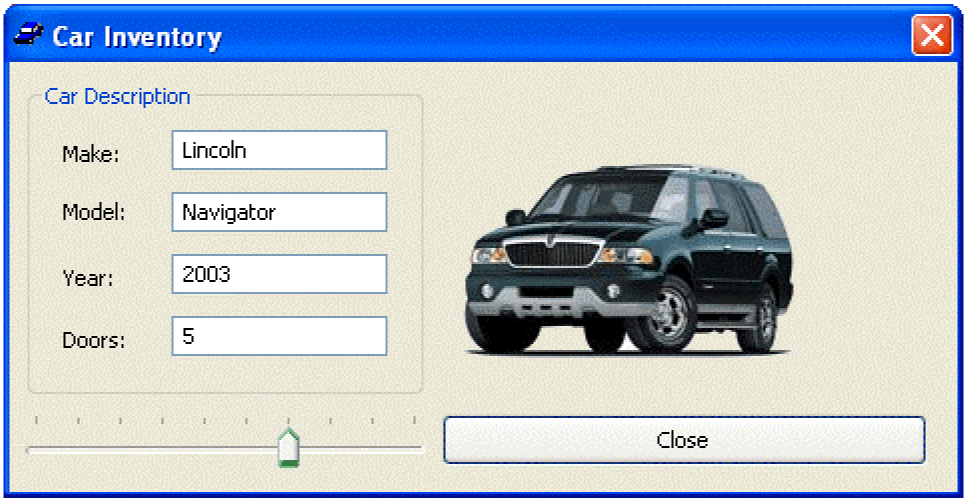
1. Change the **OnPaint()** event of the dialog class as follows:

```
void CCarInventory1Dlg::OnPaint()
{
    int CurPos = m_CarSlider.GetPos() - 1;
    CBitmap Bmp;

    Bmp.LoadBitmap(Car[CurPos].CarPicture);

    m_Make.Format("%s", Car[CurPos].Make);
    m_Model.Format("%s", Car[CurPos].Model);
    m_Year.Format("%d", Car[CurPos].CarYear);
    m_Doors.Format("%d", Car[CurPos].Doors);
    m_Picture.SetBitmap(Bmp);
    UpdateData(FALSE);

    . . .
}
```

2. Using either the ClassWizard (MSVC 6) or the Messages button, for the dialog, generate the **WM_HSCROLL** message and implement it as follows:

```
void CSlider1Dlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    Invalidate();

    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

3. Test the application



Make: Lincoln

Model: Navigator

Year: 2003

Doors: 5

4. Return to MSVC

Chapter 18: Progress-Based Controls

✍ Timers

✍ Progress Controls

✍ Progress Bars

☞ Scroll Bars

☞ Flat Scroll Bars

18.1 Timers

18.1.1 Overview

A timer is a non-spatial object that uses recurring lapses of time from a computer or from your application. To work, every lapse of period, the control sends a message to the operating system. The message is something to the effect of "I have counted the number of lapses you asked me to count".

As opposed to the time set on your computer, a timer is partly but greatly under your control. Users do not see nor use a timer as a control. As a programmer, you decide if, why, when, and how to use this control.

Practical Learning: Introducing the Timer Control

1. Start MSVC and create a new project named **RandShapes**
2. Create it as a **Dialog Based** application with no About box
3. Delete the TODO line, the OK, and the Cancel buttons
4. Set the **Border** style to **None**
5. In the header file of the dialog box, declare two integer variables named **DlgWidth** and **DlgHeight**

```
public:
	int DlgWidth;
	int DlgHeight;
};
```

6. To make the dialog box occupy the whole screen when it comes up, change the OnInitDialog() event as follows:

```
BOOL CRandShapesDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	// when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	DlgWidth = GetSystemMetrics(SM_CXSCREEN);
	DlgHeight = GetSystemMetrics(SM_CYSCREEN);

	SetWindowPos(&wndTopMost, 0, 0, DlgWidth, DlgHeight, SWP_SHOWWINDOW);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

7. Because you cannot close System Menu-less window, to make sure you can close it by moving the mouse, generate the **WM_MOUSEMOVE** message for the dialog box and implement its **OnMouseMove()** event as follows:

```
void CRandShapesDlg::OnMouseMove(UINT nFlags, CPoint point)
{
	// TODO: Add your message handler code here and/or call default
	static int MoveCounter = 0;

	if( MoveCounter >= 20 )
		DestroyWindow();
	MoveCounter++;

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

8. To change the background of the dialog box to black, change the **OnPaint()** event as follows:

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

	GetClientRect(&ScreenRecto);
	CBrush BrushBlack(RGB(0, 0, 0));

	CBrush *pOldBrush = dc.SelectObject(&BrushBlack);

	dc.Rectangle(ScreenRecto);
	dc.SelectObject(pOldBrush);

	if (IsIconic())
	{
		SendMessage(WM_ICONERASEBKGND,
				reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
}
```

9. Test the application and return to MSVC

10. To hide the cursor, at the end of the **OnInitDialog()** event, just before the return line, call the **ShowCursor()** function with a **FALSE** argument.

18.1.2 The Timer Control

Unlike most other controls, the MFC timer has neither a button to represent it nor a class. To create a timer, you simply call the **CWnd::SetTimer()** method. Its syntax is:

```
UINT SetTimer(UINT nIDEvent, UINT nElapse,
              void (CALLBACK EXPORT* lpfnTimer)(HWND, UINT, UINT, DWORD));
```

This function call creates a timer for your application. Like the other controls, a timer uses an identifier. This is passed as the *nIDEvent* argument. As mentioned already, when it is accessed, a timer starts counting up to a set value. Once it reaches that value, it stops and starts counting again. The *nElapse* argument specifies the number of milliseconds that the timer must count before starting again. The *lpfnTimer* argument is the name of a procedure that handles the timing event of the control. This argument can be set to NULL, in which case the timing event would rest on the CWnd's responsibility.

Practical Learning: Using Timer Controls

1. To create a timer, in the **OnInitDialog()** event, before the return line, type **SetTimer(1, 200, 0);**:

```
BOOL CRandShapesDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	DlgWidth = GetSystemMetrics(SM_CXSCREEN);
	DlgHeight = GetSystemMetrics(SM_CYSCREEN);

	SetWindowPos(&wndTopMost, 0, 0, DlgWidth, DlgHeight,
			SWP_SHOWWINDOW);

	SetTimer(1, 200, 0);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. Save All

18.1.3 The Timer Messages and Methods

When a timer is accessed or made available, it starts counting. Once the *nElapse* value of the **CWnd::SetTimer()** method is reached, its sends a **WM_TIMER** message to the application.

We saw that a timer is initiated with a call to **SetTimer()**. When you do not need the timer anymore, call the **CWnd::KillTimer()** method. Its syntax is:

```
BOOL KillTimer(int nIDEvent);
```

The *nIDEvent* argument identifies the timer that was created with a previous call to **SetTimer().**

Practical Learning: Using the OnTimer Event

1. To generate the **OnTimer()** event, in the Class View tab, click the CRandShapesDlg class. In the Properties window, click the Messages button
2. Click the **WM_TIMER** field and click the arrow of its combo box. Select <Add> OnTimer and implement the event as follows:

```
void CRandShapesDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC dc(this);

    int x = (rand() % DlgWidth) + 10;
    int y = (rand() % DlgHeight) + 10;
    CBrush BrushRand(RGB(rand() % 255, rand() % 255, rand() % 255));
    CPen   PenRand(PS_SOLID, 1, RGB(rand() % 255,
                                    rand() % 255, rand() % 255));

    CBrush *pOldBrush = dc.SelectObject(&BrushRand);
    CPen   *pOldPen   = dc.SelectObject(&PenRand);

    switch(rand() % 5)
    {
    case 0:
            dc.Ellipse(x, abs(y-200), abs(y-x), y);
            break;
    case 1:
            dc.Rectangle(y, x, abs(y-x), (x+y)%255);
            break;
    case 2:
            dc.RoundRect(y, x, y, x, abs(x-y), x+y);
            break;
    case 3:
            dc.Ellipse(y, x, abs(x-y), x+y);
            break;
    case 4:
            dc.Rectangle(x, y, abs(x-y), x+y);
            break;
    }

    dc.SelectObject(pOldBrush);
    dc.SelectObject(pOldPen);

    CDialog::OnTimer(nIDEvent);
}
```

3. To generate a random seed and to hide the cursor, in **the OnInitDialog()** event, call **ShorCursor(FALSE)** and **srand()**:

```
BOOL CRandShapesDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);            // Set big icon
    SetIcon(m_hIcon, FALSE);           // Set small icon

    // TODO: Add extra initialization here
    DlgWidth = GetSystemMetrics(SM_CXSCREEN);
    DlgHeight = GetSystemMetrics(SM_CYSCREEN);

    SetWindowPos(&wndTopMost, 0, 0, DlgWidth, DlgHeight,
                 SWP_SHOWWINDOW);

    ShowCursor(FALSE);

    srand((unsigned)time(NULL));

    SetTimer(1, 200, 0);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

4. Test the application and return to MSVC

18.1.4 The Tick Counter

The Win32 library provides a special function used to count a specific number of lapses that have occurred since you started your computer. This information or counter is available through the **GetTickCount()** function. Its syntax is:

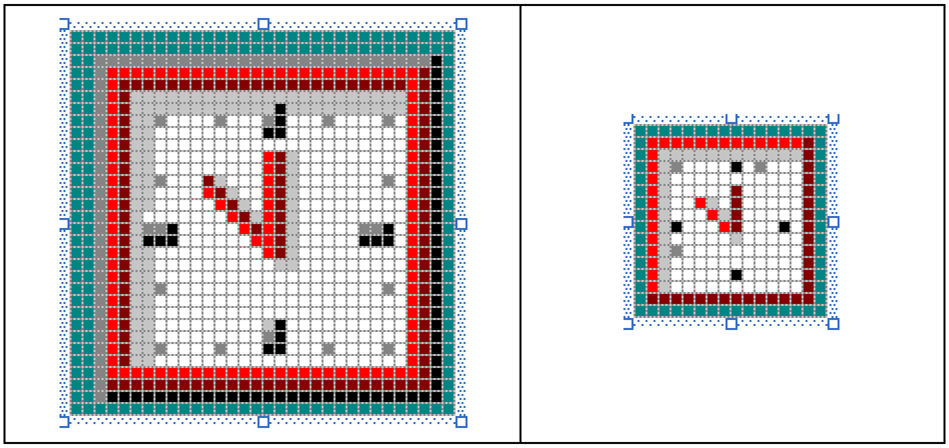
```
DWORD GetTickCount(VOID);
```

This function takes no argument. If it succeeds in performing its operation, which it usually does, it provides the number of milliseconds that have elapsed since you started your computer. Just like the timer control, what you do with the result of this function is up to you and it can be used in various circumstances. For example, computer games and simulations make great use of this function.

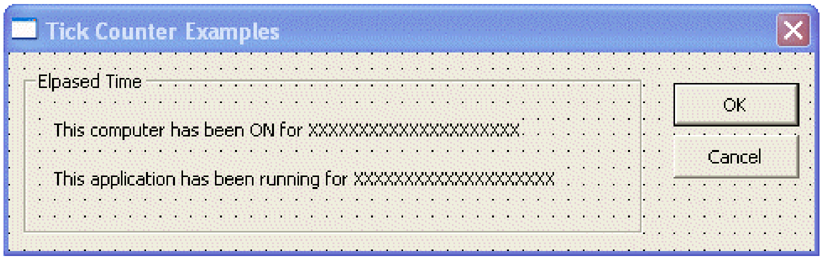
After retrieving the value that this function provides, you can display it in a text-based control.

Practical Learning: Counting the Computer's Ticks

1. Start a new MFC Application named **TickCounter**
2. Create it as Dialog Based with no About Box
3. Change the design of the IDR_MAINFRAME icon as follows:



Here is how we will design the dialog box:



Tick Counter Examples

Elpased Time

This computer has been ON for XXXXXXXXXXXXXXXXXXXX

This application has been running for XXXXXXXXXXXXXXXXXXX

4. Add a Group Box control to the dialog box and set its Caption to Elapsed Time

5. Add a long Static Text control to the group box. Change its ID to **IDC_COMP_TIME** and its Caption to
This computer has been ON for XXXXXXXXXXXXXXXXXXXX

6. Add another Static Text control to the group box. Change its ID to IDC_APP_TIME and its Caption to
This application has been running for XXXXXXXXXXXXXXXXXXX

7. Add a Value Variable for the IDC_COMP_TIME identifier and name it **m_CompTime**

8. Add a Value Variable for the IDC_COMP_TIME identifier and name it **m_AppTime**

9. In the header file of the dialog box, declare an unsigned integer as follows:

```
public:
	CString m_CompTime;
	CString m_AppTime;
	unsigned int CompTime;
};
```

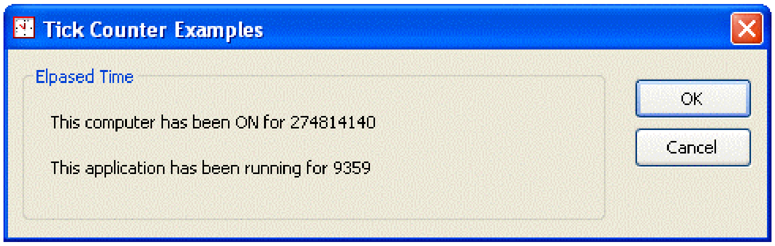
10. To create a timer control, in the OnInitDialog() event, call the SetTimer() method and initialize the CompTime variable as follows:

```
BOOL CTickCounterDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	CompTime = GetTickCount();
	SetTimer(1, 100, NULL);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

11. Generate the **WM_TIMER** message for the dialog class and implement it as follows:

```
void CTickCounterDlg::OnTimer(UINT nIDEvent)
{
	// TODO: Add your message handler code here and/or call default
	unsigned long CurTickValue = GetTickCount();
	unsigned int  Difference    = CurTickValue - CompTime;

	m_CompTime.Format("This computer has been ON for %d", CurTickValue);
	m_AppTime.Format("This application has been running for %d", Difference);
	UpdateData(FALSE);

	CDialog::OnTimer(nIDEvent);
}
```

12. Test the application



This computer has been ON for 274814140

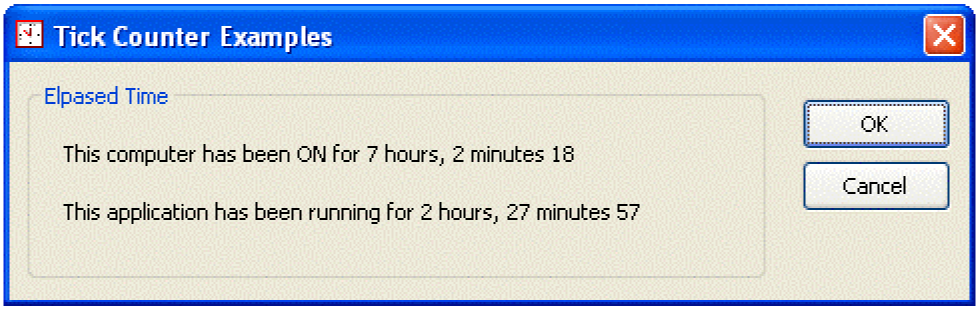
This application has been running for 9359

13. After testing the application, close it and return to MSVC
14. To make the values easier to read, change the code of the OnTimer event as follows:

```
void CTickCounterDlg::OnTimer(UINT nIDEvent)
{
	// TODO: Add your message handler code here and/or call default
	unsigned long CurTickValue = GetTickCount();
	unsigned int  Difference    = CurTickValue - CompTime;

	unsigned int ComputerHours, ComputerMinutes, ComputerSeconds;
	unsigned int ApplicationHours, ApplicationMinutes, ApplicationSeconds;

	ComputerHours       = (CurTickValue / (3600 * 999)) % 24;
	ComputerMinutes     = (CurTickValue / (60 * 999)) % 60;
	ComputerSeconds     = (CurTickValue / 999) % 60;
	ApplicationHours    = (Difference / (3600 * 999)) % 24;
	ApplicationMinutes = (Difference / (60 * 999)) % 60;
	ApplicationSeconds = (Difference / 999) % 60;

	m_CompTime.Format("This computer has been ON for %d hours, %d minutes
%d seconds", ComputerHours, ComputerMinutes, ComputerSeconds);
	m_AppTime.Format("This application has been running for %d hours, %d
minutes %d seconds", ApplicationHours, ApplicationMinutes, ApplicationSeconds);

	UpdateData(FALSE);
```

```
    CDialog::OnTimer(nIDEvent);
}
```

15. Test the application



This computer has been ON for 7 hours, 2 minutes 18

This application has been running for 2 hours, 27 minutes 57

16. After testing the application, close it

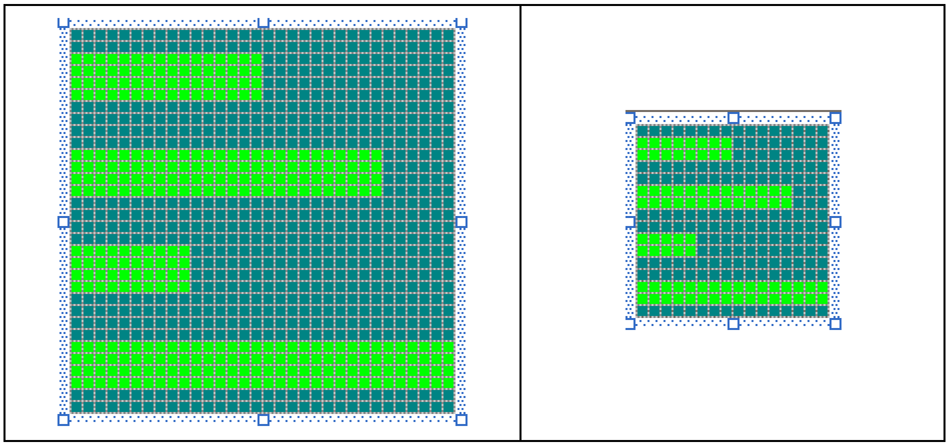
18.2 Progress Controls

18.2.1 Overview

The Progress control is used to display the evolution of an activity, especially for a long operation. Like a label control, a progress control is used only to display information to the user who cannot directly change it.

Practical Learning: Introducing Progress Bars

1. Start a new MFC Application named PClock
2. Create it as Dialog Box without an About Box
3. Delete the TODO line and change the design of the IDR_MAINFRAME as follows:



4. Add three Static Text controls to the left section of the dialog box with captions from top down as Hours:, Minutes:, and Seconds respectively

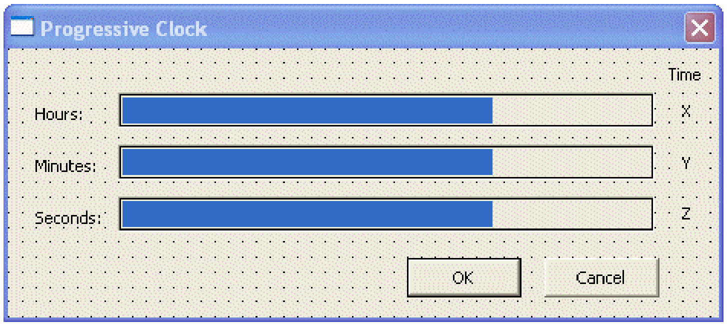
18.2.2 Progress Bar Properties

To create a progress control, on the Controls toolbox, you can click the Progress button and click the desired area on the parent window. The progress control is based on the **CProgressCtrl** class, which can help you dynamically create the control.

After adding a progress control to a host, it assumes a horizontal orientation by default. If you want the progress control to be vertical, check its **Vertical** property or set it to True. If you are dynamically creating the control and you want it vertical, add the **PBS_VERTICAL** style to it.

A progress bar displays regular small rectangles inside a longer or taller rectangle. This outside rectangle serves as their border. If you do not want the border that delimits the control, uncheck its default checked **Border** property or set it to False. The small rectangles appear distinct from one another. If you do not want the small rectangles visible, you can create a smooth progress bar by checking the **Smooth** property or setting it to True. This is equivalent to adding the **PBS_SMOOTH** style.

Practical Learning: Creating a Progress Bar



1. From the Controls toolbox, click the Progress button and click on the right side of the Hours label
2. Using the Properties window, check its Smooth check box or set it to True
3. Change its ID to **IDC_PRGS_HOURS**
4. In the same way, add a Smooth progress bar to the right side of the Minutes label. Change its ID to **IDC_PRGS_MINUTES**
5. Once more, add a Smooth progress control to the right side of the Seconds static control and change its ID to **IDC_PRGS_SECONDS**
6. Add a Static Text control to the right side of the Hours progress control and change its ID to **IDC_VAL_HOURS**
7. Add another Static Text control to the right side of the Minutes progress bar and change its ID to **IDC_VAL_MINUTES**
8. Add another Static Text control to the right side of the Seconds progress bar and change its ID to **IDC_VAL_SECONDS**
9. Add a Static Text to the top right section of the dialog box and set its Caption to **Time**

10. Add a Control Variable for the progress controls and name them from top down as: **m_ProgressHours**, **m_ProgressMinutes**, and **m_ProgressSeconds** respectively
11. Add a **CString** Value Variable for the IDC_VAL_HOURS, the IDC_VAL_MINUTES, and the IDC_VAL_SECONDS identifiers and name them **m_ValueHours**, **m_ValueMinutes**, and **m_ValueSeconds** respectively
12. Test the application and return to MSVC

18.2.3 Progress Control Methods and Events

We mentioned already, an MFC progress bar is based on the **CProgressCtrl** class. Therefore, to dynamically create a progress bar, you can declare a variable or a pointer to **CProgressCtrl** using its constructor. To initialize it, call its **Create()** method and set the necessary characteristics. Here is an example:

```
BOOL CDlgProgress1Dlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        SetIcon(m_hIcon, TRUE);                     // Set big icon
        SetIcon(m_hIcon, FALSE);            // Set small icon

        // TODO: Add extra initialization here
        CProgressCtrl *Progress = new CProgressCtrl;

        Progress->Create(WS_CHILD | WS_VISIBLE, CRect(10, 10, 288, 35), this,0x16);

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

To express its evolution, a progress bar uses values from a minimum to a maximum. These limit values are set using the **CProgressCtrl::SetRange()** or the **CProgressCtrl::SetRange32()** methods. Their syntaxes are:

```
void SetRange(short nLower, short nUpper);
void SetRange32(int nLower, int nUpper);
```

The *nLower* argument sets the minimum value for the control. The *nUpper* argument is the maximum value of the control. Here is an example:

```
BOOL CDlgProgress1Dlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        SetIcon(m_hIcon, TRUE);             // Set big icon
        SetIcon(m_hIcon, FALSE);            // Set small icon

        // TODO: Add extra initialization here
        CProgressCtrl *Progress = new CProgressCtrl;

        Progress->Create(WS_CHILD | WS_VISIBLE, CRect(10, 10, 288, 35), this,0x16);
        Progress->SetRange(1, 100);

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

If the range of values has already been set and you want to find it out, you can call the **CProgressCtrl::GetRange()** method. Its syntax is:

```
void GetRange(int& nLower, int& nUpper);
```

This member function returns two values: *nLower* is the current minimum value of the control while *nUpper* is returned as its maximum value.

Once a progress control has been created and when it comes up, it assumes its minimum value. While the control is working, it keeps changing its value by stepping to the next value. By default, its next value is set to 10. If you want to specify a different incremental value, call the **CProgressCtrl::SetStep()** method whose syntax is:

```
int SetStep(int nStep);
```

The *nStep* argument is the value set to increment the position of the progress control. Once this value is set, when the control is progressing, it increments its value by that value to get to the next step. If you want to explicitly ask it to step to the next incremental value, call the **CProgressCtrl::StepIt()** method. Its syntax is:

```
int StepIt( );
```

Although the user cannot directly modify it, you can change the value of a progress bar by calling the **CProgressCtrl::SetPos()** method whose syntax is:

```
int SetPos(int nPos);
```

This method can be called either to set the initial value of the control or to simply change it at any time. Here is an example:

```
BOOL CDlgProgress1Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	CProgressCtrl *Progress = new CProgressCtrl;

	Progress->Create(WS_CHILD | WS_VISIBLE, CRect(10, 10, 288, 35), this,0x16);
	Progress->SetRange(1, 100);
	Progress->SetPos(38);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

While the control is working, if you want to find out the value it is holding, you can call the **CProgressCtrl::GetPos()** method. Its syntax is:

```
int GetPos( );
```

This method returns the current position of the progress control.

Practical Learning: Using a Progress Bar

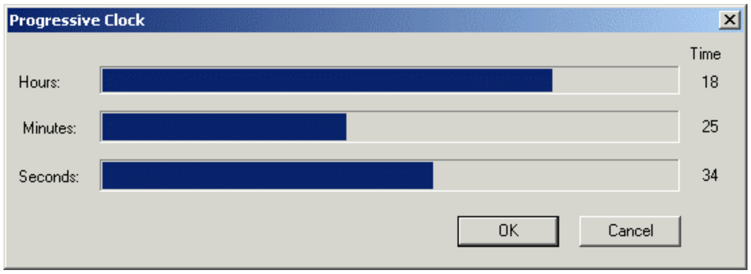
1. To specify the initial values of the progress controls, set their range appropriately in the **OnInitDialog()** event. Also, create a timer control

```
BOOL CPClockDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_ProgressHours.SetRange(0, 23);
	m_ProgressHours.SetStep(1);
	m_ProgressMinutes.SetRange(0, 59);
	m_ProgressMinutes.SetStep(1);
	m_ProgressSeconds.SetRange(0, 59);
	m_ProgressSeconds.SetStep(1);

	SetTimer(1, 40, NULL);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. Generate the **WM_TIMER** message for the dialog box and implement it as follows:

```
void CPClockDlg::OnTimer(UINT nIDEvent)
{
	// TODO: Add your message handler code here and/or call default
	// Get the current time of the computer
	CTime CurTime  = CTime::GetCurrentTime();

	// Find the hour, the minute, and the second values of the time
	int ValHours   = CurTime.GetHour();
	int ValMinutes = CurTime.GetMinute();
	int ValSeconds = CurTime.GetSecond();

	// Change each progress bar accordingly
	m_ProgressHours.SetPos(ValHours);
	m_ProgressMinutes.SetPos(ValMinutes);
	m_ProgressSeconds.SetPos(ValSeconds);

	// Display the position of the progress in the right label
	m_ValueHours.Format("%d", m_ProgressHours.GetPos());
	m_ValueMinutes.Format("%d", m_ProgressMinutes.GetPos());
	m_ValueSeconds.Format("%d", m_ProgressSeconds.GetPos());
	UpdateData(FALSE);

	CDialog::OnTimer(nIDEvent);
}
```

3. Test the application



4. Close it and return to MSVC

18.3 Progress Bars

18.3.1 Introduction

Besides the Progress control, Visual C++ provides two other progress-oriented controls: the Microsoft Progress Control Version 5.0 and the Microsoft Progress Control Version 6.0 with the main difference on their ability to assume one or two orientations.

Practical Learning: Introducing Progress Bars

1. Start a new MFC Application named **BodyMonitor** and create it as Dialog Based
2. Delete the TODO line and move the buttons to the bottom section of the dialog

18.3.2 Creating Progress Bars

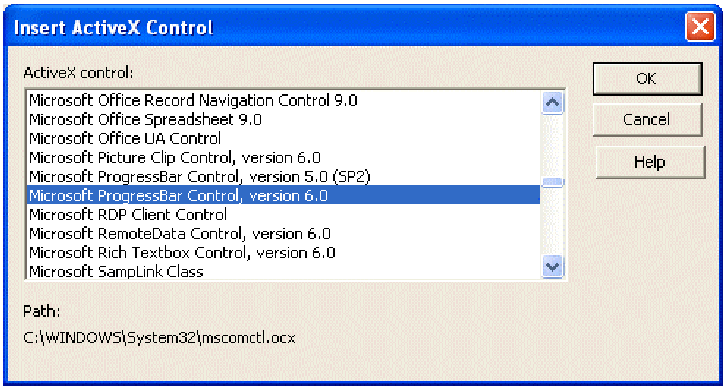
To add a progress bar to your application, from the Insert ActiveX Control dialog box, select the desired one. For this lesson, because everything available in Version 5.0 is also available in Version 6.0, we will use the later.

After adding a progress bar to a parent window, it assumes a horizontal display. This is controlled by the **Orientation** property (not available in Version 5.0) whose default value is **0 – ccOrientationHorizontal**. If you want the progress bar to be vertical, change this property to a **1 – ccOrientationVertical** value.

The range of values that a progress bar can assume is set using the **Min** property for the minimum value and the **Max** field for the highest value.

Practical Learning: Creating Progress Bars

1. Right-click anywhere on the dialog box and click Insert ActiveX Control
2. In the ActiveX Control list of the Insert ActiveX Control dialog box, click Microsoft ProgressBar Control, version 6.0 and click OK



Insert ActiveX Control

ActiveX control:

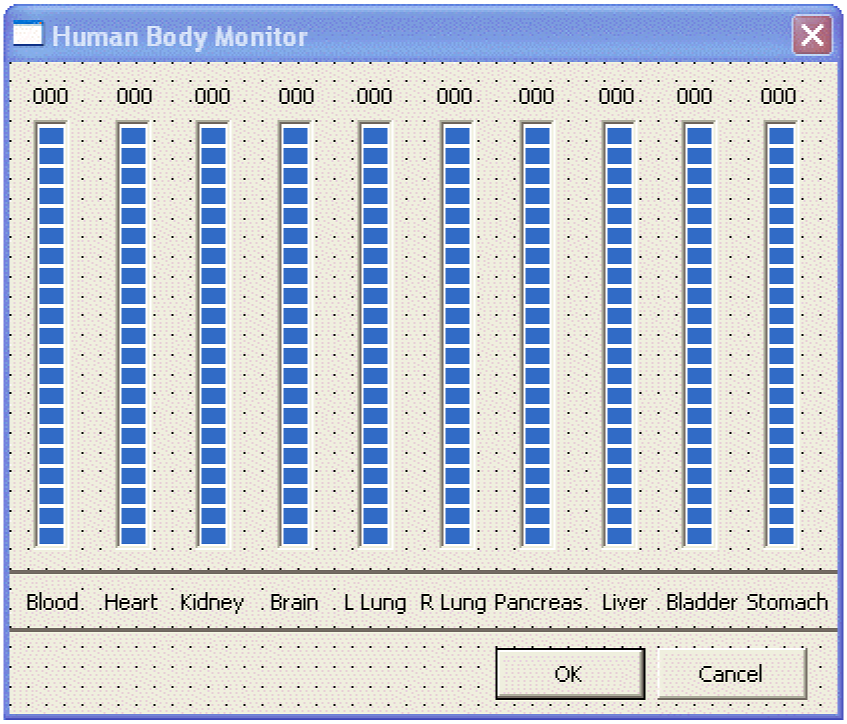
Microsoft Office Record Navigation Control 9.0
Microsoft Office Spreadsheet 9.0
Microsoft Office UA Control
Microsoft Picture Clip Control, version 6.0
Microsoft ProgressBar Control, version 5.0 (SP2)
Microsoft ProgressBar Control, version 6.0
Microsoft RDP Client Control
Microsoft RemoteData Control, version 6.0
Microsoft Rich Textbox Control, version 6.0
Microsoft SampLink Class

Path:
C:\WINDOWS\System32\mscomctl.ocx

3. Using the Properties window, change its Orientation property to **1 – ccOrientationVertical**

4. On the dialog box, right-click the ProgressBar and click Copy. Right-click anywhere on the dialog box and click Paste many times until you get 10 ProgressBar controls

5. Design the dialog box as follows:

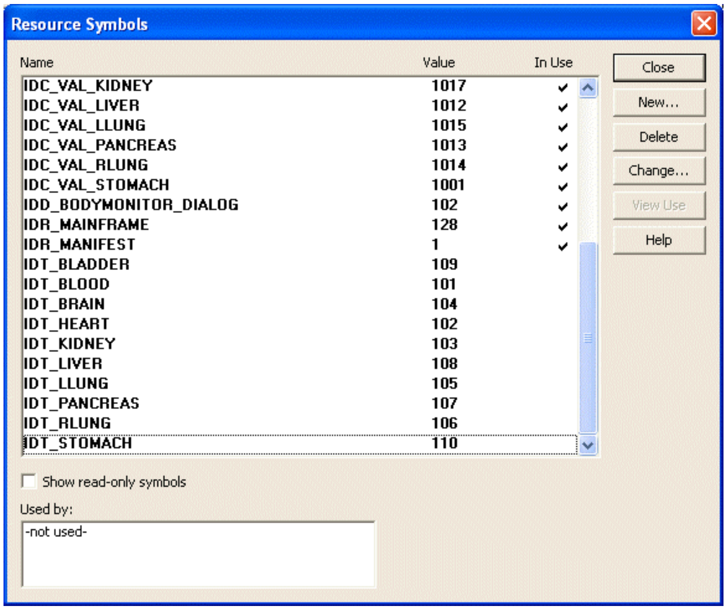


6. Change the IDs of the top Static Text controls and Add a **CString** Value Variable for each. Also, set the IDs of the ProgressBar controls and add a Control Variable for each as follows:

Control	ID to Set	Caption	Align Text	Control Variable	Value Variable
Static Text	IDC_VAL_BLOOD	000	Center		m_ValBlood
Static Text	IDC_VAL_HEART	000	Center		m_ValHeart
Static Text	IDC_VAL_KIDNEY	000	Center		m_ValKidney
Static Text	IDC_VAL_BRAIN	000	Center		m_ValBrain
Static Text	IDC_VAL_LLUNG	000	Center		m_ValLLung
Static Text	IDC_VAL_RLUNG	000	Center		m_ValRLung
Static Text	IDC_VAL_PANCREAS	000	Center		m_ValPancreas

Static Text	IDC_VAL_LIVER	000	Center		m_ValLiver
Static Text	IDC_VAL_BLADDER	000	Center		m_ValBladder
Static Text	IDC_VAL_STOMACH	000	Center		m_ValStomach
ProgressBar	IDC_PRGR_BLOOD			m_Blood	
ProgressBar	IDC_PRGR_HEART			m_Heart	
ProgressBar	IDC_PRGR_KIDNEY			m_Kidney	
ProgressBar	IDC_PRGR_BRAIN			m_Brain	
ProgressBar	IDC_PRGR_LLUNG			m_LLung	
ProgressBar	IDC_PRGR_RLUNG			m_RLung	
ProgressBar	IDC_PRGR_PANCREAS			m_Pancreas	
ProgressBar	IDC_PRGR_LIVER			m_Liver	
ProgressBar	IDC_PRGR_BLADDER			m_Bladder	
ProgressBar	IDC_PRGR_STOMACH			m_Stomach	

7. Using the Resource Symbols dialog box, add ten new identifiers as **IDT_BLOOD**, **IDT_HEART**, **IDT_KIDNEY**, **IDT_BRAIN**, **IDT_LLUNG**, **IDT_RLUNG**, **IDT_PANCREAS**, **IDT_LIVER**, **IDT_BLADDER**, and **IDT_STOMACH**



Name	Value	In Use
IDC_VAL_KIDNEY	1017	✓
IDC_VAL_LIVER	1012	✓
IDC_VAL_LLUNG	1015	✓
IDC_VAL_PANCREAS	1013	✓
IDC_VAL_RLUNG	1014	✓
IDC_VAL_STOMACH	1001	✓
IDD_BODYMONITOR_DIALOG	102	✓
IDR_MAINFRAME	128	✓
IDR_MANIFEST	1	✓
IDT_BLADDER	109	
IDT_BLOOD	101	
IDT_BRAIN	104	
IDT_HEART	102	
IDT_KIDNEY	103	
IDT_LIVER	108	
IDT_LLUNG	105	
IDT_PANCREAS	107	
IDT_RLUNG	106	
IDT_STOMACH	110	

□ Show read-only symbols

Used by:

-not used-

8. In the OnInitDialog() event of the dialog class, create the following timers and generate a random seed:

```
BOOL CBodyMonitorDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	SetTimer(IDT_BLOOD, 650, NULL);
	SetTimer(IDT_HEART, 200, NULL);
	SetTimer(IDT_KIDNEY, 450, NULL);
	SetTimer(IDT_BRAIN, 1000, NULL);
	SetTimer(IDT_LLUNG, 750, NULL);
```

```
    SetTimer(IDT_RLUNG, 850, NULL);
    SetTimer(IDT_PANCREAS, 800, NULL);
    SetTimer(IDT_LIVER, 1200, NULL);
    SetTimer(IDT_BLADDER, 550, NULL);
    SetTimer(IDT_STOMACH, 1500, NULL);

    srand(static_cast<unsigned>(time(NULL)));

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

9. Save All

18.3.3 Progress Bars Methods and Events

The values of a progress bar are float, unlike the progress control that mostly uses integers. To set the minimum value for the progress bar, call its **SetMin()** method. To set the maximum value, call the **SetMax()** method. These two methods expects a float number as argument. If these values have already been set, you can retrieve them by calling the **GetMin()** or the **GetMax()** methods respectively.

To set the initial value of the progress bar or to change its value any timer, you can call the **SetValue()** method. Also, at anytime, you can retrieve the position of the control by calling the **GetValue()** method.

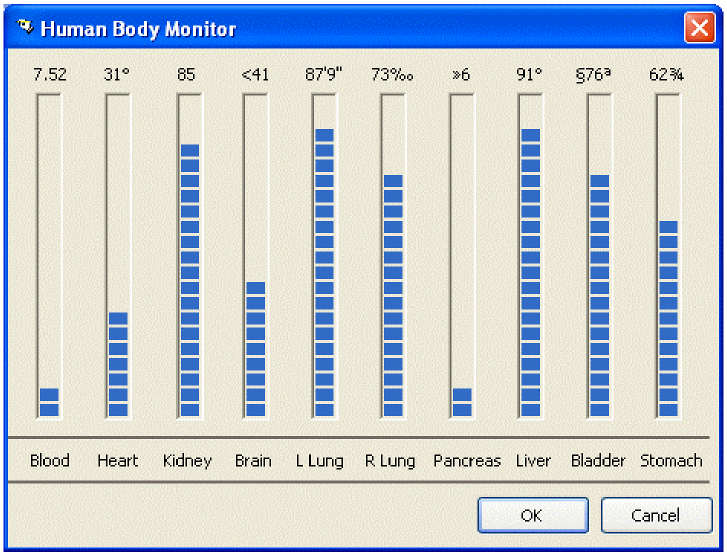
Because the user cannot change the value of a progress bar, it does not fire value-related events. On the other hand, it provides you with the ability to do something when the user clicks or positions the mouse over the control. If you user simply positions the mouse or moves it on top of the progress bar, it fires the **MouseMove()** event. At the same time, if the user presses a mouse button on the progress bar, it fires the **MouseDown()** event. When the user releases the mouse, the **MouseUp()** event is sent. Clicking the progress bar causes it to fire the **Click()** event.

Practical Learning: Using a ProgressBar Control

1. Generate a **WM_TIMER** message for the dialog box and implement it as follows:

```
void CBodyMonitorDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    float Blood = static_cast<float>(rand() % 100);
    float Heart = static_cast<float>(rand() % 100);
    float Kidney = static_cast<float>(rand() % 100);
    float Brain = static_cast<float>(rand() % 100);
    float LeftLung = static_cast<float>(rand() % 100);
    float RightLung = static_cast<float>(rand() % 100);
    float Pancreas = static_cast<float>(rand() % 100);
    float Liver = static_cast<float>(rand() % 100);
    float Bladder = static_cast<float>(rand() % 100);
    float Stomach  = static_cast<float>(rand() % 100);

    if( nIDEvent == IDT_BLOOD )
    {
            m_Blood.put_Value(Blood);
            // Make a sound if the value of the heart is too high
            if( Blood >= 85 )
            MessageBeep(MB_ICONEXCLAMATION);
    }
    if( nIDEvent == IDT_HEART )
        m_Heart.put_Value(Heart);
    if( nIDEvent == IDT_KIDNEY )
        m_Kidney.put_Value(Kidney);
    if( nIDEvent == IDT_BRAIN )
        m_Brain.put_Value(Brain);
    if( nIDEvent == IDT_LLUNG )
        m_LLung.put_Value(LeftLung);
    if( nIDEvent == IDT_RLUNG )
        m_RLung.put_Value(RightLung);
    if( nIDEvent == IDT_PANCREAS )
        m_Pancreas.put_Value(Pancreas);
    if( nIDEvent == IDT_LIVER )
        m_Liver.put_Value(Liver);
    if( nIDEvent == IDT_BLADDER )
        m_Bladder.put_Value(Bladder);
    if( nIDEvent == IDT_STOMACH )
        m_Stomach.put_Value(Stomach);

    m_ValBlood.Format("%.f.%d", m_Blood.get_Value(), rand()%99);
    m_ValHeart.Format("%.f\260", m_Heart.get_Value());
    m_ValKidney.Format("%.f\045", m_Kidney.get_Value());
    m_ValBrain.Format("<%.f", m_Brain.get_Value());
    m_ValLLung.Format("%.f'%d\"", m_LLung.get_Value(), 2+rand()%9);
    m_ValRLung.Format("%.f\261", m_RLung.get_Value());
    m_ValPancreas.Format("\273%.f", m_Pancreas.get_Value());
    m_ValLiver.Format("%.f\260", m_Liver.get_Value());
    m_ValBladder.Format("\247%.f\252", m_Bladder.get_Value());
    CString SCode;
    switch(rand()%3)
    {
    case 0:
        SCode.Format("%s", "\274");
        break;
    case 1:
        SCode.Format("%s", "\275");
        break;
    case 2:
        SCode.Format("%s", "\276");
        break;
    }
    m_ValStomach.Format("%.f%s", m_Stomach.get_Value(), SCode);

    UpdateData(FALSE);

    CDialog::OnTimer(nIDEvent);
}
```

2. Test the application



3. Close it and return to MSVC

18.4 Scroll Bars

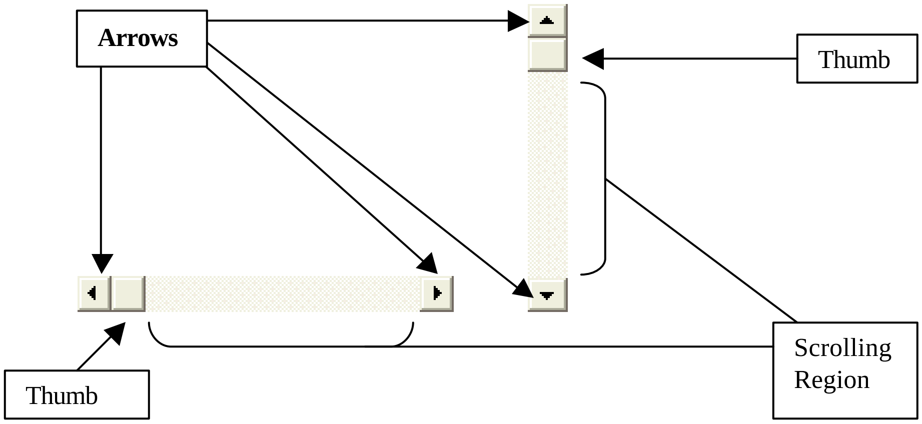
18.4.1 Introduction

A scroll bar is a control that allows the user to navigate a document in two directions along a control by clicking an arrow. This control can assume one of two directions: horizontal or vertical. For its functionality, a scroll bar is made of three parts:

?? An arrow on each side of the control: up and down pointing for a vertical scroll bar, or left and right pointing for a horizontal scroll bar

?? A thumb

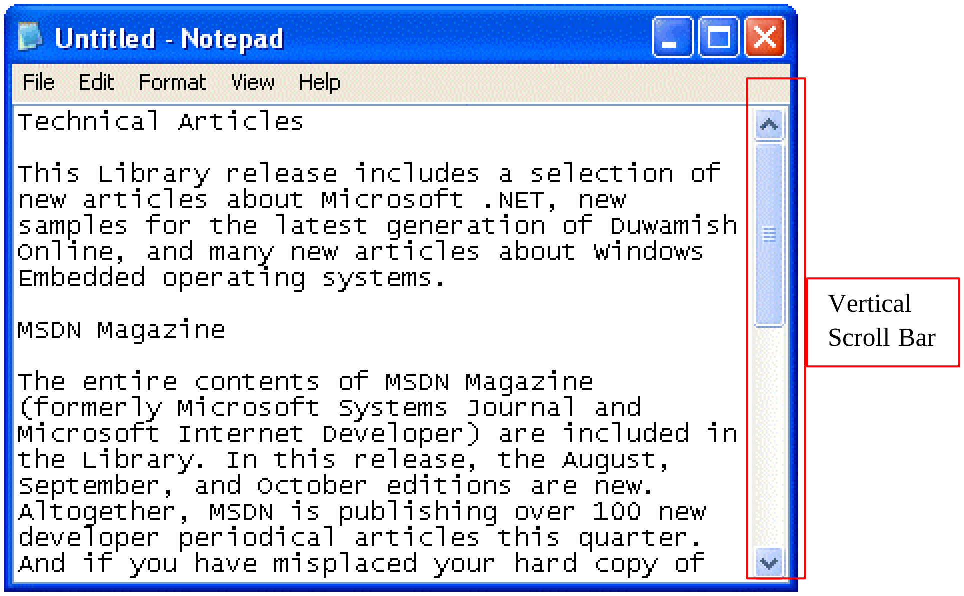
?? A scrolling region



To use the scroll bar, the user can click one of the arrows. This action moves the thumb towards the arrow being clicked. Another way to use a scroll bar is to click and hold the thumb, then slide in the desired direction. Yet another technique the user ca apply consists of clicking in the scrolling region.

18.4.2 Creating Scroll Bars on Views and Dialog Boxes

A scroll bar is usually applied in one of two scenarios. On a document, the view can be equipped with one or two scroll bars. A vertical scroll bar can be positioned to the right side of the view and associated with the frame. Such a scroll bar allows the user to navigate up and down if the view cannot completely display its document. Here is an example of a vertical scroll bar on Notepad:

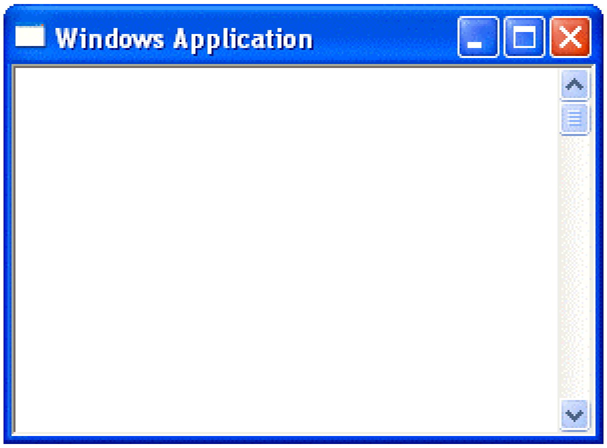


```
Technical Articles

This Library release includes a selection of
new articles about Microsoft .NET, new
samples for the latest generation of Duwamish
Online, and many new articles about Windows
Embedded operating systems.

MSDN Magazine

The entire contents of MSDN Magazine
(formerly Microsoft Systems Journal and
Microsoft Internet Developer) are included in
the Library. In this release, the August,
September, and October editions are new.
Altogether, MSDN is publishing over 100 new
developer periodical articles this quarter.
And if you have misplaced your hard copy of
```

To provide this scroll bar, create the frame with the **WS_VSCROLL** style. Here is an example:

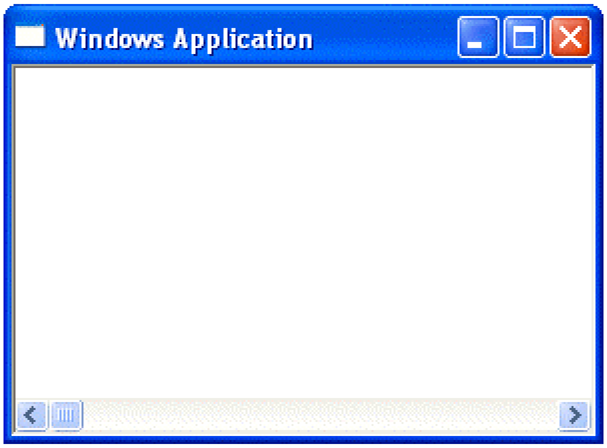
```
#include <afxwin.h>

class CMainFrame : public CFrameWnd
{
public:
        CMainFrame()
        {
                // Create the window's frame
                Create(NULL, "Windows Application",
                        WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                        CRect(120, 100, 420, 320));
        }
};

class CSimpleApp : public CWinApp
{
public:
        BOOL InitInstance()
        {
                m_pMainWnd = new CMainFrame();
```

```
				// Show the window
				m_pMainWnd->ShowWindow(SW_SHOW);
				m_pMainWnd->UpdateWindow();

				return TRUE;
		}
};

CSimpleApp theApp;
```



To equip your frame with a horizontal scroll bar, create it with the **WS_HSCROLL** style:

```
class CMainFrame : public CFrameWnd
{
public:
		CMainFrame()
		{
				// Create the window's frame
				Create(NULL, "Windows Application",
						WS_OVERLAPPEDWINDOW | WS_HSCROLL,
						CRect(120, 100, 420, 320));
		}
};
```

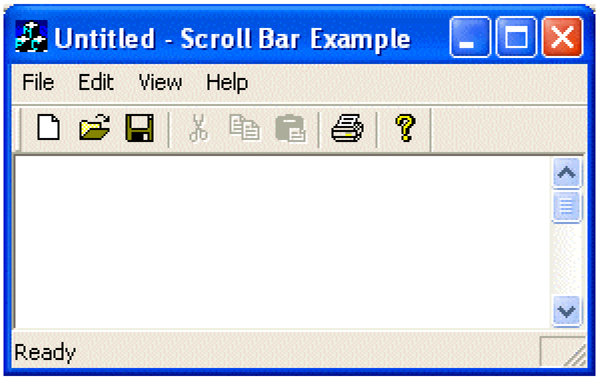


To get both scroll bars, combine both styles:

```
class CMainFrame : public CFrameWnd
{
public:
	CMainFrame()
	{
		// Create the window's frame
		Create(NULL, "Windows Application",
			WS_OVERLAPPEDWINDOW |
			WS_VSCROLL | WS_HSCROLL,
			CRect(120, 100, 420, 320));
	}
};
```

Suppose you created your application using AppWizard. If you created a CView-based application, to add a vertical scroll bar, in your **CView::PreCreateWindow()** event, add the WS_VSCROLL style to the **CREATESTRUCT** object. Here is an example:

```
BOOL CFrame3View::PreCreateWindow(CREATESTRUCT& cs)
{
	// Keep the existing styles but add the WS_VSCROLL value
	cs.style |= WS_VSCROLL;

	return CView::PreCreateWindow(cs);
}
```



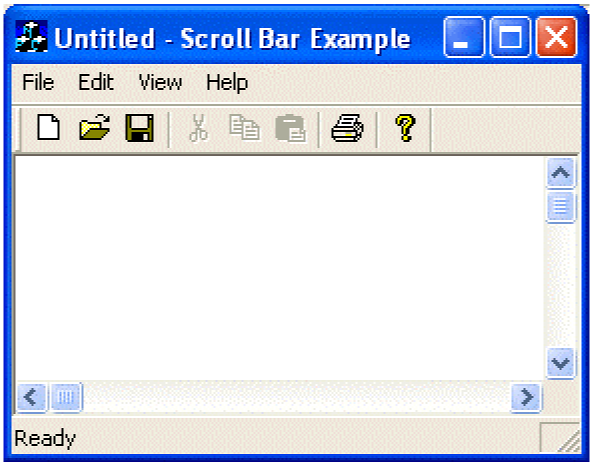
In the same way, you can add only the **WS_HSCROLL** value to get only the horizontal scroll bar:

```
BOOL CFrame3View::PreCreateWindow(CREATESTRUCT& cs)
{
	// Keep the existing styles but add the WS_VSCROLL value
	cs.style |= WS_HSCROLL;

	return CView::PreCreateWindow(cs);
}
```

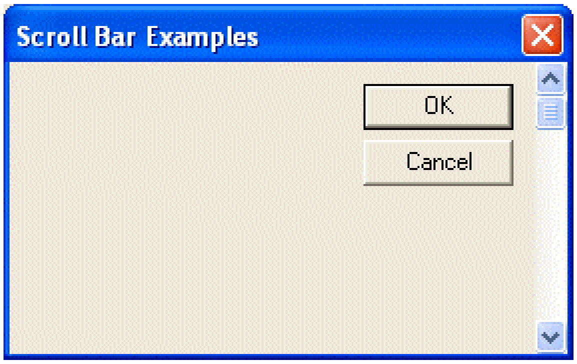
If you want both scroll bars, combine both the WS_VSCROLL and the WS_HSCROLL values:

```
BOOL CFrame3View::PreCreateWindow(CREATESTRUCT& cs)
{
	// Keep the existing styles but add the WS_VSCROLL value
```

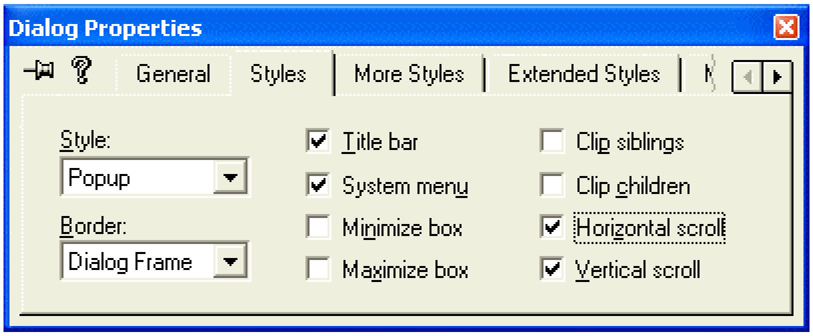
```
        cs.style |= WS_VSCROLL | WS_HSCROLL;

        return CView::PreCreateWindow(cs);
}
```



If you are creating a dialog box and you want to equip it with a vertical scroll bar, at design time, set its **Vertical** Scroll property to **True**:



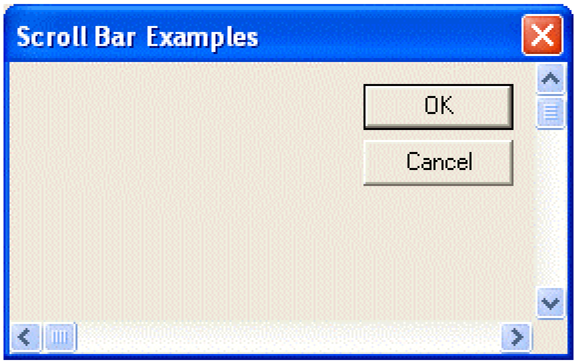
To equip the dialog box with only a horizontal scroll bar, set its Horizontal Scroll property to True. In the same way, to provide both scroll bars to a dialog box, set both properties to True:



Style: Popup

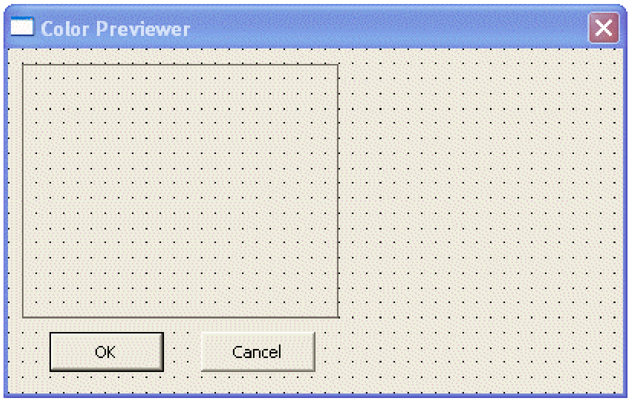
Border: Dialog Frame

☑ Title bar
☑ System menu
☐ Minimize box
☐ Maximize box
☐ Clip siblings
☐ Clip children
☑ Horizontal scroll
☑ Vertical scroll



Practical Learning: Introducing Scroll Bar Controls

1. Start a new MFC Application named **Previewer**
2. Create it as Dialog Based without an About Box
3. Move the OK and the Cancel buttons to the bottom section of the dialog box
4. Add a Picture control and draw a rectangle on the top left section of the dialog box. Change the Picture's ID to **IDC_PREVIEW**



5. Add a CStatic Control Variable for the picture control and name it **m_Preview**
6. In the header file of the dialog, declare three integer variables named ColorRed, ColorGreen, and ColorBlue
7. Also, declare a member function of type **void**and named **PaintPreviewArea**

```
private:
	int ColorRed;
	int ColorGreen;
	int ColorBlue;
public:
	void PaintPreviewArea(void);
};
```

8. Implement the method as follows:

```
void CPreviewerDlg::PaintPreviewArea(void)
{
	CClientDC dc(this);
```

```
    CRect RectPreview;
    // Create a brush to use
    CBrush BrushColor(RGB(ColorRed, ColorGreen, ColorBlue));

    // Get the location and dimensions of the picture control
    m_Preview.GetWindowRect(&RectPreview);
    // Select the brush
    CBrush *pOldBrush = dc.SelectObject(&BrushColor);

    // Use the client coordinates
    ScreenToClient(&RectPreview);
    // Draw or update the background of the picture control
    dc.Rectangle(&RectPreview);

    // Restore the previous brush
    dc.SelectObject(pOldBrush);
}
```

9. In the source file, initialize each member variable with 192

```
CPreviewerDlg::CPreviewerDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CPreviewerDlg::IDD, pParent)
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

    ColorRed   = 192;
    ColorGreen = 192;
    ColorBlue  = 192;
}
```

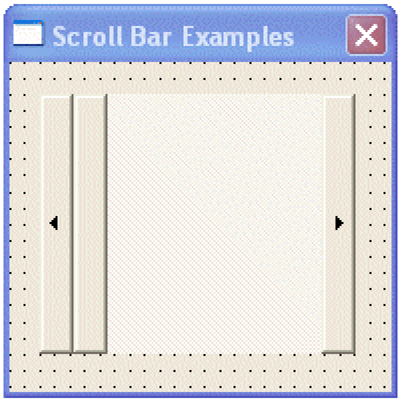
10. Save All

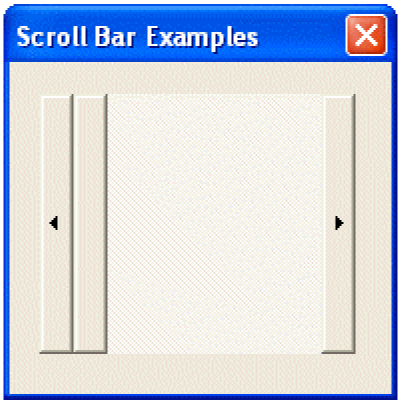
18.4.3 Creating a Scroll Bar Control

The scroll bars we have added to the view classes above are inherently provided by the operating system and can sometimes transparently added to a view of an application with little of your intervention. Alternatively, Visual C++ provides two scroll bar controls you can add to a dialog box or a form and positioned anywhere you judge necessary.

To add a vertical scroll bar control to a dialog box or a form at design time, on the Controls toolbox, click the Vertical Scroll Bar button and click an area on the host. In the same way, to add a horizontal scroll bar control, on the Controls toolbox, click the Horizontal Scroll Bar button and click the body of dialog box or that of the form.

After adding the control, it assumes a default size. You can resize it as you would do with any other control. Here is an example:



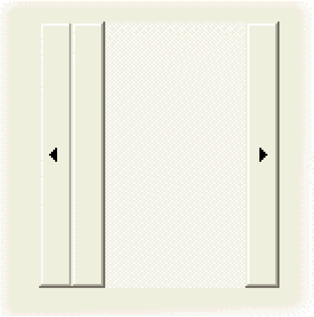


To dynamically create a scroll bar control, declare a pointer to CScrollBar using the the new operator and call its Create() method to initialize it. Here is an example:

```
class CScrollingDlg : public CDialog
{
// Construction
public:
	CScrollingDlg(CWnd* pParent = NULL);   // standard constructor
	virtual ~CScrollingDlg();

	. . .

// Implementation
protected:

	// Generated message map functions
	//{{AFX_MSG(CScrollingDlg)
	virtual BOOL OnInitDialog();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()

private:
	CScrollBar *Scroller;
};

. . .
```

```
// ScrollingDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Frame3.h"
#include "ScrollingDlg.h"

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

/////////////////////////////////////////////////////////////////////////////
// CScrollingDlg dialog
```

```
CScrollingDlg::CScrollingDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CScrollingDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CScrollingDlg)
		// NOTE: the ClassWizard will add member initialization here
	//}}AFX_DATA_INIT

	Scroller = new CScrollBar;
}

CScrollingDlg::~CScrollingDlg()
{
	delete Scroller;
}

. . .

/////////////////////////////////////////////////////////////////////////////
// CScrollingDlg message handlers

BOOL CScrollingDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	Scroller->Create(WS_CHILD | WS_VISIBLE,
			CRect(200, 15, 320, 148), this, 0x16);

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



Practical Learning: Creating Scroll Bars

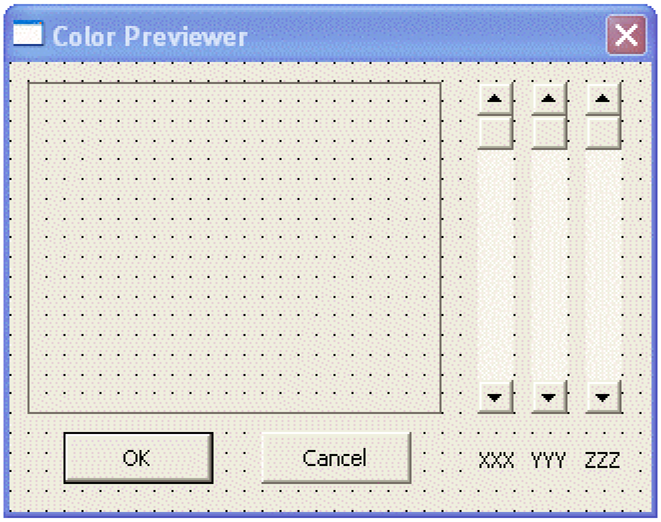
1. On the Controls toolbox, click the Vertical Scroll Bar button and click on the right side of the picture control. Resize it to have the same height as the picture control and change its ID to **IDC_SCROLL_RED**
2. Add a Control Variable for the new scroll bar control and name it **m_ScrollRed**
3. Add a Static Text under the new scroll bar
   CChange its ID to **IDC_VAL_RED**
   Add a CString Value Variable for the new label and name it **m_ValRed**

4. Add another Vertical Scroll Bar to the right of the erxisting one
   Change its ID to **IDC_SCROLL_GREEN**
   Add a Control Variable for the new scroll bar and name it **m_ScrollGreen**

5. Add a Static Text under the new scroll bar
   CChange its ID to **IDC_VAL_GREEN**
   Add a CString Value Variable for the new label and name it **m_ValGreen**

6. Add one more scroll bar control to the right of the others
   Change its ID to **IDC_SCROLL_BLUE**
   Add a Control Variable for it and name it **m_ScrollBlue**

7. Add a Static Text under the new scroll bar
   CChange its ID to **IDC_VAL_BLUE**
   Add a CString Value Variable for the new label and name it **m_ValBlue**

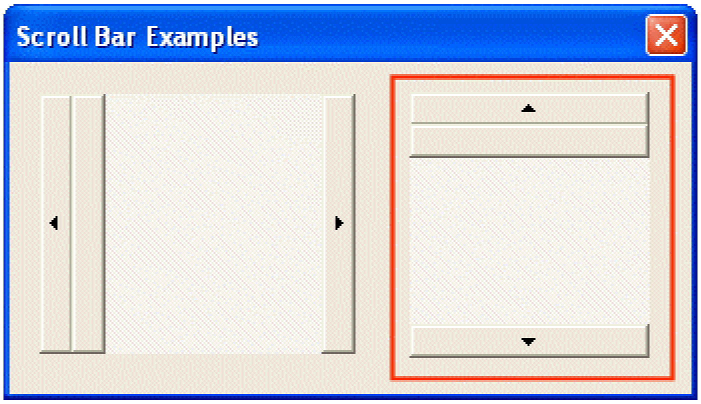


8. Save All

18.4.4 ScrollBar Properties

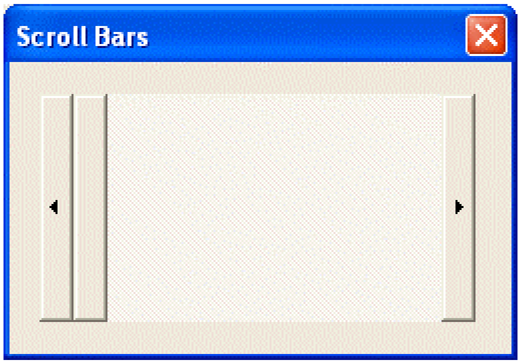
There are two forms of the scroll bars. The horizontal scroll bar allows the user to scroll in the left and right directions. The vertical scroll bar allows scrolling up and down. Visual C++ allows you to visual select the desired one and add to it a host. If you are programmaticall creating the control, you can add the SBS_HORZ style to get horizontal scroll bar. This is also the default, meaning that it you do not specify the orientation, the scroll bar would be horizontal. If you want a vertical scroll bar instead, add the SBS_VERT style:

```
BOOL CScrollingDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here


        Scroller->Create(WS_CHILD | WS_VISIBLE | SBS_VERT,
                                        CRect(200, 15, 320, 148), this, 0x16);

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

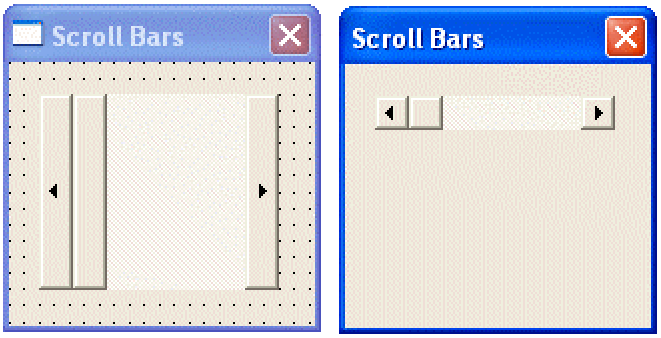


As mentioned already, you can resize a scroll bar control to have the dimensions of your choice. If you are programmatically creating the control, it would assume the size allocated during design or the *nWidth* and the *nHeight* members of the *rect* argument of the **Create()** method. Internally, the operating system has a default width for a vertical scroll bar and a default height for a horizontal scroll bar. If you prefer to use those default values, you have various options.

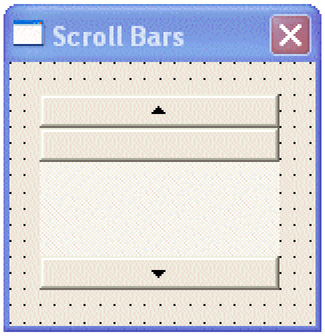
If you are visually creating the scroll bar, change the value of the Align property. Imagine you resize the scroll bar as follows:

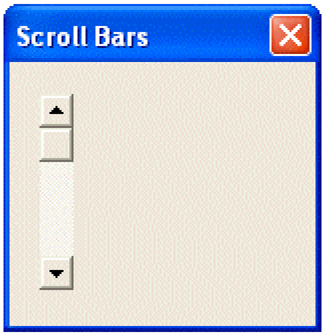


If you set the Align property to **Top/Left** for a horizontal scroll bar, it would keep its top border and bring its border up to get the default height set by the operating system:



If you select this same value for a vertical scroll bar, it would keep its left border and narrow the control to assume the default width of the operating system:

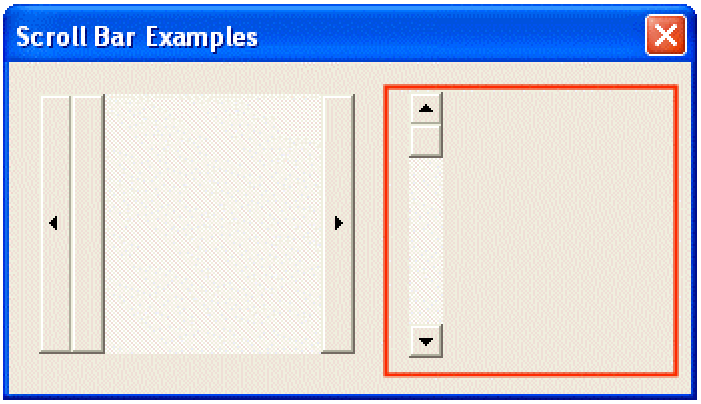




In the same way, you can set the **Align** property to a **Bottom/Right** value to keep the bottom border for a horizontal scroll bar or the right border for a vertical scroll bar and resize them to the default height or width respectively.

If you are programmaticall creating the control, to resize a horizontal scroll bar to assume the default height known to the operating system combine the **SBS_HORZ** with the **SBS_TOPALIGN** and possibly the **SBS_BOTTOMALIGN** values to its style. On the other hand, to get the default width of a vertical scroll bar, besides the **SBS_VERT** style, add the **SBS_LEFTALIGN** and the **SBS_RIGHTALIGN** values:

```
BOOL CScrollingDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here


	Scroller->Create(WS_CHILD | WS_VISIBLE |
			       SBS_VERT | SBS_LEFTALIGN | SBS_RIGHTALIGN,
			      CRect(200, 15, 320, 148), this, 0x16);

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



We mentioned already that the operating system keeps a default height for a horizontal scroll bar and a default width for a vertical scroll bar. To get these default values and apply them to your scroll bar control, call the **GetSystemMetrics()** function and pass it either the **SM_CXVSCROLL** or the **SM_CXHSCROLL** value. Then add the value to the x member of the rect argument for the Create() method. Here is an example:

```
BOOL CScrollingDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	int ButtonWidth = GetSystemMetrics(SM_CXHSCROLL);

	Scroller->Create(WS_CHILD | WS_VISIBLE |
			SBS_VERT,
			CRect(200, 15, 200+ButtonWidth, 148), this, 0x16);

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

18.4.5 Scroll Bar Methods

A scroll bar is based on the **CScrollBar** class. We already that, to programatically create this control, you can declare a pointer to its class and initialize it using its **Create()** method. Since a scroll bar is a value-range control, you must specify its minimum and maximum values from which extremes the thumb can be navigated. To set the range of values of a scroll bar, you can call the **CScrollBar::SetScrollRange()** method. Its syntax is:

```
void SetScrollRange(int nMinPos, int nMaxPos, BOOL bRedraw = TRUE);
```

The *nMinPos* argument specifies the minimum value of the control. The *nMaxPos* argument is the maximum value of the range. The difference between nMaxPos and nMinPos must not be greater than 32,767. When the values have been changed, you may want to refresh the control to reflect the change. This is the default bahavior. If for some reason you do not want the control to be redrawn, pass a third argument as FALSE.

Here is an example:

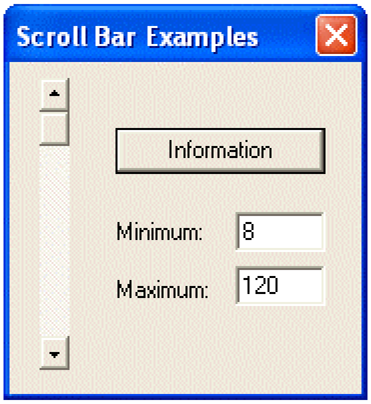
```
BOOL CScrollingDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	m_Scroller.SetScrollRange(8, 120);

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

If the range of a values has already been set for a scroll bar and you want to get its minimum and its maximum possible values, call the **CScrollBar::GetScrollRange()** method. Its syntax is:

```
void GetScrollRange(LPINT lpMinPos, LPINT lpMaxPos) const;
```

This method returns two values, lpMinPos and lpMaxPos. Here is an example:

```
void CScrollingDlg::OnBtnInfo()
{
	// TODO: Add your control notification handler code here
	int Min, Max;

	m_Scroller.GetScrollRange(&Min, &Max);
	m_MinValue.Format("%d", Min);
	m_MaxValue.Format("%d", Max);

	UpdateData(FALSE);
}
```



Once the control has been created and it has the minimum and maximum values set, its thumb would appear in the minimum position. If you want it to assume another position within the range, call the **CScrollBar::SetScrollPos()** method. Its syntax is:

```
int SetScrollPos(int nPos, BOOL bRedraw = TRUE);
```

The *nPos* argument hold the value of the new position for the thumb. This value must be between the *nMinPos* and the *nMaxPos* values of the **SetScrollRange()** method. You may need the control to be redraw after this value has been set to reflect the new change, which is the default behavior. If you do not want the control the control to be redrawn, pass the FALSE value as a second argument.

Here is an example:

```
BOOL CScrollingDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	Scroller->Create(WS_CHILD | WS_VISIBLE |
			SBS_VERT | SBS_LEFTALIGN | SBS_RIGHTALIGN,
			CRect(200, 15, 240, 148), this, 0x16);

	Scroller->SetScrollRange(12, 248);
	Scroller->SetScrollPos(165);

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

If the position of the thumb on a scroll bar control has changed, which happens while the user is scrolling,, you can get the current position of the thumb by calling the **CScrollBar::GetScrollPos()** method. Its syntax is:

```
int GetScrollPos() const;
```

This method returns the position of the thumb on the scroll bar.

To set the minimum and the maximum values of a scroll bar, as well as the initial position of the thumb, you can call the CScrollBar::SetScrollInfo() method. Its syntax is:

```
BOOL SetScrollInfo(LPSCROLLINFO lpScrollInfo, BOOL bRedraw = TRUE);
```

The new values to be passed to this method are stored in a SCROLLINFO variable. Therefore, you should first build that variable. The SCROLLINFO structure is defined as follows:

```
typedef struct tagSCROLLINFO {
    UINT cbSize;
    UINT fMask;
    int  nMin;
    int  nMax;
    UINT nPage;
    int  nPos;
    int  nTrackPos;
}   SCROLLINFO, *LPSCROLLINFO;
typedef SCROLLINFO CONST *LPCSCROLLINFO;
```

The *cbSize* member variable is the size of the structure in bytes. The *fMask* value specifies the type of value that you want to set. The possible values of fMask are:

Value	Description
SIF_RANGE	Used to set only the minimum and the ma ximum values
SIF_POS	Used to set only the initial position of the scroll thumb
SIF_PAGE	Used to set the page size to scroll when using the control
SIF_DISABLENOSCROLL	Used to disable the scroll bar
SIF_ALL	Used to perform all allowed operations

The *nMin* value is used to set the minimum value of the scroll bar while *nMax* specifies its maximum value. The *nPage* value holds the value of the page scrolling. The nPos is used to set the initial position of the thumb. The *nTrackPos* member variable must be ignored if you are setting values for the scroll bar.

Here is an example:

```
BOOL CScrollingDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	SCROLLINFO ScrInfo;

	ScrInfo.cbSize = sizeof(SCROLLINFO);
	ScrInfo.fMask = SIF_ALL;
	ScrInfo.nMin = 25;
```

```
        ScrInfo.nMax = 368;
        ScrInfo.nPage = 2;
        ScrInfo.nPos  = 186;

        m_Scroller.SetScrollInfo(&ScrInfo);

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

If a scroll bar has already been initialized, even it was not initialized using the **SetScrollInfo()** method, to get information about its values, you can call the **CScrollBar::GetScrollInfo().** Its syntax is:

```
BOOL GetScrollInfo(LPSCROLLINFO lpScrollInfo, UINT nMask);
```

This method returns a **SCROLLINFO** value that holds the values of the scroll bar. The possible values of the SCROLLINFO::fMask member variable are:

Value	Description
SIF_RANGE	Used to retrieve the minimum and the maximum values
SIF_POS	Used to retrieves the current position of the scroll thumb. The thumb should not be scrolling when the **GetScrollInfo()** method is called to get the value associated with this mask
SIF_PAGE	Used to retrieve the page size to scroll when using the control
SIF_TRACKPOS	Used to retrieve the current position of the thumb while the user is scrolling
SIF_ALL	Used to retrieve all above values

Practical Learning: Using Scroll Bars

1. Using the OnInitDialog() event, set the range of each scroll bar to (0, 255) and set their initial values to 192

```
BOOL CPreviewerDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // Set the icon for this dialog.  The framework does this automatically
        //  when the application's main window is not a dialog
        SetIcon(m_hIcon, TRUE);                    // Set big icon
        SetIcon(m_hIcon, FALSE);            // Set small icon

        // TODO: Add extra initialization here
        m_ScrollRed.SetScrollRange(0, 255);
        m_ScrollRed.SetScrollPos(63);
        m_ScrollGreen.SetScrollRange(0, 255);
        m_ScrollGreen.SetScrollPos(63);
        m_ScrollBlue.SetScrollRange(0, 255);
        m_ScrollBlue.SetScrollPos(63);
        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. Save All

18.4.6 Scroll Bar Events

As described already, to use a scroll bar, the user clicks either one of its buttons, a thumb or the scrolling region. When the user clicks, the scroll bar sends a message to its parent or host, which can be a view, a dialog box, or a form. If the scroll bar is horizontal, the message sent is **WM_HSCROLL**, which fires an **OnHScroll()** event. The **CWnd** class, as the parent of all MFC Windows controls and views, carries this event and makes it available to the child classes (and controls) that can perform the scrolling. The syntax of the **OnHScroll()** event is:

```
afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
```

The *nSBCode* argument is the type of scrolling that is being performed. Its possible values are:

Value	Description	Role
SB_THUMBTRACK	The thumb is being dragged to a specific position	
SB_LEFT	The thumb has been positioned to the minimum value of the scroll bar, to the extreme left.	Used to position the thumb to the lowest value of the range
SB_LINELEFT	The thumb is scrolling to the left one, character or column at a time	Used to decrease the position of the thumb, usually by subtracting 1 to the current value, unless the thumb is already positioned there
SB_PAGELEFT	The thumb is scrolling to the left, one page at a time	Refers to the page size of the SCROLLINFO value to decrease the current position by one page
SB_PAGERIGHT	The thumb is scrolling to the right, one page at a time	Refers to the page size of the SCROLLINFO value to increase the current position by one page
SB_LINERIGHT	The thumb is scrolling to the right one, character or column at a time	Used to increase the position of the thumb, usually by adding 1, unless the thumb is already positioned there
SB_RIGHT	The thumb has been positioned to the maximum value of the scroll bar, to the extreme right	Used to position the thumb to the highest value of the range, unless the thumb is already positioned there
SB_THUMBPOSITION	Absolute postion	
SB_ENDSCROLL	End Scroll	Used to determine the end of the scrolling

The *nPos* argument is used in connection with the **SB_THUMBTRACK** and the **SB_THUMBPOSITION** values of the nSBCode argument. The value of nPos specifies how much scrolling should be done.

The *pScrollBar* argument can be used to identify the particular scroll bar control needs to be dealt with on this **OnHScroll()** event.

If your scroll bar control is vertical, when the user clicks it, it sends a **WM_VSCROLL** message which fires the **OnVScroll()** event. Its syntax is:

```
afx_msg void OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar);
```

The *nSBCode* argument is the type of scrolling that is being performed. Its possible values are:

Value	Description	Role
SB_THUMBTRACK	The thumb is being dragged to a specific position	
SB_UP	The thumb has been positioned to the maximum value of the scroll bar, to the extreme right	Used to position the thumb to the highest value of the range, unless the thumb is already positioned there
SB_LINEUP	The thumb is scrolling up, one line at a time	Used to increase the position of the thumb, usually by adding 1, unless the thumb is already positioned there
SB_PAGEUP	The thumb is scrolling up, one page at a time	Refers to the page size of the SCROLLINFO value to increase the current position by one page
SB_PAGEDOWN	The thumb is scrolling to the bottom, one page at a time	Refers to the page size of the SCROLLINFO value to decrease the current position by one page
SB_LINEDOWN	The thumb is scrolling to the bottom, one line at a time	Used to decrease the position of the thumb, usually by subtracting 1 to the current value, unless the thumb is already positioned there
SB_BOTTOM	The thumb has been positioned to the minimum value of the scroll bar, to the bottom side of the scroll bar	Used to position the thumb to the lowest value of the range
SB_THUMBPOSITION	Absolute postion	
SB_ENDSCROLL	End Scroll	Used to determine the end of the scrolling

The *nPos* argument is used in connection with the **SB_THUMBTRACK** and the **SB_THUMBPOSITION** values of the nSBCode argument. The value of nPos specifies how much scrolling should be done.

The *pScrollBar* argument can be used to identify the particular scroll bar control needs to be dealt with on this **OnHScroll()** event.

Practical Learning: Using the Scroll Bar Events

1. Generate the **WM_VSCROLL** message of the dialog box and implement it as follows:

```
void CPreviewerDlg::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
	// TODO: Add your message handler code here and/or call default
	int CurPos = pScrollBar->GetScrollPos();

	switch (nSBCode)
	{
	case SB_TOP:
		CurPos = 0;
		break;

	case SB_LINEUP:
		if (CurPos > 0)
```

```
				CurPos--;
			break;

		case SB_LINEDOWN:
			if (CurPos < 255)
				CurPos++;
			break;

		case SB_BOTTOM:
			CurPos = 255;
			break;

		case SB_ENDSCROLL:
			break;

		case SB_THUMBPOSITION:
			CurPos = nPos;
			break;

		case SB_THUMBTRACK:
			CurPos = nPos;
			break;
		}

		// Set the new position of the thumb (scroll box).
		pScrollBar->SetScrollPos(CurPos);

		// Update each color based on the positions of the scroll bar controls
		ColorRed   = 255 - m_ScrollRed.GetScrollPos();
		ColorGreen = 255 - m_ScrollGreen.GetScrollPos();
		ColorBlue  = 255 - m_ScrollBlue.GetScrollPos();

		CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

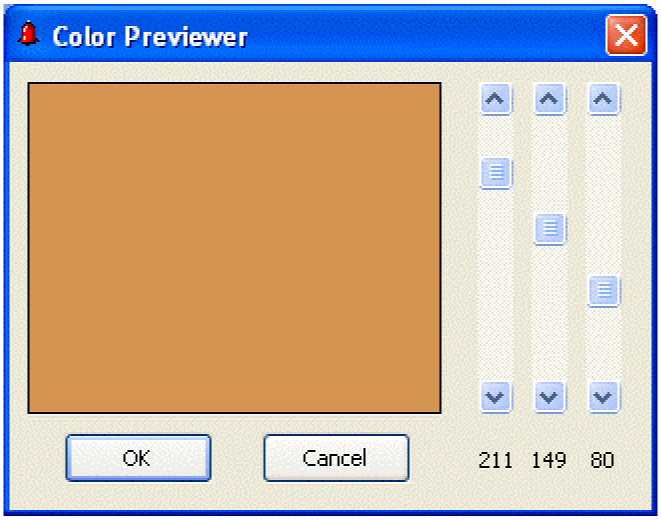
2. Create a timer in the **OnInitDialog()** event as follows:

```
	SetTimer(1, 40, NULL);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

3. Generate a **WM_TIMER** message for the dialog box and implement it as follows:

```
void CPreviewerDlg::OnTimer(UINT nIDEvent)
{
	// TODO: Add your message handler code here and/or call default
	PaintPreviewArea();

	CDialog::OnTimer(nIDEvent);
}
```

4. Test the application

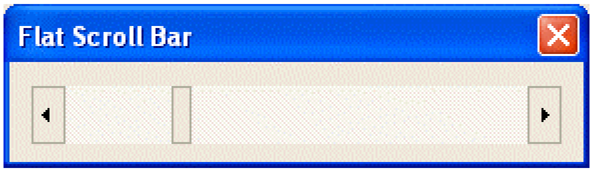


5. Close it and return to MSVC

18.5 Flat Scroll Bars

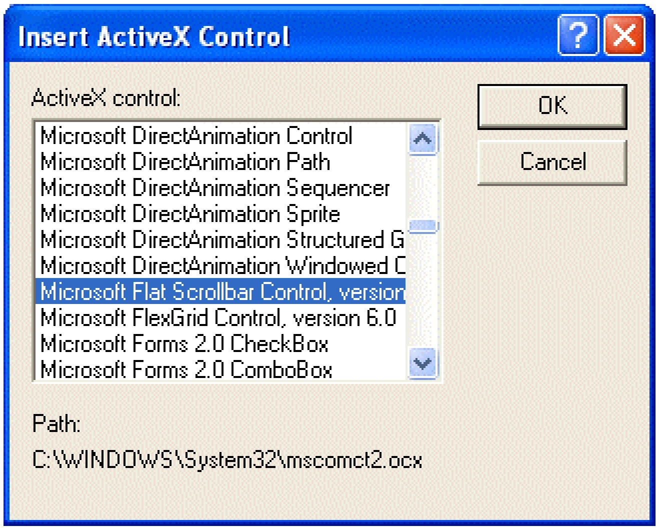
18.5.1 Overview

Besides the Controls toolbox' regular scroll bar controls, the Win32 and the MFC libraries provide a more elaborate and friendlier object called the flat scroll bar. This control provides the same functionality and and features as the above scroll bars but tends to be friendlier.



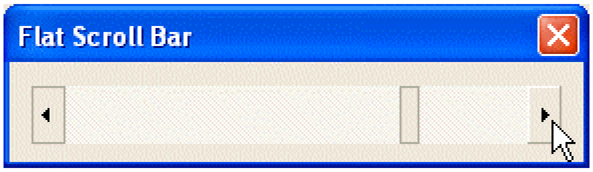
18.5.2 Flat Scroll Bar Properties

To add a flat scroll bar control to your application, from the Insert ActiveX dialog box, double-click Microsoft Flat Scroll Bar Control


Insert ActiveX Control
ActiveX control:
Microsoft DirectAnimation Control
Microsoft DirectAnimation Path
Microsoft DirectAnimation Sequencer
Microsoft DirectAnimation Sprite
Microsoft DirectAnimation Structured G
Microsoft DirectAnimation Windowed C
Microsoft Flat Scrollbar Control, version
Microsoft FlexGrid Control, version 6.0
Microsoft Forms 2.0 CheckBox
Microsoft Forms 2.0 ComboBox
OK
Cancel

Path:
C:\WINDOWS\System32\mscomct2.ocx

When a Microsoft Flat Scroll Bar control has been added to a parent window, it appears flat, its buttons and its thumb have no borders. This display is controlled by the **Appearance** property whose default value is **1 – fsbFlat**. With this value, a simple line is drawn around each button and the central thumb. To add a visual effect, you can change the **Appearance's** value to the **2 – fsbTrack3D** value. In this case, when the mouse is positioned on top of a button or the thumb, the button or the thumb's borders are raised, creating a 3-D effect:



If you prefer the classic 3-D look with borders permentently raised on the buttons and the thumb, set the Appearance to **0 – fsb3D**.

By default, a newly added flat scroll control is oriented horizontally on its parent window and display a left and a right pointing arrow buttons. The direction of this control is set using the **Orientation** combo box. The default value is **1 – cc2OrientationHorizontal**. If you want a vertical scroll bar, change the Orientation value to **0 - cc2OrientationVertical**.

Like the regular scroll bar, this flat control displays a button equipped with an arrow at each end and a thumb in the middle. The user can click or hold a button to scroll in the desired direction. The flat scroll bar control allows you to decide what arrow button the user would be allowed to use. This can be controlled using the **Arrows** property. Because the user can scroll by default in both directions, this property has a **0 – cc2Both** value. If you do not want the user to scroll left on a horizontal scroll bar or to scroll up on a vertical scroll bar, you can set the **Arrows** property to **1 – cc2LeftUp**. In the same way, to prevent the user from scrolling in the right direction for a horizontal scroll bar or from scrolling in the bottom direction for a vertical scroll bar, set the **Arrows** property to **2 – cc2RightDown**.

The limiting values of the flat scroll bar are set using the **Min** field for the lowest value and the **Max** field for the highest value. The value must be in the range of a short integer, from 0 to 32767. The **Min** value should be set lower than the **Max** value.

When a newly added scroll bar has been added to a parent window, it gets the position of its lowest or **Min** value. To set a different initial value, use the **Value** field.

One of the ways the user explores a scroll bar is by clicking one of the arrow buttons. This causes the middle thumb to move towards the arrow being clicked. By default, one click corresponds to one unit, which could be one line of text. To control how much value should be incremented when the user clicks one of the arrow buttons or presses the arrow keys, change the value of the **SmallChange** property.

Another way the user can use a scroll bar is by clicking between an arrow button and the thumb or by pressing either Page Up or Page Down. To control how many units should be covered by this move, change the value of the **LargeChange** property.

18.5.3 Flat Scroll Bar Methods and Events

The Microsoft Scroll Bar Control is based on the CFlatSB class which itself is derived from CWnd.

If the user clicks or holds one of the arrow buttons to scroll, the Flat Scroll Bar fires the **Scroll()** event

When the user clicks or holds one of the buttons, the value of the scroll bar changes. The value changes also if the user drags the thumb. This change of the Value property causes the Flat Scroll Bar to fire the **Change()** event.

Chapter 19: Selection-Based Controls

- ✍ Radio Buttons
- ✍ Check Boxes

19.1 Radio Buttons

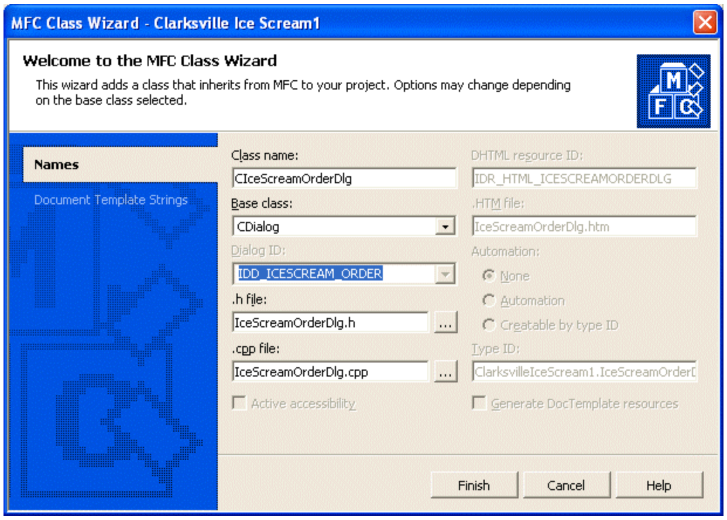
19.1.1 Introduction

A radio button is a control that appears as a (proportionately big) dot surrounded by a round box ? . In reality, a radio button is accompanied by one or more other radio buttons that appear and behave as a group. The user decides which button is valid by selecting only one of them. When he or she clicks one button, its round box fills with a (big) dot: ? . When one button is selected, the other round buttons of the (same) group are empty ? . The user can select another by clicking a new choice, which empties the previous selection. This technique of selecting is referred to as mutually-exclusive.

To indicate what a radio butto is used for, each one of them is accompanied by a label. The label is simply a string and it can be positioned to the left, above, to the right, or under the round box.

Practical Learning: Introducing Radio Buttons

1. Start Microsoft Visual Studio or Visual C++ if necessary.
   Open the **Clarksville Ice Scream1** application. If you did not create, then create a Single Document Interface application named Clarksville Ice Scream1
2. To add a new object, display the Add Resource dialog box and double-click Dialog
3. Change the Caption of the dialog box to
   Clarksville Ice Scream – Customer Order
4. Change its ID to **IDD_ICESCREAM_ORDER**
5. Add a class for the new dialog box and name it CIceScreamOrderDlg
   Make sure the class is based on CDialog



MFC Class Wizard - Clarksville Ice Scream1

Welcome to the MFC Class Wizard

This wizard adds a class that inherits from MFC to your project. Options may change depending on the base class selected.

Class name: CIceScreamOrderDlg

DHTML resource ID: IDR_HTML_ICESCREAMORDERDLG

Base class: CDialog

.HTM file: IceScreamOrderDlg.htm

Dialog ID: IDD_ICESCREAM_ORDER

Automation: ● None ○ Automation ○ Creatable by type ID

.h file: IceScreamOrderDlg.h

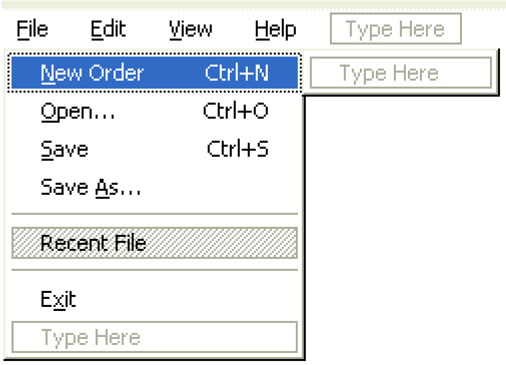
.cpp file: IceScreamOrderDlg.cpp

Type ID: ClarksvilleIceScream1.IceScreamOrderD

□ Active accessibility

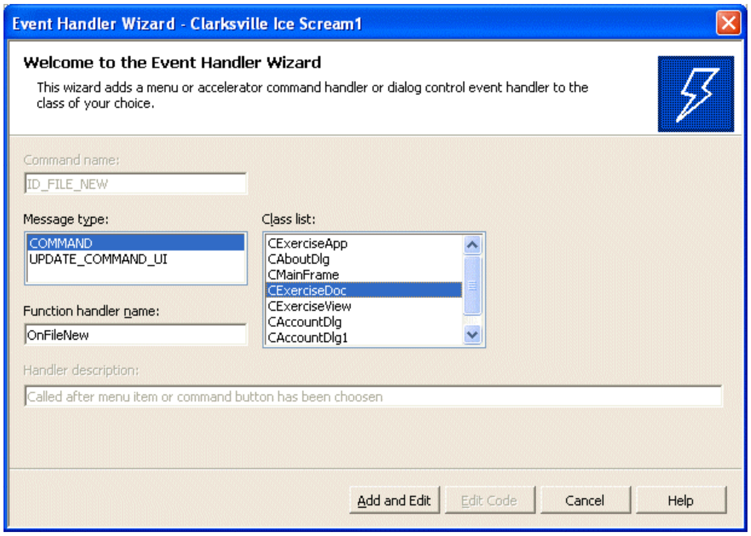
□ Generate DocTemplate resources

6. Press Enter
7. Display the menu editor and change the caption of the first menu item under File from &New\tCtrl+N to &New Order…\tCtrl+N



8. Change its Prompt to **Process a new ice scream order\nNew**

9. Add an Event Handler to the New Order item associated with the document class:



Event Handler Wizard - Clarksville Ice Scream1

Welcome to the Event Handler Wizard

This wizard adds a menu or accelerator command handler or dialog control event handler to the class of your choice.

Command name: ID_FILE_NEW

Message type: COMMAND, UPDATE_COMMAND_UI (COMMAND selected)

Class list: CExerciseApp, CAboutDlg, CMainFrame, CExerciseDoc, CExerciseView, CAccountDlg, CAccountDlg1 (CExerciseDoc selected)

Function handler name: OnFileNew

Handler description: Called after menu item or command button has been choosen

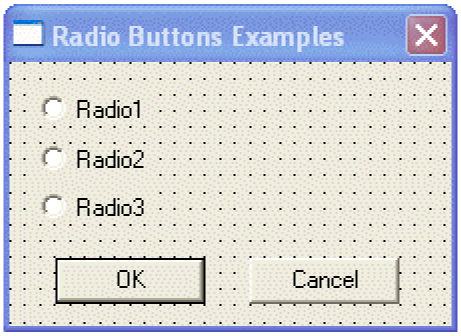
10. Implement the event as follows:

```
#include "stdafx.h"
#include "Clarksville Ice Scream1.h"

#include "ExerciseDoc.h"
#include "IceScreamOrderDlg.h"

. . .

void CExerciseDoc::OnFileNew()
{
	// TODO: Add your command handler code here
	CIceScreamOrderDlg Dlg;

	Dlg.DoModal();
}
```

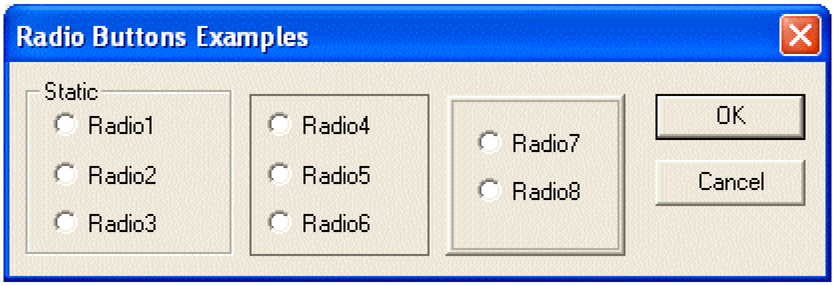
11. Test the application and return to MSVC

19.1.2 Creating Radio Buttons

Because they come in a group, to create radio buttons, on the Controls toolbox, click the Radio Button button ◉ and click the desired area on the host window. In the same way, add one or more radio buttons:



To indicate that the radio buttons belong to a group, you should (with emphasis) place them inside of a frame, the most common of which is the Group Box control. Alternatively, you can add them to a Picture frame (other allowed but unlikely frames are a Static control or an Animation box):



The main advantage of a Group Box control, which explains why it is the most common container of radio buttons, is that it inherently provides a caption you can use to indicate what its set of radio buttons is used for.

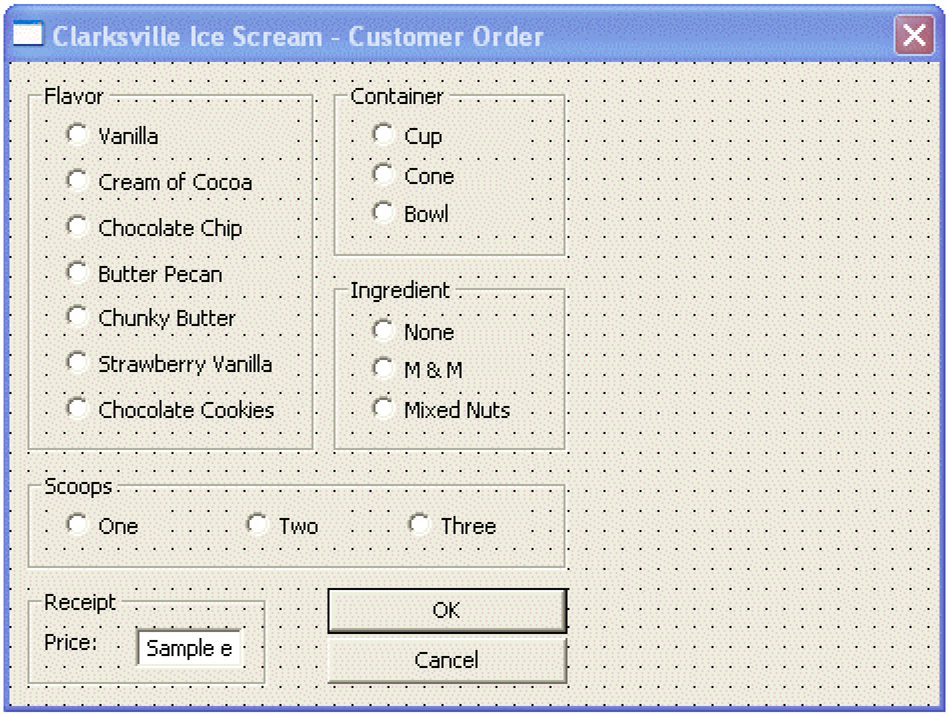
Two properties are of particular importance to both you and the user: the label and the state of the control. The label is text that specifies what a particular radio button is used for. The label should be explicit enough for the user to figure out the role of an item in the group.

To change the label of a radio button, after adding the control, replace the value of the *Caption* field in the Properties window. If the control exists already, to change its caption, call the **CWnd::SetWindowText()** method.

Pratical Learning: Creating Radio Buttons

1. From the Controls toolbox, click the Group Box button □ and click on the upper left section of the dialog box.
2. Change its **Caption** to **Flavor**
3. From the Controls toolbox, click the Radio Button ◉ and click inside the Flavor group box

4. Change its **Caption** to **&Vanilla** and change its ID to **IDC_FLAVORS**
5. Complete the design of the dialog box as follows (when adding the radio buttons, start with the Vanilla and add those under it. Then add the Container group box and add its radio buttons. Then add the Ingredient group box and its radio buttons. Finally, add the Scoops group box followed by its radio buttons. If you do not, neglect to, or forget to, follow this order, when you have finished, display the Tab Order and reorder th sequence. This is because we will keep this sequence in mind when creating the groups in the next section):

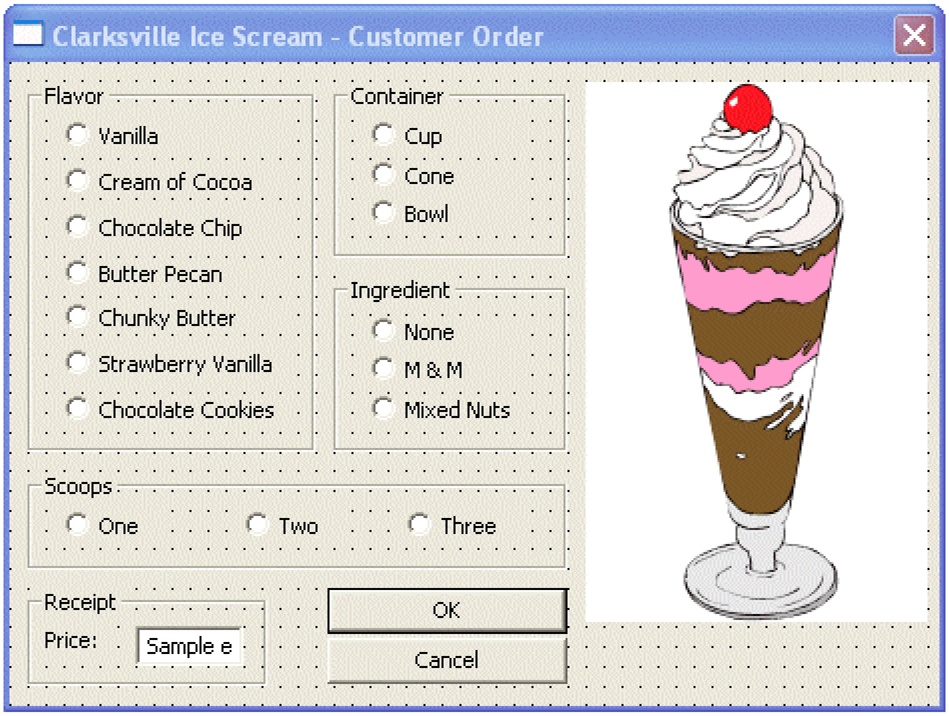


Except for the last Edit control, all controls on this table are radio buttons

ID	Caption	ID	Caption
IDC_FLAVORS	&Vanilla	IDC_CONTAINER	Cu&p
IDC_RDO_CREAMOFCOCOA	&Cream of Cocoa	IDC_RDO_CONE	C&one
IDC_RDO_CHOCOLATE	C&hocolate Chip	IDC_RDO_BOWL	Bo&wl
IDC_RDO_BUTTERPECAN	&Butter Pecan	IDC_INGREDIENT	&None
IDC_RDO_CHUNKYBUTTER	Ch&unky Butter	IDC_RDO_MM	&M && M
IDC_RDO_STRAWVAN	&Strawberry Vanilla	IDC_RDO_MIXEDNUTS	Mi&xed Nuts
IDC_RDO_CHOCOCOOKIES	Chocol&ate Cookies		
IDC_SCOOPS	On&e		
IDC_SCOOPS_ TWO	&Two		
IDC_SCOOPS_ THREE	Th&ree	IDC_PRICE	

6. From the resources that accompany this book, import the cup.bmp picture. If you are using MSVC 6, you will receive a message box. Simply click OK
Change its ID to IDB_CUP
7. In the same way, import the cone.bmp picture and change its ID to IDB_CONE
8. Import the bowl.bmp picture and change its ID to IDB_BOWL
9. Add a Picture control to the right side of the dialog box and change its ID to IDC_PICTURE

10. Set its Type to Bitmap and, in its Image combo box, select IDB_CUP
11. Set its **Center Image** property to True

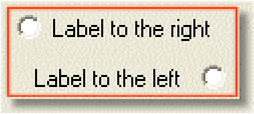


12. Add a Control variable for the Picture control and name it **m_Picture**
13. Save All

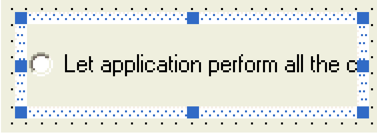
19.1.3 Radio Button Properties

As described already, to select a radio button, the user clicks it. To indicate to the user that a button has been selected, it should display a dot in its round box. To display this dot automatically, set the **Auto** property to True.

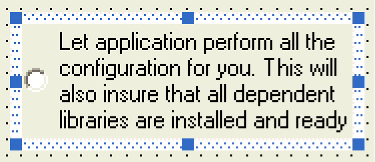
When adding a radio button to a host, it displays a text label to the right side of its round box. If you prefer to position the label to the left of the small circle, set the **Left Text** property to True or, at run time, add the **BS_LEFTTEXT** style:



The label that accompanies a radio button is static text that is included in a rectangular shape. Since the radio button by default is configured to display one line of text, the round box is locate to the left side of the label. If you plan to use multiple lines of text for the label, at design time, change the rectangular area of the control accordingly:

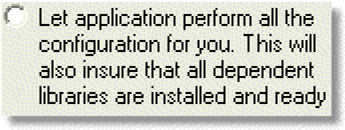
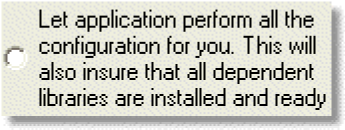
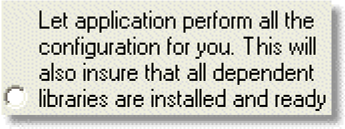
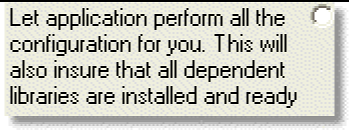
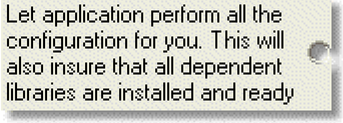
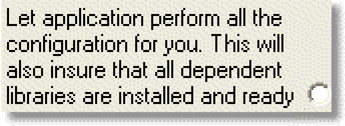


If you are programmatically creating the control, specify an appropriate *rect* value to the **Create()** method. Then, set the Multiline property to True or, at run time, add the **BS_MULTILINE** style:



Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready

Once the text can fit in the allocated area, you can accept the default alignment of the label or change it as you see fit. The alignment of text is specified using the Horizontal Alignment and the Vertical Alignment properties. Here are the possible combinations:

Left Text: False Horz Align: Default or Left Vert Align: Top	 Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready
Left Text: False Horz Align: Default or Left Vert Align: Default or Center	 Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready
Left Text: False Horz Align: Default or Left Vert Align: Bottom	 Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready
Left Text: True Horz Align: Default or Left Vert Align: Top	 Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready
Left Text: True Horz Align: Default or Left Vert Align: Default or Center	 Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready
Left Text: True Horz Align: Default or Left Vert Align: Bottom	 Let application perform all the configuration for you. This will also insure that all dependent libraries are installed and ready

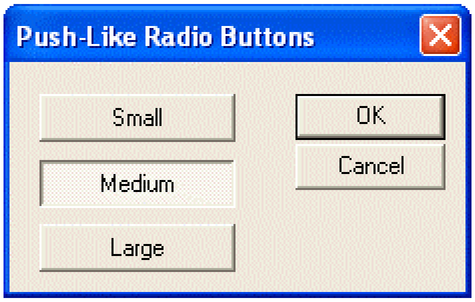
In the same way, you can add other radio buttons and individually configure each. For harmony, all radio of a group should have the same design.

For either the user or the programmer, radio buttons must behave as entities of one group. When you freshly add radio buttons to a host, they are created as a group. If your form or dialog box will be made of only a few radio buttons, you may not need to do much. If you plan to use more than one set of radio buttons, then you must separate them into groups. To do that, set the Group property of each new button of a group to True.

Imagine you will use two sets of group buttons on a dialog box. Add the first radio button and set its **Group** property to True (checked) or add the **WS_GROUP** value to its style; then, add the other radio buttons of the set but set their **Group** property to False (unchecked) or do not add the **WS_GROUP** value to their styles.

When starting the new set, add a new radio button and set its **Group** property to True (checked). Add the other radio buttons with the **Group** property set to False (unchecked) or without the **WS_GROUP** style.

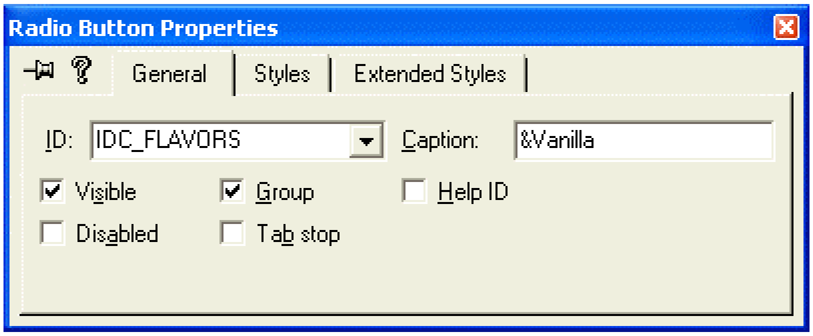
Once the radio buttons belong to the same group, if the user clicks one that is empty ? , it gets filled with a big dot ? and all the others become empty ? . This is the default and most common appearance these controls can assume. Alternatively, you can give them the appearance of a regular command button with 3-D borders. To do this, set the **Push-Like** property to True. When the radio buttons appear as command buttons, the control that is selected appear pushed or down while the others are up:



If you do not want the default 3-D design of radio buttons, you can make them flat by setting the **Flat** property to True.

Practical Learning: Configuring Radio Buttons

1. On the dialog box, click the Vanilla radio button and, on the Properties window, check the Group check box or set its value to True:

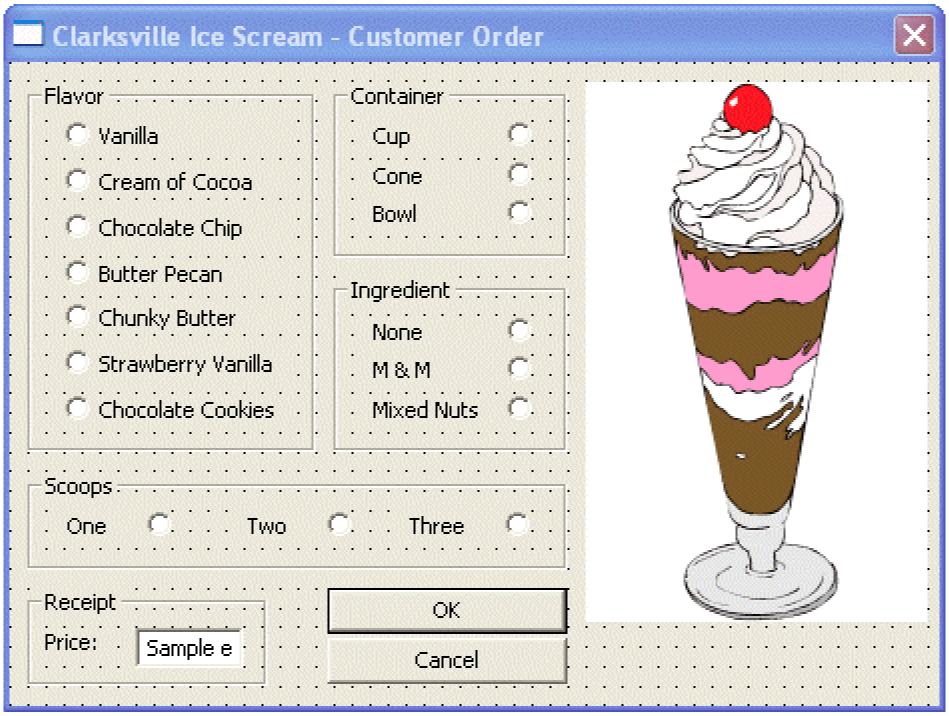


ID: IDC_FLAVORS Caption: &Vanilla

☑ Visible ☑ Group □ Help ID

□ Disabled □ Tab stop

2. In the same way check the **Group** property of the Cup, None, and One radio buttons or set this property to True for them
3. Set the Group box of all the other radio buttons to unchecked or False
4. Except for the radio buttons in the Flavor group, check the Left Text property of all the other radio buttons and set their **Left Text** value to True



5. Execute the application to test the radio buttons
6. Close the dialog box and return to MSVC

19.1.4 Radio Buttons Methods

As a Windows control, the radio button is based on the CButton class which, like all other constrols, is based on CWnd. As far as the Win32 and the MFC libraries are concerned, a radio button is first of all, a button. Therefore, to programmatically create a radio button, declare a variable or a pointer to CButton using its default constructor. Like all other MFC controls, the CButton class provides a Create() method to initialize it. The syntax of this method for a button is:

BOOL Create(LPCTSTR *lpszCaption*, DWORD *dwStyle*, const RECT& *rect*, CWnd* *pParentWnd*, UINT *nID*);

The *lpszCaption* argument is the string of the label that accomponies the radio button. The differences of radio buttons are set using the *dwStyle* argument. A radio button must have the **BS_RADIOBUTTON** value.

When setting the *rect* argument, provide enough space for the control because, by default, it will be confined to that rectangle and cannot display beyond that. Here are two examples:

```
BOOL CDialog12Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon
	// TODO: Add extra initialization here
	CButton *btnSmall = new CButton;
	CButton *btnMedium = new CButton;

	btnSmall->Create("&Small",
	                  WS_CHILD | WS_VISIBLE | BS_RADIOBUTTON,
	                 CRect(130, 40, 200, 50), this, 0x12);
	btnMedium->Create("&Medium",
	                    WS_CHILD | WS_VISIBLE | BS_RADIOBUTTON,
	                    CRect(130, 70, 200, 80), this, 0x14);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

If you want the radio button to get filled with a dot when it has been selected and to remove the dot from the other radio buttons of the same group, create it with the **BS_AUTORADIOBUTTON** style.

While using the radio buttons, the user is presented with the controls and requested to select one. To make a selection, the user must click the desired choice. The control selected fills its circle with a (big) dot ? . To programmatically change the state of a radio button, you have various options.

The easiest way to select a radio button consists of calling the **CWnd::CheckRadioButton()** method (actually, you should call this method as **CDialog::CheckRadioButton()**). Its syntax is:

void CheckRadioButton(int *nIDFirstButton*, int *nIDLastButton*, int *nIDCheckButton*);

To use this method, you must know the identifiers of the radio buttons involved. The *nIDFirstButton* is the identifier of the first radio button of the group. The *nIDLastButton* is the identifier of the last radio button of the group. The last argument, *nIDCheckButton* is the identifier of the radio button that must be selected. Here is an example:

```
void CDialog12Dlg::OnBtnSelect2()
{
	// TODO: Add your control notification handler code here
	CheckRadioButton(IDC_RADIO1, IDC_RADIO3, IDC_RADIO3);
}
```

To change the state of a radio button, you can call the **CButton::SetCheck()** method. Its syntax is:

void SetCheck(int nCheck);

The *nCheck* value indicates how to fill the control's circle. To fill it with the selection dot, pass the value of 0. To deselect a radio button, pass the argument as 1. Here is an example:

```
void CDialog12Dlg::OnBtnSelect2()
{
	// TODO: Add your control notification handler code here
	CButton *btnSecond;

	btnSecond = reinterpret_cast<CButton *>(GetDlgItem(IDC_RADIO2));
	btnSecond->SetCheck(1);
}
```

To find out what radio button of a group is selected, you can call the **CWnd::GetCheckedRadioButton().** Its syntax is:

```
int GetCheckedRadioButton( int nIDFirstButton, int nIDLastButton );
```

When calling this method, you must specify the identifier of the first radio button of the group as *nIDFirstButton* and the identifier of the last radio button of the group as *nIDLastButton*. The method returns the identifier of the radio button whose circle is filled.

Alternatively, to find the selected radio button, call the **CButton::GetCheck()** method. Its syntax is:

```
int GetCheck( ) const;
```

This method returns 0 if the radio button is not selected. Otherwise, it returns 1 if the radio button is selected.

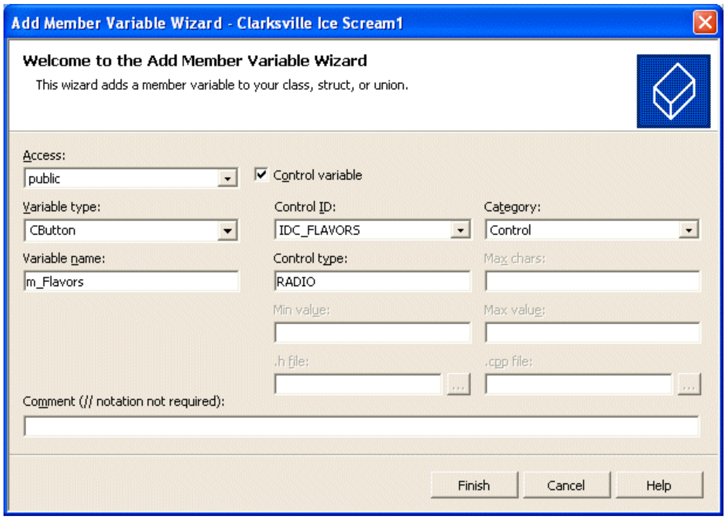
If you want to want to find whether a particular radio button is currently selected, you can call the **CWnd::IsDlgButtonChecked()** method. Its syntax:

```
UINT IsDlgButtonChecked(int nIDButton) const;
```

When calling this method, pass the identifier of the radio button as the nIDButton argument. This member function will check the state of the radio button. If the radio button is selected, the method will return 1. If the radio button is not selected, the method returns 0.

Practical Learning: Selecting Radio Buttons

1. Add a Control variable for the IDC_FLAVORS radio button and name it **m_Flavors**



Add Member Variable Wizard - Clarksville Ice Scream1

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: public

☑ Control variable

Variable type: CButton

Control ID: IDC_FLAVORS

Category: Control

Variable name: m_Flavors

Control type: RADIO

Max chars:

Min value:

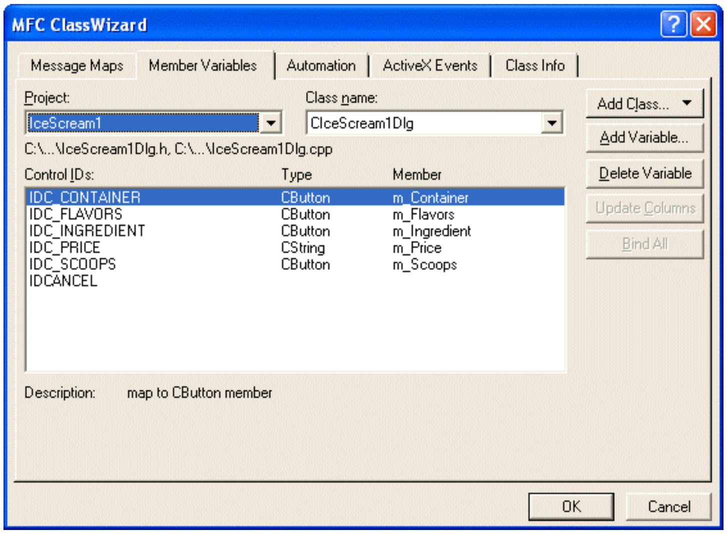
Max value:

.h file:

.cpp file:

Comment (// notation not required):

2. In the same way, add a Control variable for the IDC_CONTAINER, IDC_INGREDIENT, and IDC_SCOOPS named m_ Container, m_Ingredient, and m_Scoops respectively
3. Add a **CString** value variable for the IDC_PRICE edit control and name it **m_Price**



Project: IceScream1

Class name: CIceScream1Dlg

C:\...\IceScream1Dlg.h, C:\...\IceScream1Dlg.cpp

Control IDs:	Type	Member
IDC_CONTAINER	CButton	m_Container
IDC_FLAVORS	CButton	m_Flavors
IDC_INGREDIENT	CButton	m_Ingredient
IDC_PRICE	CString	m_Price
IDC_SCOOPS	CButton	m_Scoops
IDCANCEL		

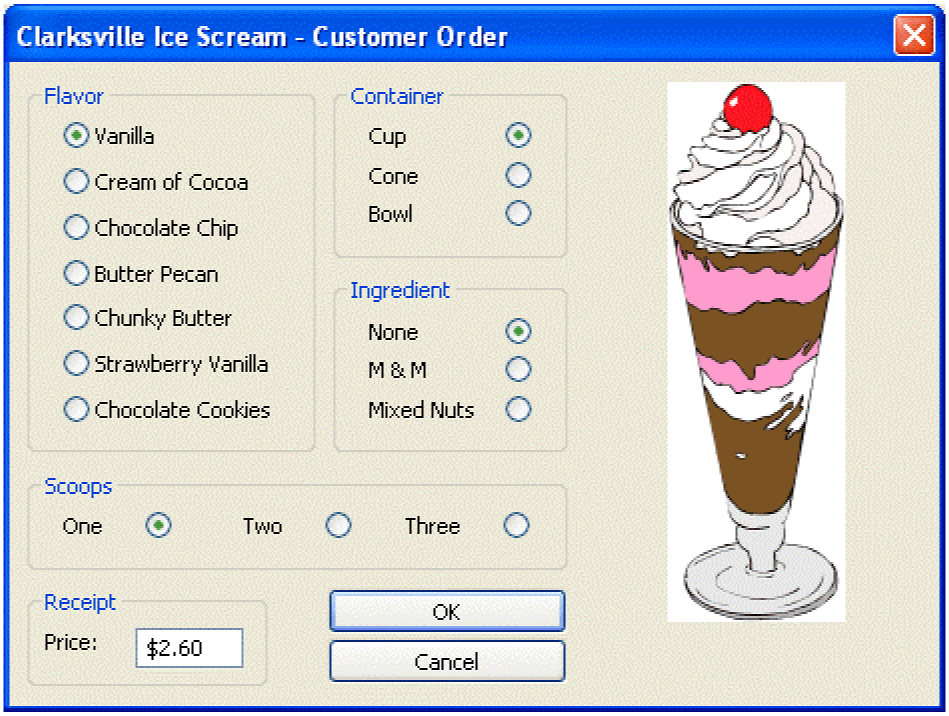
Description: map to CButton member

4. When the clerk is about to process a new ice scream order, some radio buttons should be selected already, those that would be the default.
   To select the default radio buttons, display the **OnInitDialog()** event of the dialog box and change it as follows:

```
// IceScreamOrderDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Clarksville Ice Scream1.h"
#include "IceScreamOrderDlg.h"

// CIceScreamOrderDlg dialog

IMPLEMENT_DYNAMIC(CIceScreamOrderDlg, CDialog)
CIceScreamOrderDlg::CIceScreamOrderDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CIceScreamOrderDlg::IDD, pParent)
	, m_Price(_T("$2.60"))
{
}

CIceScreamOrderDlg::~CIceScreamOrderDlg()
{
}

void CIceScreamOrderDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	DDX_Control(pDX, IDC_FLAVORS, m_Flavors);
	DDX_Control(pDX, IDC_RDO_CUP, m_Container);
	DDX_Control(pDX, IDC_IGR_NONE, m_Ingredient);
	DDX_Control(pDX, IDC_SCOOPS_ONE, m_Scoops);
	DDX_Text(pDX, IDC_PRICE, m_Price);
}

BEGIN_MESSAGE_MAP(CIceScreamOrderDlg, CDialog)
END_MESSAGE_MAP()


// CIceScreamOrderDlg message handlers
```

```
BOOL CIceScreamOrderDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO:  Add extra initialization here
	m_Flavors.SetCheck(1);
	m_Container.SetCheck(1);
	m_Ingredient.SetCheck(1);
	m_Scoops.SetCheck(1);

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

5. Test the application



Flavor
● Vanilla
○ Cream of Cocoa
○ Chocolate Chip
○ Butter Pecan
○ Chunky Butter
○ Strawberry Vanilla
○ Chocolate Cookies

Container
Cup ●
Cone ○
Bowl ○

Ingredient
None ●
M & M ○
Mixed Nuts ○

Scoops
One ● Two ○ Three ○

Receipt
Price: $2.60

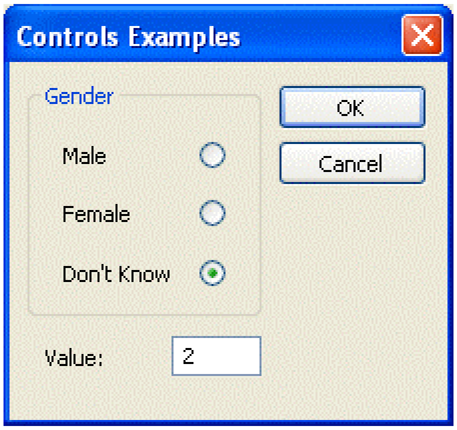
6. Return to MSVC

19.1.5 Radio Buttons Events

Because a radio is first of all a control of **CButton** type, when it is clicked, it generates the **BN_CLICKED** message. You can use this message to take action in response to the user's.

We have mentioned already that a control can have two types of variables: Control or Value. For a radio button, you can declare or add either a CButton or a regular C++ variable. If you decide to use a **CButton** variable as done in the previous exercise, you can call the necessary method to take advantage of the class. On the other hand, if you want to use a value instead. You can add a variable for only the first radio button of the group, the radio button that has the Group property set to true. The variable should be an integer, such as int. In such a case, each radio button of the group can be referred to using a number. The first radio button of the group would have the index 0, the second would be 1, etc.

Here is an example:



Value: 2

```
#pragma once

// CControlsDlg dialog

class CControlsDlg : public CDialog
{
	DECLARE_DYNAMIC(CControlsDlg)

public:
	CControlsDlg(CWnd* pParent = NULL);   // standard constructor
	virtual ~CControlsDlg();

// Dialog Data
	enum { IDD = IDD_DLG_CONTROLS };

protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

	DECLARE_MESSAGE_MAP()
public:
	CString m_Value;
	int m_Gender;
	void UpdateChoice();
	afx_msg void OnBnClickedRdoMale();
	afx_msg void OnBnClickedRdoFemale();
	afx_msg void OnBnClickedRdoDontknow();
};
```

```
// ControlsDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Dialog1.h"
#include "ControlsDlg.h"

// CControlsDlg dialog

IMPLEMENT_DYNAMIC(CControlsDlg, CDialog)
```

```
CControlsDlg::CControlsDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CControlsDlg::IDD, pParent)
        , m_Value(_T(""))
        , m_Gender(0)
{
}

CControlsDlg::~CControlsDlg()
{
}

void CControlsDlg::DoDataExchange(CDataExchange* pDX)
{
        CDialog::DoDataExchange(pDX);
        DDX_Text(pDX, IDC_VALUE, m_Value);
        DDX_Radio(pDX, IDC_RDO_MALE, m_Gender);
}

BEGIN_MESSAGE_MAP(CControlsDlg, CDialog)
        ON_BN_CLICKED(IDC_RDO_MALE, OnBnClickedRdoMale)
        ON_BN_CLICKED(IDC_RDO_FEMALE, OnBnClickedRdoFemale)
        ON_BN_CLICKED(IDC_RDO_DONTKNOW, OnBnClickedRdoDontknow)
END_MESSAGE_MAP()

// CControlsDlg message handlers
void CControlsDlg::UpdateChoice()
{
        UpdateData();
        int Value;

        switch(m_Gender)
        {
        case 0:
                Value = 0;
                break;
        case 1:
                Value = 1;
                break;
        case 2:
                Value = 2;
                break;
        }

        m_Value.Format("%d", Value);

        UpdateData(FALSE);
}

void CControlsDlg::OnBnClickedRdoMale()
{
        // TODO: Add your control notification handler code here
        UpdateChoice();
}

void CControlsDlg::OnBnClickedRdoFemale()
{
        // TODO: Add your control notification handler code here
        UpdateChoice();
}
```

```
void CControlsDlg::OnBnClickedRdoDontknow()
{
	// TODO: Add your control notification handler code here
	UpdateChoice();
}
```

Practical Learning: Implementing Radio Buttons

1. When the clerk clicks a Container radio button, we will change the right picture based on the selected container.
   In the header file of the Customer Order dialog box class, declare a member variable named **ContSelected**and of type **int**

```
private:
	int Selected;
};
```

2. Generate the WM_PAINT message of the dialog box and implement it as follows:

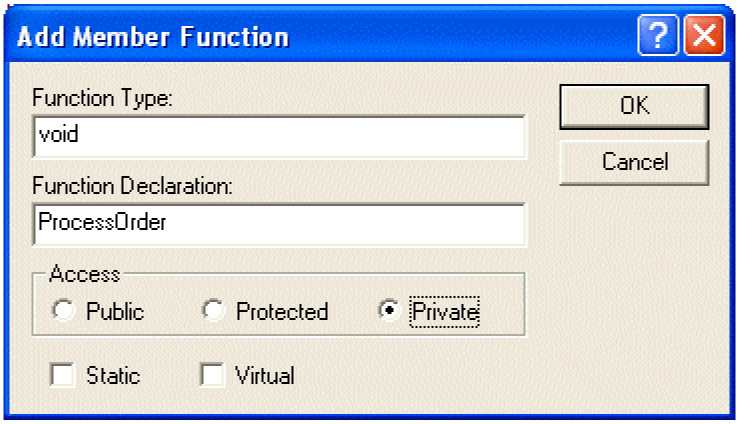
```
void CIceScreamOrderDlg::OnPaint()
{
	CPaintDC dc(this); // device context for painting

	// TODO: Add your message handler code here
	// Do not call CDialog::OnPaint() for painting messages
	CBitmap Bmp;

	switch(Selected)
	{
	case 0:
		Bmp.LoadBitmap(IDB_CUP);
		m_Picture.SetBitmap(Bmp);
		break;
	case 1:
		Bmp.LoadBitmap(IDB_CONE);
		m_Picture.SetBitmap(Bmp);
		break;
	case 2:
		Bmp.LoadBitmap(IDB_BOWL);
		m_Picture.SetBitmap(Bmp);
		break;
	}
}
```

3. On the dialog box, right-click the Cup radio button
   If you are using MSVC 6, click Events
   If you are using MSVC 7, click Add Event Handler
4. Select the Message Type as BN_CLICKED. Accept the name of the event as OnBnClickedContainer
5. If you are using MSVC 6, click Add Handler, accept the name and click Edit Existing
   If using MSVC 7, click Edit Code
6. In the same way, add a **BN_CLICKED** event for the Cone and the Bowl radio buttons
7. Implement the events as follows:

```
void CIceScreamOrderDlg::OnBnClickedContainer()
```

```
{
	// TODO: Add your control notification handler code here
	// The first radio button was selected
	Selected = 0;

	// Get the rectangle that holds the picture
	CRect PictArea;
	m_Picture.GetWindowRect(&PictArea);
	// Repaint the picture
	InvalidateRect(&PictArea);
}

void CIceScreamOrderDlg::OnBnClickedRdoCone()
{
	// TODO: Add your control notification handler code here
	// The second radio button was selected
	Selected = 1;

	// Get the rectangle that holds the picture
	CRect PictArea;
	m_Picture.GetWindowRect(&PictArea);
	// Repaint the picture
	InvalidateRect(&PictArea);
}

void CIceScreamOrderDlg::OnBnClickedRdoBowl()
{
	// TODO: Add your control notification handler code here
	// The third radio button was selected
	Selected = 2;

	// Get the rectangle that holds the picture
	CRect PictArea;
	m_Picture.GetWindowRect(&PictArea);
	// Repaint t he picture
	InvalidateRect(&PictArea);
}
```

8. Execute the application and test the Container radio buttons
9. To process an order, add a new Member Function to the **CIceScreamOrderDlg** class. Set its type as **void** and its name as **ProcessOrder**



Function Type: void

Function Declaration: ProcessOrder

Access: ○ Public ○ Protected ● Private

□ Static □ Virtual

10. Implement the method as follows:

```
// This is the central function that the other controls will refer to
```

```
void CIceScreamOrderDlg::ProcessOrder(void)
{
      double PriceFlavor, PriceContainer, PriceIngredient;
    double TotalPrice;

      int ContainerSelected = GetCheckedRadioButton(IDC_CONTAINER, IDC_RDO_BOWL);

      if( ContainerSelected == IDC_CONTAINER )
       PriceContainer = 0.35;
    else if( ContainerSelected == IDC_RDO_CONE )
       PriceContainer = 0.55;
    else if( ContainerSelected == IDC_RDO_BOWL )
       PriceContainer = 0.75;

      int IngredientSelected = GetCheckedRadioButton(IDC_INGREDIENT, IDC_RDO_MIXEDNUTS);

      if( IngredientSelected == IDC_INGREDIENT )
       PriceIngredient = 0.00;
      else if( IngredientSelected == IDC_RDO_MM )
       PriceIngredient = 0.55;
    else if( IngredientSelected == IDC_RDO_MIXEDNUTS )
       PriceIngredient = 0.75;

      int NumberOfScoops = GetCheckedRadioButton(IDC_SCOOPS, IDC_SCOOPS_THREE);
    if( NumberOfScoops == IDC_SCOOPS )
                PriceFlavor  = 2.25;
      else if( NumberOfScoops == IDC_SCOOPS_TWO )
                PriceFlavor  = 3.50;
      else if( NumberOfScoops == IDC_SCOOPS_THREE )
                PriceFlavor  = 4.25;

    TotalPrice = PriceContainer + PriceFlavor + PriceIngredient;

    m_Price.Format("$%.2f", TotalPrice);
      UpdateData(FALSE);
}
```

11. To make sure the new member function is called whenever the user clicks a radio button, double-click each radio button on the dialog and accept the suggested name for the eventWhen a customer is passing an order and request his or her ice scream on a cone or a cup, because of the size of the cone or the cup, it cannot contain three scoops. Therefore, if the clerk selects the Cone or the Cup radio buttons upon the customer’s request, we can disable the Three radio button. While doing this, we must check whether the user had previously selected the Three radio button. If that button was selected, we should deselect it and select the One radio, giving the clerk the time to select the One or Two radio button.
    To implement this behavior, change the above events as follows:

```
void CIceScreamOrderDlg::OnBnClickedContainer()
{
      // TODO: Add your control notification handler code here
      // The first radio button was selected
      Selected = 0;

      // Get the rectangle that holds the picture
      CRect PictArea;
      m_Picture.GetWindowRect(&PictArea);
      // Repaint the picture
```

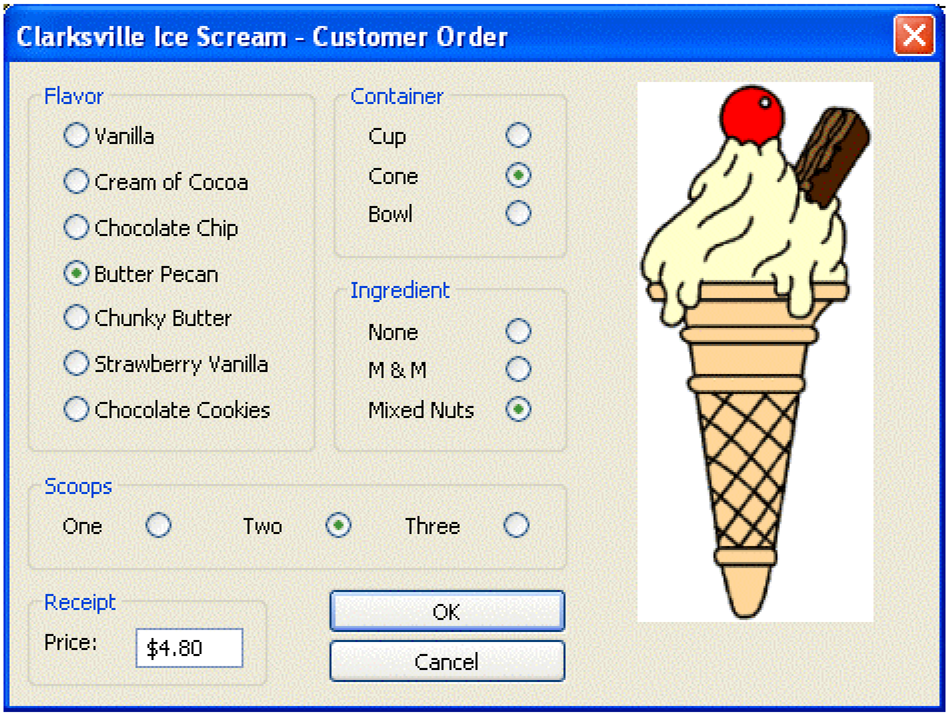
```
	InvalidateRect(&PictArea);

	CButton *rdoThree;

	rdoThree = reinterpret_cast<CButton *>(GetDlgItem(IDC_SCOOPS_THREE));
	rdoThree->EnableWindow(FALSE);

	// Check whether it is checked
	int CheckedButton = rdoThree->GetCheck();

	// If it is checked
	if( CheckedButton == 1 ) // then select the One radio button
		CheckRadioButton(IDC_SCOOPS, IDC_SCOOPS_THREE, IDC_SCOOPS);

	ProcessOrder();
}

void CIceScreamOrderDlg::OnBnClickedRdoCone()
{
	// TODO: Add your control notification handler code here
	// The second radio button was selected
	Selected = 1;
	// Let the UpdatePicture() method change the picture

	CRect PictArea;
	m_Picture.GetWindowRect(&PictArea);
	InvalidateRect(&PictArea);

	CButton *rdoThree;

	rdoThree = reinterpret_cast<CButton *>(GetDlgItem(IDC_SCOOPS_THREE));
	rdoThree->EnableWindow(FALSE);


	// Check whether it is checked
	int CheckedButton = rdoThree->GetCheck();

	// If it is checked
	if( CheckedButton == 1 ) // then select the One radio button
		CheckRadioButton(IDC_SCOOPS, IDC_SCOOPS_THREE, IDC_SCOOPS);

	ProcessOrder();
}

void CIceScreamOrderDlg::OnBnClickedRdoBowl()
{
	// TODO: Add your control notification handler code here
	// The third radio button was selected
	Selected = 2;

	CRect PictArea;
	m_Picture.GetWindowRect(&PictArea);
	InvalidateRect(&PictArea);

	CButton *rdoThree;

	// Get a handle to the Three radio button
	rdoThree = reinterpret_cast<CButton *>(GetDlgItem(IDC_SCOOPS_THREE));
```

```
        // Disable the Three radio button
        rdoThree->EnableWindow(TRUE);

        ProcessOrder();
}
```

12. Call the above member function in EACH event. Here is one example (to save printing paper):

```
void CIceScreamOrderDlg::OnBnClickedFlavors()
{
        // TODO: Add your control notification handler code here
        ProcessOrder();
}
```

13. Test the application



Flavor: ○ Vanilla ○ Cream of Cocoa ○ Chocolate Chip ● Butter Pecan ○ Chunky Butter ○ Strawberry Vanilla ○ Chocolate Cookies

Container: Cup ○ Cone ● Bowl ○

Ingredient: None ○ M & M ○ Mixed Nuts ●

Scoops: One ○ Two ● Three ○

Price: $4.80

14. Return to MSVC

19.2 Check Boxes

19.2.1 Introduction

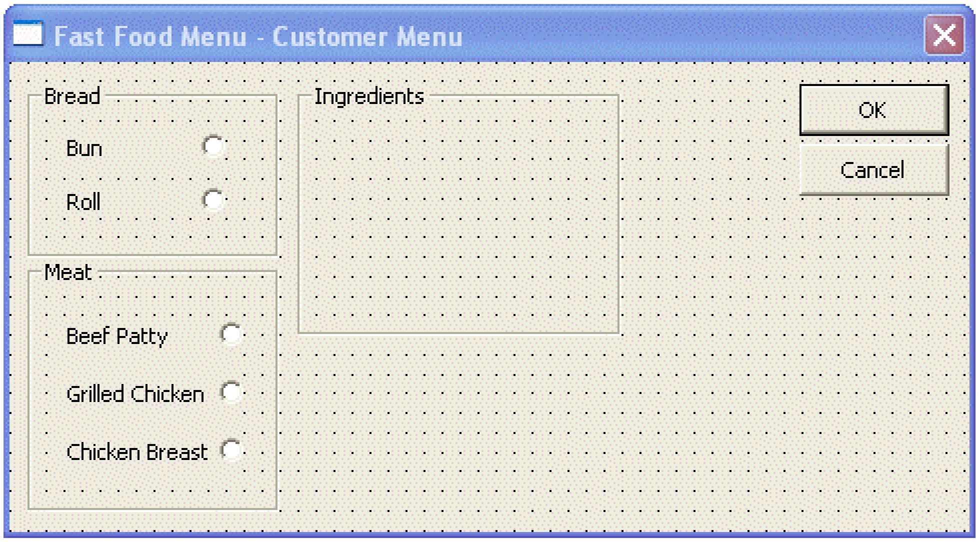
A check box is a Windows control that allows the user to set or change the value of an item as true or false. The control appears as a small square ☞. When this empty square is clicked, it gets marked by a check symbol ☞. These two states control the check box as checked ☞ or unchecked ☞.

Like the radio button, the check box does not indicate what it is used for. Therefore, it is usually accompanied by a label that displays an explicit and useful string. The label can be positioned on either part of the square box, depending on the programmer who implemented it.

Unlike the radio buttons that implement a mutual-exclusive choice, a check box can appear by itself or in a group. When a check box appears in a group with other similar controls, it assumes a completely independent behavior and it must be implemented as its own entity. This means that clicking a check box has no influence on the other controls.

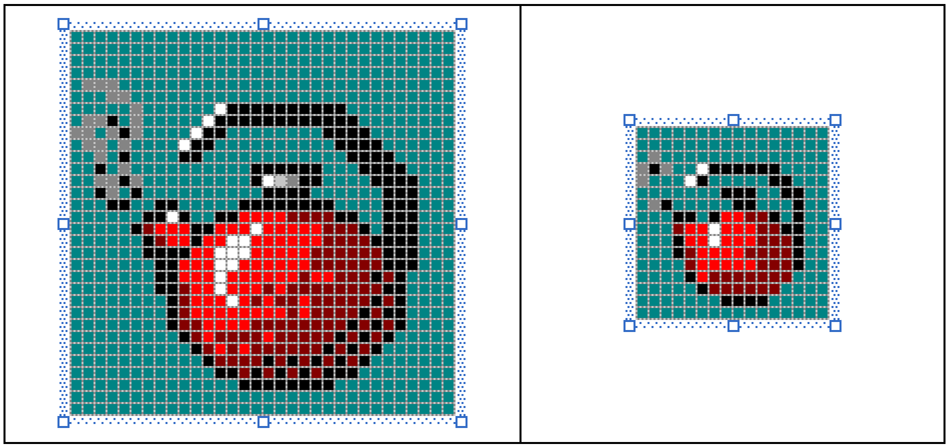
Practical Learning: Introducing Check Boxes

1. Start a new Dialog Based application named **FastFood** and set the Dialog Title as **Fast Food Restaurant – Customer Menu**
2. Delete the TODO line
3. Design the dialog box as follows:



Control	ID	Caption	Other Property
Group Box	*Don't Care*	Bread	
Radio Button	IDC_BREAD	&Bun	Group = True Left Text = True
Radio Button	IDC_ROLL	&Roll	Left Text = True
Group Box	*Don't Care*	Meat	
Radio Button	IDC_MEAT	Bee&f Patty	Group = True Left Text = True
Radio Button	IDC_GRILLED_CHICKEN	&Grilled Chicken	Left Text = True
Radio Button	IDC_CHICKEN_BREAST	&Chicken Breast	Left Text = True
Group Box	*Don't Care*	Ingredients	

4. Change the designs of the IDR_MAINFRAME icon as follows:



5. Save All

19.2.2 Check Box Properties

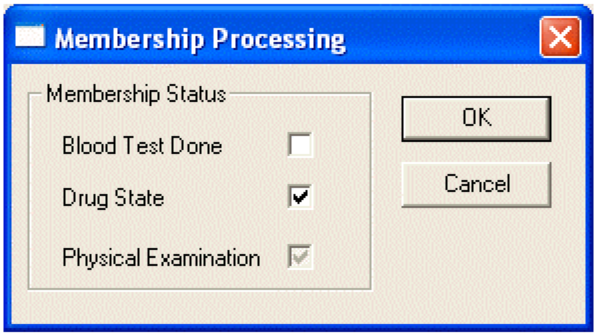
To provide a check box to an application, on the Controls toolbox, click the **Check Box** button ☒ and click the desired area on the dialog box or form. If you require more than one check box, you can keep adding them as you judge necessary.

Like the radio button, a check box is a button with just a different style. Therefore, it shares the same characteristics as the radio button. This means that we can simply avoid repeating the descriptions we reviewed for the radio control.

Like the radio button, the accompanying label of the check box can appear on the left or the right side of the square box. Also, you can control the horizontal and the vertical alignments of its label using the **Horizontal Alignment** or the **Vertical Alignment** properties.

We saw that a radio button can appear as a regular command button with 3-D borders. This feature is also available for check boxes. When a check box with the **Push-Like** property is up and then clicked, it stays down until it is clicked again, regardless of the appearance of the other controls. Remember that a check box behaves independently of the other controls even if it appears in a group.

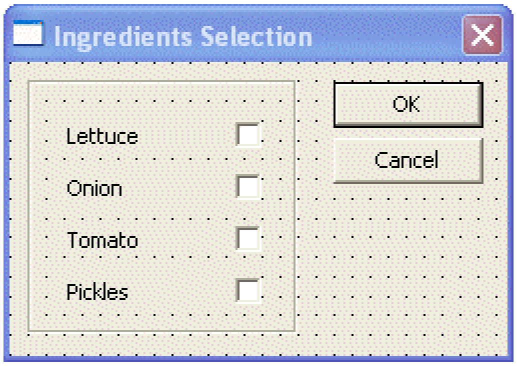
A check box can be checked or unchecked. The check box as a control adds a third state for its appearance. Instead of definitely appearing as checked or unchecked, a check box can appear “half-checked”. This is considered as a third state. To apply this behavior to a check box, set its **Tri-State** property to True or add the **BS_3STATE** style to it:



A check box that has the **Tri-State** property set or the **BS_3STATE** style, can appear checked, unchecked, or dimmed. When using this kind of check box, the user may have to click the control three times in order to get the necessary state.

Practical Learning: Creating Check Boxes

1. Add a new Dialog resource to the application
2. Change its ID to IDD_INGREDIENTS and its Caption to Ingredients Selection
   Here is how you will design it:

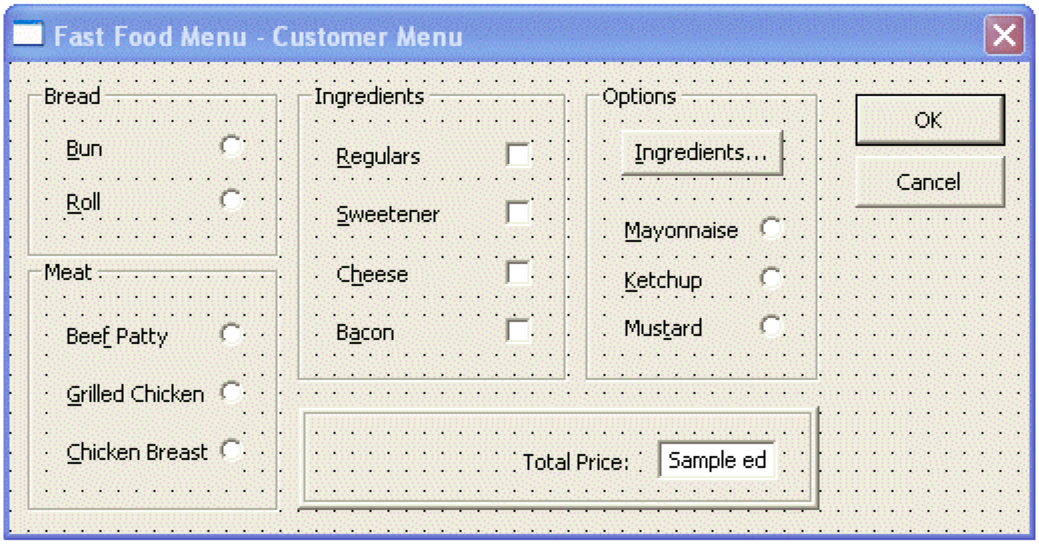


3. Add a Group Box control ▭ to the dialog box and delete its caption
4. On the Controls toolbox, click the Check Box button ☒ and click inside the newly added group box
5. Change its ID to **IDC_CHK_LETTUCE** and change its Caption to **&Lettuce**
6. Set its **Left Text** to True or checked
7. Add another check box under the first one. Set its ID to **IDC_CHK_ONION**, its Caption to **&Onion**, and its **Left Text** property to True
8. Add another check box under the first one. Set its ID to **IDC_CHK_TOMATO**, its Caption to **&Tomato**, and its **Left Text** property to True
9. Add another check box under the first one. Set its ID to **IDC_CHK_PICKLES**, its Caption to **&Pickle**, and its **Left Text** property to True
10. Add a class for the dialog box based on CDialog and named **CIngredientsDlg**
11. Add a Value variable for each check box as follows:

ID	Value Variable	ID	Value Variable
IDC_CHK_LETTUCE	m_bLettuce	IDC_CHK_ONION	m_bOnion

IDC_CHK_TOMATO	m_bTomato	IDC_CHK_PICKLES	m_bPickles

12. Display the main dialog box (Customer Menu)
13. In the top section inside the Ingredients group box, add a check box ☒
14. Set the ID of the new control to **IDC_CHK_REGULARS**
15. Change its **Caption** to **&Regulars**
16. Set its **Left Text** property to True or checked
17. Set its **Tri-State** property to True
18. Complete the design of the dialog box as follows:



Control	ID	Caption	Other Property
Group Box	*Ingredients*	Ingredients	
Check Box	IDC_CHK_REGULARS	&Regulars	Left Text = True
Check Box	IDC_CHK_SWEETENER	&Sweetener	Left Text = True
Check Box	IDC_CHK_CHEESE	C&heese	Left Text = True
Check Box	IDC_CHK_BACON	B&acon	Left Text = True
Group Box	*Don't Care*	Options	
Button	IDC_INGREDIENTS	&Ingredients	
Radio Button	IDC_OPTIONS	&Mayonnaise	Group = True Left Text = True
Radio Button	IDC_KETHUP	&Ketchup	Left Text = True
Radio Button	IDC_MUSTARD	Mus&tard	Left Text = True
Picture			Color = Etched Modal Frame = True
Static		Total Price:	
Edit Box	IDC_TOTAL_PRICE		Align Text = Right

19. Add a Control variable for all non-IDC_STATIC controls with the following names:

ID	Control Variable	ID	Control Variable
IDC_BREAD	m_Bread	IDC_MEAT	M_Meat
IDC_REGULARS	m_Regulars	IDC_SWEETENER	m_Sweetener
IDC_CHEESE	m_Cheese	IDC_BACON	m_Bacon

IDC_INGREDIENTS	m_Ingredients	IDC_OPTIONS	M_SweetOptions
IDC_TOTAL_PRICE	m_TotalPrice		

20. To create a basic sandwich based on what we know so far, in the **OnInitDialog()** event of the main dialog class, type the following:

```
BOOL CFastFoodDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	. . .

	// TODO: Add extra initialization here
	m_Bread.SetCheck(1);
	m_Meat.SetCheck(1);
	m_SweetOptions.SetCheck(1);
	m_TotalPrice.SetWindowText("$2.35");

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

21. Add a **BN_CLICKED** event for the Ingredients button and implement it as follows:

```
// FastFoodDlg.cpp : implementation file
//

#include "stdafx.h"
#include "FastFood1.h"
#include "FastFoodDlg.h"
#include "IngredientsDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

. . .

void CFastFoodDlg::OnBnClickedBtnIngredients()
{
	// TODO: Add your control notification handler code here
	CIngredientsDlg Dlg;

	Dlg.DoModal();
}
```

22. Test the application

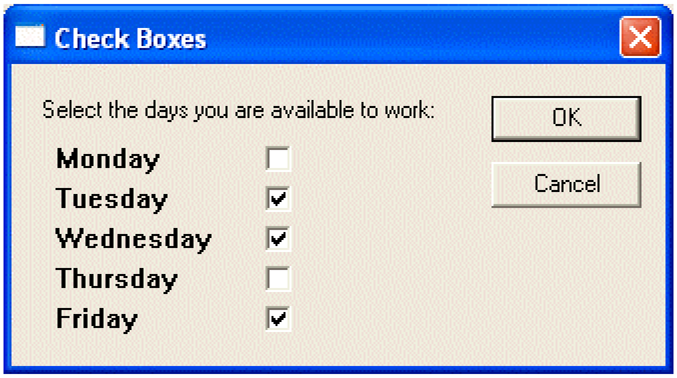
19.2.3 Check Box Methods

Like the radio button, a check box is based on the **CButton** class. Therefore, to programmatically create a check box, declare **CButton** variable or pointer using its constructor. Here is an example:

```
CButton *HotTempered = new CButton;
```

After declaring a variable or a pointer to **CButton**, you can call its **Create()** method to initialize the control. The syntax is the same as for the radio button. To make the button a check box, its style must include the **BS_CHECKBOX** style. If you want the control to display or hide a check mark when it gets clicked, create it with the **BS_AUTOCHECKBOX** style.

```
BOOL CCheckBoxes::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CButton *chkDay[5];

	chkDay[0] = new CButton;
	chkDay[0]->Create("&Monday", WS_CHILD | WS_VISIBLE |
	                  BS_AUTOCHECKBOX | BS_LEFTTEXT,
	                  CRect(20, 40, 140, 55), this, 0x11);
	chkDay[1] = new CButton;
	chkDay[1]->Create("&Tuesday", WS_CHILD | WS_VISIBLE |
	                  BS_AUTOCHECKBOX | BS_LEFTTEXT,
	                  CRect(20, 60, 140, 75), this, 0x12);
	chkDay[2] = new CButton;
	chkDay[2]->Create("&Wednesday", WS_CHILD | WS_VISIBLE |
	                  BS_AUTOCHECKBOX | BS_LEFTTEXT,
	                  CRect(20, 80, 140, 95), this, 0x13);
	chkDay[3] = new CButton;
	chkDay[3]->Create("&Thursday", WS_CHILD | WS_VISIBLE |
	                  BS_AUTOCHECKBOX | BS_LEFTTEXT,
	                  CRect(20, 100, 140, 115), this, 0x14);
	chkDay[4] = new CButton;
	chkDay[4]->Create("&Friday", WS_CHILD | WS_VISIBLE |
	                  BS_AUTOCHECKBOX | BS_LEFTTEXT,
	                  CRect(20, 120, 140, 135), this, 0x15);

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



We mentioned already that a check box can assume one of three states: checked, unchecked, or dimmed. If you want to apply one of the states to such a check box, that is, to programmatically check, uncheck, or dim it, you can call the **CWnd::CheckDlgButton()** method whose syntax is:

void CheckDlgButton(int *nIDButton*, UINT *nCheck*);

The *nIDButton* argument is the identifier of the check box whose state you want to modify. The *nCheck* argument is the value to apply. It can have one of the following values:

State Value	Description
BST_UN CHECKED	The check mark will be completely removed
BST_ CHECKED	The button will be checked
BST_INDETERMINATE	A dimmed checked mark will appear in the control's square. If the Tri-State property is not set or the BS_3STATE style is not applied to the control, this value will be ignored

Here is an example that automatically sets a check box to a dimmed appearance when the dialog box comes up:

```
BOOL CCheckBoxes::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        CheckDlgButton(IDC_CHECK_PHYSICAL, BST_INDETERMINATE);

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

Alternatively, to change the state of a check box, you can call the **CButton::SetCheck()** and pass of the above state values. For a check box, to set a dimmed check mark, here is an example:

```
BOOL CCheckBoxes::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here
    CButton *ChkPhysical = new CButton;

    ChkPhysical = reinterpret_cast<CButton *>(GetDlgItem(IDC_CHECK_PHYSICAL));
    ChkPhysical->SetCheck(BST_INDETERMINATE);

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

To get the current state of a check box, you can call the **CWnd::IsDlgButtonChecked()** method. If a check mark appears in the square box, this method will return 1. If the square box is empty, this method returns 0. If the check box control has the BS_3STATE style, the method can return 2 to indicate that the check mark that appears in the square box is dimmed.

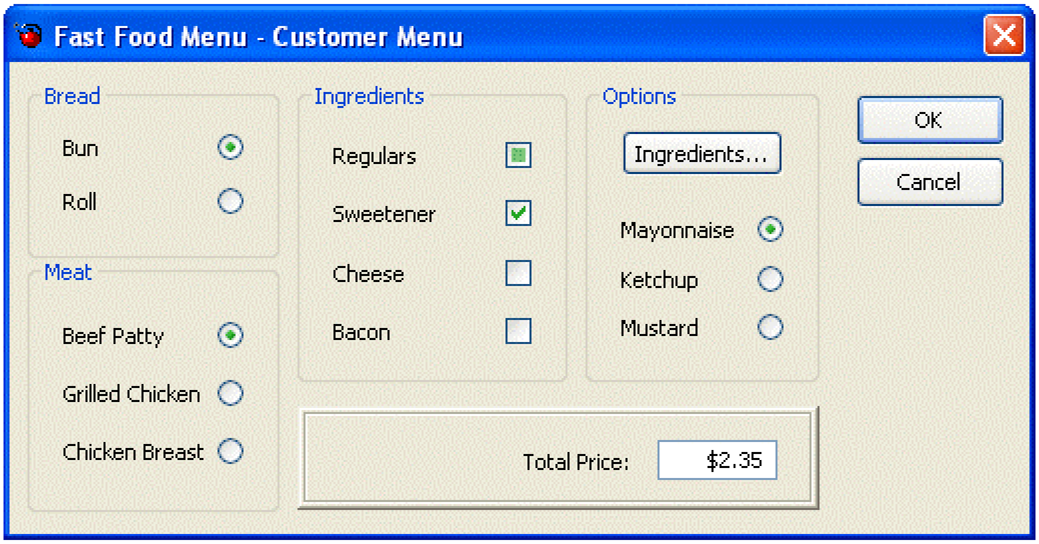
Practical Learning: Implementing Check Boxes

1. To show that at least one sweetener sauce is selected, in the **OnInitDialog()** event of the main (Customer Menu) dialog box, call the **CheckDlgButton()** method. Pass it the ID of the Sweetener control and the **BST_CHECKED** constant

2. To show that some ingredients are selected by default for a sandwich, call the **CheckDlgButton()** method. Pass it the ID of the Regulars control and the **BST_INDETERMINATE** constant:

```
BOOL CFastFoodDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	. . .

	// TODO: Add extra initialization here
	m_Bread.SetCheck(1);
	m_Meat.SetCheck(1);
	m_SweetOptions.SetCheck(1);
	m_TotalPrice.SetWindowText("$2.35");

	CheckDlgButton(IDC_CHK_SWEETENER, BST_CHECKED);
	CheckDlgButton(IDC_CHK_REGULARS, BST_INDETERMINATE);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

3. Test the application:



Fast Food Menu - Customer Menu

Bread: Bun ●, Roll ○

Meat: Beef Patty ●, Grilled Chicken ○, Chicken Breast ○

Ingredients: Regulars ▨, Sweetener ☑, Cheese □, Bacon □

Options: Mayonnaise ●, Ketchup ○, Mustard ○

Total Price: $2.35

4. Close the dialog and return to MSVC

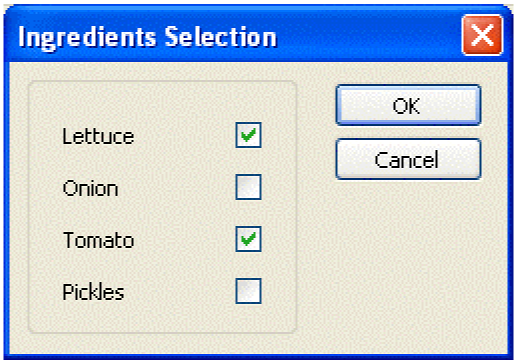
5. Add an OnInitDialog event for the Ingredients dialog box (If you are using MSVC 6, display the ClassWizard, select the CIngredientsDlg class and, in the Messages list, double-click WM_INITDIALOG. If you are using MSVC 7, in Class View, click CIngredientsDlg and, in the Properties window, click the Overrides button ; then click the arrow of the OnInitDialog combo box and select the only item in the combo box

6. Implement the event as follows:

```
BOOL CIngredientsDlg::OnInit Dialog()
```

```
{
	CDialog::OnInitDialog();

	// TODO:  Add extra initialization here
	CheckDlgButton(IDC_CHK_LETTUCE, BST_CHECKED);
	CheckDlgButton(IDC_CHK_TOMATO, BST_CHECKED);

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

7. Execute the program and click the Ingrdients button:



8. Return to MSVC

19.2.4 Check Box Events

Like all other MFC buttons, the check box can natively send only the click and the double-click messages. These messages respectively are **BN_CLICKED** and **BN_DOUBLECLICKED**, which originate when the user clicks or double-clicks the control. If you need to, you can use either the events of the parent window or hand code the messages yourself.

Practical Learning: Using Check Boxes Messages

1. If the user completely removes the check mark on the Sweetener control, this suggests that the customer does not want this item on the sandwich. Consequently, the radio buttons in the Options group should be disabled. When the user clicks a check box, whether the control was already checked or not, the **BN_CLICKED** message is sent. Therefore, in this case, the first thing you should do it to check the state of the check button and then implement a behavior accordingly.
Display the Customer Menu dialog box. Right-click the Sweetener control
2. If you are using MSVC 6, click Events
If you are using MSVC 7, click Add Event Handler
3. If you are using MSVC 6, double-click BN_CLICKED. Accept the suggested name. Click OK and click Add And Edit
If you are using MSVC 7, make sure the Message Type is set to BN_CLICKED and the Class List is set to CfastFoodDlg. Then click Add And Edit
4. Implement the event as follows:

```
void CFastFoodDlg::OnBnClickedChkSweetener()
{
    // TODO: Add your control notification handler code here
    // Get the state of the Sweetener check box
    int CurState = IsDlgButtonChecked(IDC_CHK_SWEETENER);

    // Get a handle to the radio buttons of the Options group
    CWnd *rdoKetchup, *rdoMustard;

    rdoKetchup = reinterpret_cast<CButton *>(GetDlgItem(IDC_KETCHUP));
    rdoMustard = reinterpret_cast<CButton *>(GetDlgItem(IDC_MUSTARD));

    // If the Sweetener check box is not checked
    if( CurState == 0 )
    {
        // Disable the Options radio buttons
        m_SweetOptions.EnableWindow(FALSE);
        rdoKetchup->EnableWindow(FALSE);
        rdoMustard->EnableWindow(FALSE);
    }
    else
    {
        // Otherwise, enable the Options radio buttons
        m_SweetOptions.EnableWindow(TRUE);
        rdoKetchup->EnableWindow(TRUE);
        rdoMustard->EnableWindow(TRUE);
    }
}
```

5. Now, we will give complete or more control to the user with regards to the ingredients. When the Regulars check box is not checked at all, no basic ingredient is selected, when this control is checked, all ingredients will be added to the sandwich. If at least one ingredient is selected and at least one ingredient is not selected, the Regulars check box should appear dimmed.
   To implement this, first access the **OnInitDialog()** event of the CIngredientsDlg class and delete the initializations of its check boxes:

```
BOOL CIngredientsDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO:  Add extra initialization here
//  CheckDlgButton(IDC_CHK_LETTUCE, BST_CHECKED);
//  CheckDlgButton(IDC_CHK_TOMATO, BST_CHECKED);

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

6. When the user clicks the Ingredients button, we will display the Ingredients Selection dialog box and allow the user to select the ingredients. If the user clicks OK to dismiss the dialog box, we will apply the above scenario.
   Access the OnBnClickedBtnIngredients() event of the main dialog box and change its code as follows:

```
void CFastFoodDlg::OnBnClickedBtnIngredients()
{
    // TODO: Add your control notification handler code here
    // Check the Ingredients check box state
    int IngredientState = IsDlgButtonChecked(IDC_CHK_REGULARS);

    // A variable for the Ingredients dialog box
    CIngredientsDlg Dlg;// = new CIngredientsDlg;

    // Before displaying the dialog box, synchronize its
    // check boxes with the state of the Ingredients check box
    // If the Ingredients check box is not checked at all
    if( IngredientState == 0 )
    {
            // then uncheck all the check boxes of the Ingredients dialog
            Dlg.m_bLettuce = FALSE;
            Dlg.m_bOnion   = FALSE;
            Dlg.m_bTomato  = FALSE;
            Dlg.m_bPickles = FALSE;
    } // If the Ingredients check box is checked
    else if( IngredientState == 1 )
    {
            // then check all the check boxes of the Ingredients dialog
            Dlg.m_bLettuce = TRUE;
            Dlg.m_bOnion   = TRUE;
            Dlg.m_bTomato  = TRUE;
            Dlg.m_bPickles = TRUE;
    } // Since the Ingredients check box appears indeterminate,
    else if( IngredientState == 2 )
    {
            // select the lettuce and the tomato
            Dlg.m_bLettuce = TRUE;
            Dlg.m_bOnion   = FALSE;
            Dlg.m_bTomato  = TRUE;
            Dlg.m_bPickles = FALSE;
    }
    // Now is the time to display the dialog box

    // Call the Ingregients Selection dialog box for the user
    // If the user clicked OK when closing the dialog box,
    if( Dlg.DoModal() == IDOK )
    {
            // if no check box is checked
          if( (Dlg.m_bLettuce == FALSE) &&
              (Dlg.m_bOnion   == FALSE) &&
              (Dlg.m_bTomato  == FALSE) &&
              (Dlg.m_bPickles == FALSE) )
              m_Regulars.SetCheck(0);               // then uncheck this one
          // if all check boxes are checked
          else if( (Dlg.m_bLettuce == TRUE) &&
                   (Dlg.m_bOnion    == TRUE) &&
                   (Dlg.m_bTomato   == TRUE) &&
                   (Dlg.m_bPickles  == TRUE) )
                    m_Regulars.SetCheck(1);         // then check this one
         else // if at least one check box is checked and at one is unchecked
              m_Regulars.SetCheck(2);  // then set this one as indeterminate
    }
    // If the user clicked Cancel, don't do nothing
}
```

7. Now we can calculate the price of the sandwich.
   In the CFastFoodDlg class, declare a member function of type void and named EvaluatePrice

```
private:
    void EvaluatePrice(void);
};
```

8. Implement the method as follows:

```
void CFastFoodDlg::EvaluatePrice(void)
{
	double PriceBread, PriceMeat,
		     PriceCheese, PriceBacon, TotalPrice;

	// The price of bread is $0.85
	PriceBread   = 0.85;

	// To get the price of the meat, find out what button
	// is selected in the Meat group box
	switch(GetCheckedRadioButton(IDC_MEAT, IDC_CHICKEN_BREAST))
	{
	case IDC_MEAT:
		// The beef patty is $1.50
		PriceMeat  = 1.50;
		break;
	case IDC_GRILLED_CHICKEN:
	case IDC_CHICKEN_BREAST:
		// Cheicken breast and grilled chicken are $1.80 each
		PriceMeat = 1.80;
		break;
	default: // Why are we adding this?
		PriceMeat = 0.00;
	}

	// There is no extra cost for the Regular ingredients
	// and nothing to add for the sweetener

	// On the other hand,
	// if the customer wants cheese, $0.30 is added to the price
	if( IsDlgButtonChecked(IDC_CHK_CHEESE) == 1 )
		PriceCheese = 0.30;
	else // otherwise, the customer don't want no cheese
		PriceCheese = 0.00;
	// If the customer wants bacon, $0.45 is added to the price
	if( IsDlgButtonChecked(IDC_CHK_BACON) == 1 )
		PriceBacon = 0.45;
	else
		PriceBacon = 0.00;

	// Now, we can calculte the total price
	TotalPrice = PriceBread + PriceMeat + PriceCheese + PriceBacon;
	// Convert the price to a null-terminated string
	char StrTotalPrice[10];
	sprintf(StrTotalPrice, "$%.2f", TotalPrice);
	// Display the price
	m_TotalPrice.SetWindowText(StrTotalPrice);
}
```

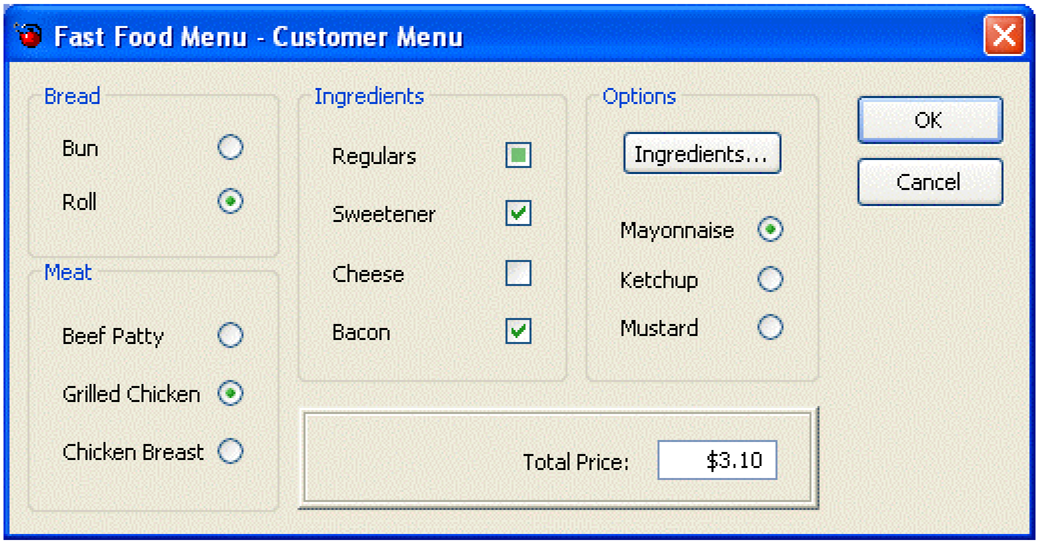
9. To update the price and its display live whenever the user makes a new selection, generate a **BN_CLICKED** message for the IDC_MEAT, the IDC_GRILLED_CHICKEN, the IDC_CHICKEN_BREAST, the IDC_CHK_CHEESE, and the IDC_CHK_BACON check boxes

10. In the body of each, simply call the above EvaluatePrice() method. Here are two examples to save printing paper:

```
void CFastFoodDlg::OnBnClickedGrilledChicken()
```

```
{
	// TODO: Add your control notification handler code here
	EvaluatePrice();
}

void CFastFoodDlg::OnBnClickedChickenBreast()
{
	// TODO: Add your control notification handler code here
	EvaluatePrice();
}

void CFastFoodDlg::OnBnClickedChkCheese()
{
	// TODO: Add your control notification handler code here
	EvaluatePrice();
}

void CFastFoodDlg::OnBnClickedChkBacon()
{
	// TODO: Add your control not ification handler code here
	EvaluatePrice();
}
```

11. Test the application



Fast Food Menu - Customer Menu

Bread
- Bun ○
- Roll ●

Meat
- Beef Patty ○
- Grilled Chicken ●
- Chicken Breast ○

Ingredients
- Regulars ▣
- Sweetener ☑
- Cheese □
- Bacon ☑

Options
- Mayonnaise ●
- Ketchup ○
- Mustard ○

Total Price: $3.10

12. Close the dialog and retun to MSVC

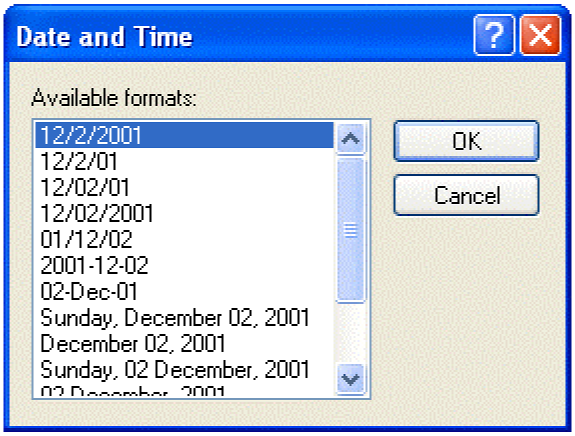
Chapter 20: List-Based Controls

✍ List Boxes

✍ Combo Boxes

20.1 List Boxes

20.1.1 Overview



12/2/2001
12/2/01
12/02/01
12/02/2001
01/12/02
2001-12-02
02-Dec-01
Sunday, December 02, 2001
December 02, 2001
Sunday, 02 December, 2001

A list box presents a list of items to choose from. Each item displays on a line. The user makes a selection by clicking in the list. Once clicked, the item or line on which the mouse landed becomes highlighted, indicating that the item is the current choice. Once an item is selected, to make a different selection, the user would click another. The user can also press the up and down arrow keys to navigate through the list and make a selection.

As far as item selection is concerned, there are two types of list boxes: single selection and multiple selection.

One of the main reasons for using a list box is to display a list of items to the user. Sometimes the list would be very large. If the list is longer than the available space on the control, the operating system would provide a scroll bar that allows the user to navigate up and down to access all items of the list. Therefore, you will have the option of deciding how many items to display on the list.

Practical Learning: Introducing List Boxes

1. Start a new MFC Application named **TableWizard**
2. Create it as Dialog Based and set the Dialog Title to **Table Wizard**
3. On the Toolbox, click the Group Box control and click on the Top left section of the dialog
4. Change its **Caption** to **Categories**
5. On the Toolbox, click the Radio Button control and click in the group box on the dialog box
6. Change its **ID** to **IDC_RDO_BUSINESS** and its **Caption** to **&Business**
7. Check the **Group** property of the radio button or set this property to True
8. Add a **CButton Control Variable** for IDC_RDO_BUSINESS. Name it **m_SelectBusiness**
9. Add another radio button under the Business control. Change its ID to **IDC_RDO_PERSONAL** ant its **Caption** to **&Personal**
10. Check the **Group** check box of the radio button or set the Group property to True
11. Add a **CButton Control Variable** for the new radio button named **m_SelectPersonal**

12. Uncheck the **Group** property of the IDC_RDO_PERSONAL radio button or set this property to False
13. On the Toolbox, click the Static Text button and click on the dialog under the group box
14. Change its **Caption** to **Sample Tables**

20.1.2 List Box Fundamentals

To include a list box in your application, from the Controls toolbox, you can click the List Box button and click on a parent window. The MFC list box is based on the **CListBox** class. A newly added list box appears as a rectangular empty box with a white background.

After creating the list box, the user can select an item by clicking it in the list. The newly selected item becomes highlighted. If the user clicks another item, the previous one looses its highlighted color and the new item becomes highlighted. By default, the user can select only one item at a time. The ability to select one or more items is controlled by the **Selection** property whose default value is **Single**. If you do not want the user to be able to select any item from the list, you have two main alterrnatives. If you set the **Selection** property to **None**, when the user clicks an item, a rectangle is drawn around the item but the highlighted color is not set. Alternatively, you can create the control with the **Disabled** property set to True.

If you want the user to be able to select more than one item, at design time, set the **Selection** property to **Multiple**. If you are programmatically creating the control, you can provide this ability by adding the **LBS_MULTIPLESEL** style. When a list box is created with the **Multiple Selection** set, to select an item, the user would click it. To select an additional item, the user can click it subsequently. If the user clicks an item that is highlighted, meaning it was already selected, the item becomes selected, loosing its highlighted color. The user can continue using this feature to build the desired list.

Microsoft Windows allows a user to select items on a list box (in fact on many other list-based controls or views) by using a combination of mouse click and the keyboard. To select items in a range, the user can press and hold Shift. Then click an item from the other range and release Shift. All items between both clicks would be selected. To select items at random, the user can press and hold Ctrl. Then click each desired item, which creates a series of cumulative selections. To provide this ability, at design time, you can set the **Selection** property to **Extended**. If you are programmatically creating the list box, add the **LBS_EXTENDEDSEL** style to it.

When building the list of items of a list box, by default, the items are rearranged in alphabetical order. Even if you add an item to an already created list, the new item is added to the right order. This arrangement means that the list is sorted. The ability to sort a list box or not is controlled by the **LBS_SORT** style. If you want the items to keep their position as they are added, set the **Sort** property to True.

Practical Learning: Creating List Boxes

1. On the Toolbox, click the List Box button and click on the dialog under the group box.

2. Change its **ID** to **IDC_SAMPLE_TABLES**
3. Remove the check box on the **Sort** property or set it to False
4. Add a Control Variable for the IDC_SAMPLE_TABLES list box and name it **m_SampleTables**
5. In the header file of the CTableWizardDlg class, declare two constant **CString** arrays as follows:

```
const CString Business[] = { "Contacts", "Customers", "Employees",
                     "Products", "Orders", "Suppliers",
                          "Payments", "Invoices","Projects",
                          "Events", "Transactions" };

const CString Personal[] = { "Addresses", "Video Collection",
                          "Authors",    "Books", "Categories",
                          "Music Collection", "Investments" };
```

6. At the end of the class, declare two private integer variables as follows:

```
private:
    // Variables used to hold the number of items in their respective array
    int SizeBusiness;
    int SizePersonal;
};
```

7. Save All

20.1.3 List Box Methods

A list box is based on the **CListBox** class. Therefore, if you want to programmatically create a list box, declare a CListBox variable or pointer using its constructor. To initialize the control, call its **Create()** method. Here is an example:

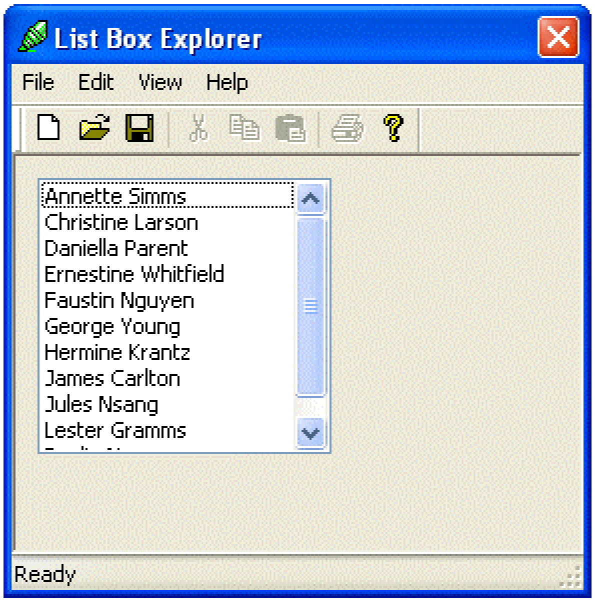
```
void CExoListBox1View::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        GetParentFrame()->RecalcLayout();
        ResizeParentToFit();

        CListBox *m_SimpleList = new CListBox;

        m_SimpleList->Create(WS_CHILD | WS_VISIBLE | WS_VSCROLL,
                             CRect(20, 20, 120, 120), this, 0x118);
```

After adding a list box, you can “fill” it up with items. This is done by calling the **CListBox::AddString()** method. Its syntax is:

```
int AddString(LPCTSTR lpszItem);
```

This method expects a null-terminated string as argument and adds this argument to the control. To add more items, you must call this method for each desired item. Here is an example:

```
void CExoListBox1View::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        GetParentFrame()->RecalcLayout();
        ResizeParentToFit();

        m_ListOfStudents.AddString("Jules Nsang");
        m_ListOfStudents.AddString("George Young");
        m_ListOfStudents.AddString("Christine Larson");
        m_ListOfStudents.AddString("James Carlton");
        m_ListOfStudents.AddString("Annette Simms");
        m_ListOfStudents.AddString("Ernestine Whitfield");
        m_ListOfStudents.AddString("Paulin Ngono");
        m_ListOfStudents.AddString("Lester Gramms");
        m_ListOfStudents.AddString("Daniella Parent");
        m_ListOfStudents.AddSt ring("Hermine Krantz");
        m_ListOfStudents.AddString("Faustin Nguyen");
}
```



Annette Simms
Christine Larson
Daniella Parent
Ernestine Whitfield
Faustin Nguyen
George Young
Hermine Krantz
James Carlton
Jules Nsang
Lester Gramms

The items of a list box are arranged as a zero-based array. The top item has an index of zero. The second item as an index of 1, etc. Although the items seem to be added randomly to the list, their new position depends on whether the list is sorted or not. If the list is sorted, each new item is added to its alphabetical position. If the list is not sorted, each item is added on top of the list as if it were the first item. The other items are then "pushed down". If the list is not sorted, you can add an item in the position of your choice. This is done using the **CListBox::InsertString()** method whose syntax is:

```
int InsertString(int nIndex, LPCTSTR lpszItem);
```

The *lpszItem* argument is the string to be added to the list. The *nIndex* argument specifies the new position in the zero-based list. If you pass it as 0, the *lpszItem* item would be made the first in the list. If you pass it as –1, *lpszItem* would be added at the end of the list, unless the list is empty.

Based on this, to delete an item from the list, pass its index to the **CListBox::DeleteString()** method. Its syntax is:

```
int DeleteString(UINT nIndex);
```

For example, to delete the second item of the list, pass a 1 value to this method. If you want to delete all items of the control, call the **CListBox::ResetContent()** method. ITs syntax is:

```
void ResetContent( );
```

This method simply dismisses the whole content of the list box.

At any time, if you want to know the number of items that a list box holds, call the **CListBox::GetCount()** method whose syntax is:

```
int GetCount() const;
```

This method simply returns a count of the items in the list.

Once a list has been created, you and your users can use its items. For example, to select an item, the user clicks it. To programmatically select an item, you can call the **CListBox::SetCurSel()** method. Its syntax is:

```
int SetCurSel(int nSelect);
```

The *nSelect* argument specifies the item to select. To select the fifth item from a list box, you can pass the nSelect argument with a value of 4

```
void CExoListBox1View::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        GetParentFrame()->RecalcLayout();
        ResizeParentToFit();

        . . .

        m_ListOfStudents.SetCurSel(4);
}
```

If an item is selected in the list and you want to find out which one, you can call the **CListBox::GetCurSel()** method. Its syntax is:

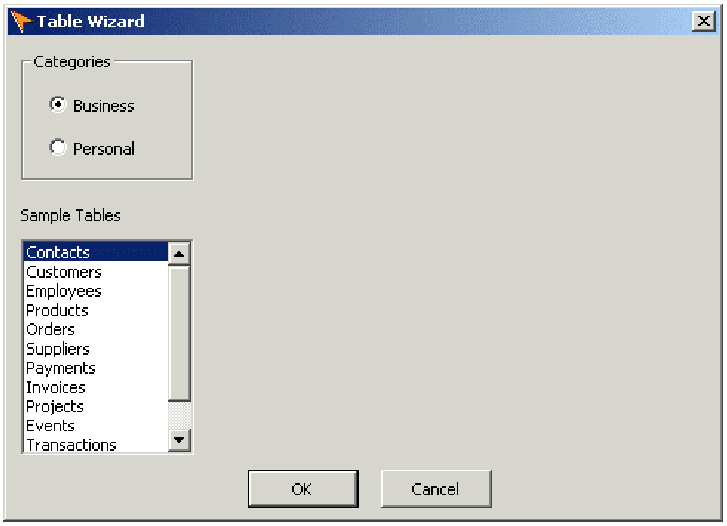
```
int GetCurSel() const;
```

Practical Learning: Using List Box Methods

1. Change the OnInitDialog() member function as follows:

```
BOOL CTableWizardDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);                 // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    // TODO: Add extra initialization here
    // Select the Business radio button
    m_SelectBusiness.SetCheck(1);
    // Get the number of items in the array
    SizeBusiness    = sizeof(Business) / sizeof(CString);
    SizePersonal    = sizeof(Personal) / sizeof(CString);

    // Fill out the Sample Tables list with the Business items
    for(int i = 0; i < SizeBusiness; i++)
                m_SampleTables.AddString(Business[i]);

    // Select the first item in the Sample Tables list box
    m_SampleTables.SetCurSel(0);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. Test your program



3. After testing the dialog box, close it and return to MSVC.

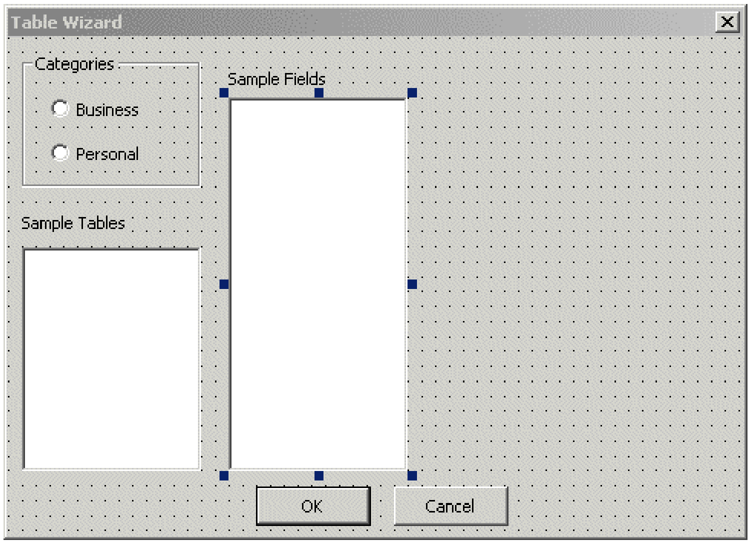
4. **Add** a **BN_CLICKED Event Handler** for the IDC_RDO_BUSINESS radio button and change the name of the function to **OnSelectBusiness**

5. **Add** a **BN_CLICKED Event Handler** for the IDC_RDO_PERSONAL radio button and change the name of the function to **OnSelectPersonal**

6. Implement both events as follows:

```
void CTableWizardDlg::OnSelectBusiness()
{
    // TODO: Add your control notification handler code here
    // Empty the list
    m_SampleTables.ResetContent();

    // Fill out the list with Business items
    for(int i = 0; i < SizeBusiness; i++)
        m_SampleTables.AddString(Business[i]);

    // Select the first item in the list
    m_SampleTables.SetCurSel(0);
}

void CTableWizardDlg::OnSelectPersonal()
{
    // TODO: Add your control notification handler code here
    // Empty the lists
    m_SampleTables.ResetContent();

    // Fill out the list
    for(int i = 0; i < SizePersonal; i++)
                    m_SampleTables.AddString(Personal[i]);

    // Select the first item in the list
    m_SampleTables.SetCurSel(0);
}
```

7. Execute the program to test it. Click the radio buttons and make sure that the content of the Sample Tables list box changes based on the selected radio button.
8. After testing the dialog box, close it and return to MSVC.
9. Add a new Static Text to the right side of the group box with a caption of **Sample Fields**
10. Add a new List Box under the Sample Fields label. Set its ID to **IDC_SAMPLEFIELDS** and remove the check box on its **Sort** property or set it to False



11. **Add** a **CListBox Control Variable** to the IDC_SAMPLEFIELDS control and name it **m_SampleFields**
12. In the header file of CTableWizardDlg class, declare additional CString arrays as follows:

```
const CString Business[] = { "Contacts", "Customers", "Employees",
                              "Products", "Orders", "Suppliers",
                              "Payments", "Invoices", "Projects",
                              "Events", "Transactions" };

const CString Personal[] = { "Addresses", "Video Collection",
                              "Authors", "Books", "Categories",
                              "Music Collection", "Investments" };

const CString Contacts[] = { "MailingListID", "SocialTitle", "FirstName",
                              "MiddleName", "LastName", "Suffix",
                              "Nickname", "HomeAddress", "City",
                                 "StateOrProvince", "PostalCode",
                                 "CountryOrRegion", "HomePhone", "WorkPhone",
                                 "Extension", "MobileNumber", "EmergencyName",
                                 "EmergencyPhone", "Notes" };

const CString Customers[] = { "CustomerID", "CompanyName",
                                 "CompanyAddress", "CompanyCity",
                                 "CompanyState", "companyZIPCode",
                                 "CompanyCountry", "CompanyPhone",
                                 "BillingAddress", "FaxNumber",
                                 "WebSite", "ContactPhone", "ContactEMail" };

const CString Employees[] = { "EmployeeID", "EmployeeNumber",
                                 "DateHired", "Title", "FirstName",
                                 "MI", "LastName", "Address",
                                 "City", "State", "ZIPCode", "Country",
                                 "HomePhone", "WorkPhone", "Extension",
                                 "CellPhone", "EmergencyName",
                                 "EmergencyPhone", "Comments" };

const CString Addresses[] = { "AddressID", "FirstName", "LastName",
                                 "MaritalStatus", "Address", "City",
                                 "StateOrProvince", "PostalCode",
                                 "CountryOrRegion", "EmergencyName",
                                 "EmergencyPhone", "Notes" };

const CString VideoCollection[] = { "VideoID", "VideoTitle",
                                       "CategoryID", "Director", "Rating",
                                       "Length", "VideoReview" };

const CString Categories[] = { "CategoryID", "Category", "Description" };

const CString Authors[] = { "AuthorID", "FullName", "Nationality",
                              "BirthDate", "BirthPlace",
                              "AliveDead", "Comments" };

const CString Books[] = { "BookID", "BookTitle", "Author", "CategoryID",
                            "Publisher", "YearPublished", "NumberOfPages",
                            "ISBNNumber", "BookReview" };
```

13. In the private section of the class, add new integer variables to hold the dimensions of the arrays as follows:

```
private:
    int SizeBusiness;
    int SizePersonal;
    int SizeContacts;
    int SizeEmployees;
    int SizeAddresses;
    int SizeCustomers;
    int SizeVideoCollection;
    int SizeCategories;
    int SizeAuthors;
    int SizeBooks;
};
```

14. Access the TableWizardDlg.cpp source file.
    To make sure that the Sample Fields list is initially filled up with the right fields, change the OnInitDialog() function as follows:

```
BOOL CTableWizardDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);                         // Set big icon
    SetIcon(m_hIcon, FALSE);                // Set small icon

    // TODO: Add extra initialization here
    // Select the Business radio button
    m_SelectBusiness.SetCheck(1);

    // Get the number of items in the array
    SizeBusiness        = sizeof(Business) / sizeof(CString);
    SizePersonal        = sizeof(Personal) / sizeof(CString);
    SizeContacts        = sizeof(Contacts) / sizeof(CString);
    SizeAddresses       = sizeof(Addresses) / sizeof(CString);
    SizeEmployees       = sizeof(Business) / sizeof(CString);
    SizeCustomers       = sizeof(Business) / sizeof(CString);
    SizeVideoCollection = sizeof(VideoCollection) / sizeof(CString);
    SizeCategories      = sizeof(Categories) / sizeof(CString);
    SizeAuthors         = sizeof(Authors) / sizeof(CString);
    SizeBooks           = sizeof(Books) / sizeof(CString);

    // Fill out the Sample Tables list with the Business items
    for(int i = 0; i < SizeBusiness; i++)
                m_SampleTables.AddString(Business[i]);

    // Select the first item in the Sample Tables list box
    m_SampleTables.SetCurSel(0);

    // Fill out the Sample Fields list box
    for(int j = 0; j < SizeContacts; j++)
        m_SampleFields.AddString(Contacts[j]);

    // Select the first item in the Sample Fields list box
    m_SampleFields.SetCurSel(0);

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

15. To make sure that the Sample Fields list is updated when one of the radio buttons is selected, change the OnSelectBusiness() and the OnSelectPersonal() functions as follows:

```
void CTableWizardDlg::OnSelectBusiness()
{
    // TODO: Add your control notification handler code here
    // Empty the list
    m_SampleTables.ResetContent();
    m_SampleFields.ResetContent();

    // Fill out the list with Business items
    for(int i = 0; i < SizeBusiness; i++)
        m_SampleTables.AddString(Business[i]);

    // Select the first item in the list
    m_SampleTables.SetCurSel(0);
```

```
	// Fill out the Sample Fields list and select the first item
	for(int j = 0; j < SizeContacts; j++)
		m_SampleFields.AddString(Contacts[j]);
	m_SampleFields.SetCurSel(0);

}

void CTableWizardDlg::OnSelectPersonal()
{
	// TODO: Add your control notification handler code here
	// Empty the lists
	m_SampleTables.ResetContent();
	m_SampleFields.ResetContent();

	// Fill out the list
	for(int i = 0; i < SizePersonal; i++)
			m_SampleTables.AddString(Personal[i]);

	// Select the first item in the list
	m_SampleTables.SetCurSel(0);

	// Fill out the Sample Fields list and select the first item
	for(int j = 0; j < SizeAddresses; j++)
		m_SampleFields.AddString(Addresses[j]);
	m_SampleFields.SetCurSel(0);
}
```

16. Test your program and return to MSVC

20.1.4 List Box Messages and Events

In order to user the list box as a control, it must first receive focus. This can be visible by either the first item having a rectangle drawn around it or at least on the items being selected. When the list box receives focus, it sends the **LBN_SETFOCUS** notification message. On the other hand, once the user clicks another control or another application, the list box loses focus and consequently sends the **LBN_KILLFOCUS** notification message.

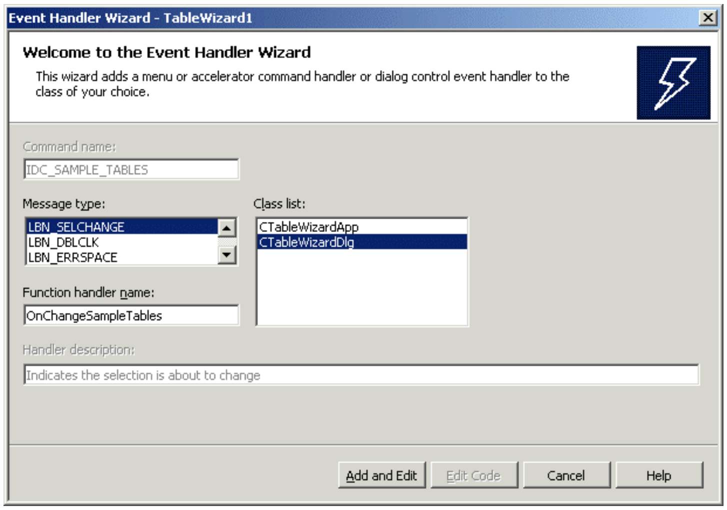
As clicking is the most performed action on a list box, when the user clicks an item, it becomes selected. If the user clicks another item of a non-Multiple Selection list box, as the selection is going to be changed, the **LBN_SELCHANGE** notification message is sent. If the selection is cancelled, the **LBN_SELCANCEL** notification message is sent.

If the user double-clicks an item in the list box, the **LBN_DBLCLK** notification message is sent.

Practical Learning: Exploring List Boxes

1. When the user clicks a table in the Sample Tables list box, we must change the content of the Sample Fields list box and fill it out with the fields that are part of the selected table.
   Right-click the IDC_SAMPLE_TABLES list box and click Add Event Handler
2. In the Messages Type list, click LBN_SELCHANGE and, in the Class List, make sure CTableWizardDlg is selected

3. In the Function Handler Name edit box, change the name to OnChangeSampleTables



Event Handler Wizard - TableWizard1

Welcome to the Event Handler Wizard

This wizard adds a menu or accelerator command handler or dialog control event handler to the class of your choice.

Command name: IDC_SAMPLE_TABLES

Message type: LBN_SELCHANGE, LBN_DBLCLK, LBN_ERRSPACE

Class list: CTableWizardApp, CTableWizardDlg

Function handler name: OnChangeSampleTables

Handler description: Indicates the selection is about to change

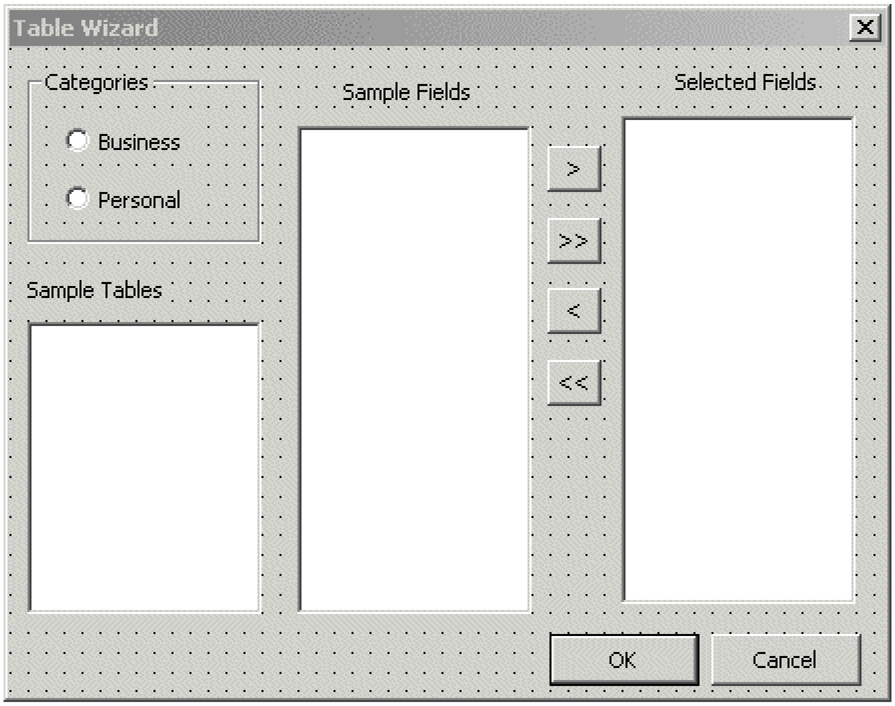
4. Click Add and Edit
5. Implement the function as follows:

```
void CTableWizardDlg::OnChangeSampleTables()
{
	// TODO: Add your control notification handler code here
	// Before filling out the Sample Fields list box, empty it first
	m_SampleFields.ResetContent();

	// This string variable will identify the selected table
	CString ItemSelected;

	// Get the name of the item selected in the Sample Tables list box
	// and store it in the CString variable declared above
	m_SampleTables.GetText(m_SampleTables.GetCurSel(), ItemSelected);

	// Fill out the Sample Fields list box according to the selected table
	if( ItemSelected == "Contacts" )
	{
		for(int i = 0; i < SizeContacts; i++)
			m_SampleFields.AddString(Contacts[i]);
	}
	else if( ItemSelected == "Customers" )
	{
		for(int i = 0; i < SizeCustomers; i++)
			m_SampleFields.AddString(Customers[i]);
	}
	else if( ItemSelected == "Employees" )
	{
		for(int i = 0; i < SizeEmployees; i++)
			m_SampleFields.AddString(Employees[i]);
	}
	else if( ItemSelected == "Addresses" )
	{
		for(int i = 0; i < SizeAddresses; i++)
			m_SampleFields.AddString(Addresses[i]);
	}
	else if( ItemSelected == "VideoCollection" )
	{
		for(int i = 0; i < SizeVideoCollection; i++)
```

```
                        m_SampleFields.AddString(VideoCollection[i]);
    }
    else if( ItemSelected == "Categories" )
    {
            for(int i = 0; i < SizeCategories; i++)
                        m_SampleFields.AddString(Categories[i]);
    }
    else if( ItemSelected == "Authors" )
    {
            for(int i = 0; i < SizeAuthors; i++)
                        m_SampleFields.AddString(Authors[i]);
    }
    else if( ItemSelected == "Books" )
    {
            for(int i = 0; i < SizeBooks; i++)
                        m_SampleFields.AddString(Books[i]);
    }
    else // If we didn't create a list for the selected table, show nothing
            m_SampleFields.ResetContent();

    // Select the first item, if any, in the Sample Fields list box
    m_SampleFields.SetCurSel(0);
}
```

6. Test the program. Click different radio buttons and click different tables in the Sample Tables list

7. Return to MSVC and design the dialog as follows:



8. Add a new Static Text to the top right section of the dialog and set its caption to **Selected Fields**

9. Add a new List Box under the Selected Fields label.

10. Change its ID to **IDC_SELECTEDFIELDS** and remove the check mark of its Sort property or set it to False
11. Add a CListBox Control Variables for the IDC_SELECTEDFIELDS control and name it **m_SelectedFields**
12. Add a Button between the right list boxes. Set its **ID** to **IDC_BTN_SELECTONE** and its **Caption** to >
13. Add a CButton Control Variable for the > button and name it **m_SelectOne**
14. Add another Button under the previous one. Set its **ID** to **IDC_BTN_SELECTALL** and its **Caption** to >>
15. Add a CButton Control Variable for the >> button and name it **m_SelectAll**
16. Add another Button under the previous one. Set its **ID** to **IDC_BTN_REMOVEONE** and its Caption to <
17. Add a CButton Control Variable for the < button and name it **m_RemoveOne**
18. Add another Button under the previous one. Set its **ID** to **IDC_BTN_REMOVEALL** and its **Caption** to <<
19. Add a CButton Control Variable for the << button and name it **m_RemoveAll**
20. First of all, when the dialog box opens, the Selected Fields list is empty, which means the Remove buttons are not useful. Therefore, we should disable them. To do this, add the following two lines to the OnInitDialog() member function:

```
	// Since the Selected Field list box is empty, disable the Remove buttons
	m_RemoveOne.EnableWindow(FALSE);
	m_RemoveAll.EnableWindow(FALSE);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

21. Add a BN_CLICKED Event Handler for the > button and name it **OnSelectOne**
22. To be able to select an item from the Sample Fields list and transfer it to the Selected Fields list box, implement the function as follows:

```
void CTableWizardDlg::OnSelectOne()
{
	// TODO: Add your control notification handler code here
	// Variable that identifies what item is selected in the sample fields
	CString SourceSelected;

	// Find out what item is selected
	// Store it in the above variable
	m_SampleFields.GetText(m_SampleFields.GetCurSel(), SourceSelected);

	// Add the item to the Selected Fields list box
	m_SelectedFields.AddString(SourceSelected);
}
```

23. To enhance the application, we need to make sure that the user can double-click an item in the Sample Fields list box and obtain the same behavior as if the > button had been clicked

    Therefore, add an LBN_DBLCLK for the IDC_SAMPLEFIELDS list box and name it **OnDblClkSampleFields**
24. Implement it as follows:

```
void CTableWizardDlg::OnDblClkSampleFields()
{
	// TODO: Add your control notification handler code here
	OnSelectOne();
}
```

25. Test the application and return to MSVC

26. One of the problems we have at this time is that the user can select the same item more than once, which is not practical for a database's table. Therefore, when the user selects an item, we first need to check whether the Selected Fields list box already contains the item. If it does, we will not allow adding the item. We will provide an alternate solution shortly. Also, once at least one item has been added to the Selected Fields list, we can enable the Remove buttons.
Change the content of the OnSelectOne() function as follows:

```
void CTableWizardDlg::OnSelectOne()
{
	// Variable that identifies what item is selected in the sample fields
	CString SourceSelected;

	// Find out what item is selected
	// Store it in the above variable
	m_SampleFields.GetText(m_SampleFields.GetCurSel(), SourceSelected);

	// Find out if the Selected Fields list is empty
	if( m_SelectedFields.GetCount() == 0 )
	{
		// Since the list is empty, add the selected item
		m_SelectedFields.AddString(SourceSelected);
		// Select the newly added item
		m_SelectedFields.SetCurSel(0);
	}
	else // Since the list is not empty
	{
		// Look for the selected item in the Selected Fields list
		int Found = m_SelectedFields.FindString(0, SourceSelected);

		// If the item is not yet in the Selected Fields list, prepare to add it
		if( Found == -1 )
		{
			// Because there is always an item selected in any of our list,
			// get the index of the currently selected item
			int CurrentlySelected = m_SelectedFields.GetCurSel();

			// Add the new item under the selected one
			m_SelectedFields.InsertString(CurrentlySelected + 1,
SourceSelected);

			// Select the newly added item
			m_SelectedFields.SetCurSel(CurrentlySelected + 1);
		}
	}

	// Since an item has been selected, enable the Remove buttons
	m_RemoveOne.EnableWindow();
	m_RemoveAll.EnableWindow();
}
```

27. Execute the program to test it. Double-click various items in the Sample Fields list box and observe the behavior.

28. After using it, close the dialog box and return to MSVC

29. Add a BN_CLICKED Event Handler for the >> button and name it OnSelectAll

30. We need to let the user select all items in the Sample Fields list and copy them to the Selected Fields list. As previously, we need to avoid duplicating the names of fields in the same table. Therefore, to add the items to the list, first select one at a time, look for it in the Selected Fields list. If it doesn't exist, then add it. If it already exists, ignore it and consider the next item. This behavior can be performed in a **for** loop that is used to navigate the items in an array.
Implement the OnSelectAll() event as follows:

```
void CTableWizardDlg::OnSelectAll()
{
	// TODO: Add your control notification handler code here
	// This string will represent an item in the Sample Fields list
	CString Item;

	// This "for" loop will be used to navigate the Sample Fields list
	for(int i = 0; i < m_SampleFields.GetCount(); i++)
	{
		// Consider an item in the Sample Fields list
		// Store it in the above declared Item variable
		m_SampleFields.GetText(i, Item);

		// Look for the item in the Selected Fields list
		// Make sure you start at the beginning of the list, which is item 0
		int Found = m_SelectedFields.FindString(0, Item);

		// If the item is not yet in the Selected Fields list, prepare to add it
		if( Found == -1 )
		{
			// Add the new item at the end of the Selected Fields list
			m_SelectedFields.AddString(Item);

			// Select the last item of the Selected Fields list
			m_SelectedFields.SetCurSel(m_SelectedFields.GetCount() - 1);
		} // Consider the next item
	}

	// Since there is at least one item in the Selected Fields list,
	// enable the Remove buttons
	m_RemoveOne.EnableWindow();
	m_RemoveAll.EnableWindow();
}
```

31. Test your program and return to MSVC

32. Add a BN_CLICKED Event Handler for the < button and name it **OnRemoveOne**

33. The < button works by removing, from the Selected Fields list box, the item that is selected. To do this, we first need to make sure that an item is selected. All the previous code took care of selecting an item in each list. The problem is that, this time, the selected item is removed. Therefore, we have the responsibility of selecting a new item.
When an item has been re moved, we will select the item that was under it. What if we had removed the item that was at the bottom of the list? In that case, we will

select the new last item in the list.
To do this, implement the OnRemoveOne() event as follows:

```
void CTableWizardDlg::OnRemoveOne()
{
    // TODO: Add your control notification handler code here
    // Based on previous code, an item should be selected
    // in the Selected Fields list
    // Get the index of the currently selected item
    int CurrentlySelected = m_SelectedFields.GetCurSel();

    // Remove the item from the list
    m_SelectedFields.DeleteString(CurrentlySelected);

    // Find out whether the list has just been emptied or not
    // If the Selected Fields list is empty,
    if( m_SelectedFields.GetCount() == 0 )
    {
            // then disable the Remove buttons
            m_RemoveOne.EnableWindow(FALSE);
            m_RemoveAll.EnableWindow(FALSE);
    }
    else // Since the Selected Fields list still contains something
    {
            // Find out if the item that has just been removed was the last in the list
            if( CurrentlySelected == m_SelectedFields.GetCount() )
            {
                    // Select the last item in the list
                    m_SelectedFields.SetCurSel(m_SelectedFields.GetCount() - 1);
            }
            else // Otherwise
            {
                    // Select the item that was above the one that was just deleted
                    m_SelectedFields.SetCurSel(CurrentlySelected);
            }
    }
}
```

34. We also want the user to be able to remove an item by double-clicking it. Of course, this is not only optional but also application dependent because another program would have a different behavior when the item is double-clicked.
    Add an LBN_DBLCLK Event Handler for the IDC_SELECTEDFIELDS list box and name it OnDblClkSelectedFields

35. Implement it as follows:

```
void CTableWizardDlg::OnDblClkSelectedFields()
{
    // When the user double-clicks an item in the Selected Fields list,
    // behave as if the < button has been clicked
    OnRemoveOne();
}
```

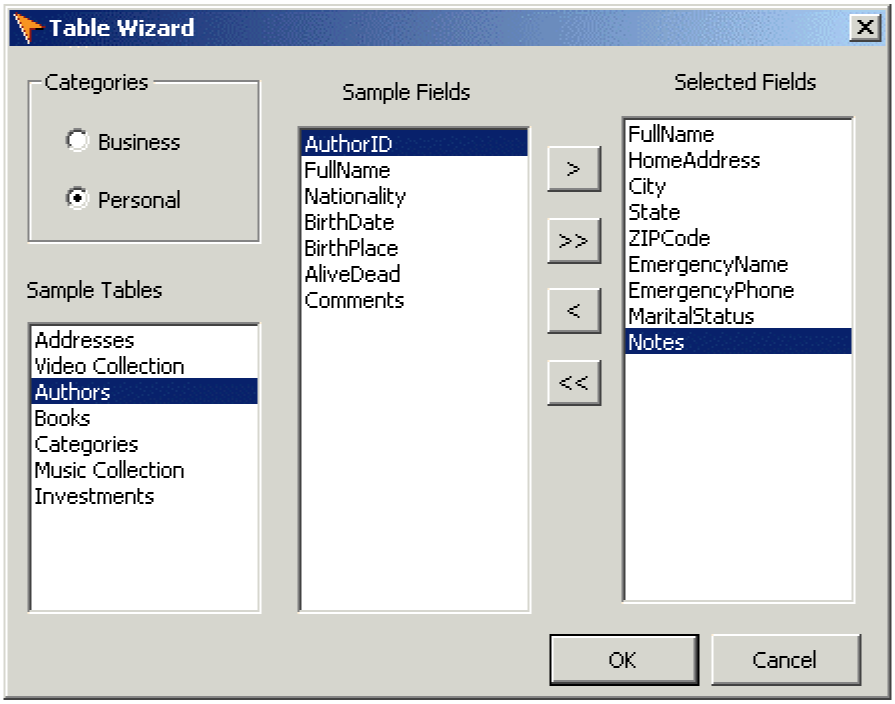
36. Test your program and return to MSVC

37. Add a BN_CLICKED Event Handler for the << button and name it **OnRemoveAll**.

38. The << button is used to simply empty the Selected Fields list. It does not perform any checking of any kind. The only thing we need to do is to disable the Remove buttons after the Selected Fields list has been emptied.
    Therefore, implement the function as follows:

```
void CTableWizardDlg::OnRemoveAll()
{
	// TODO: Add your control notification handler code here
	m_SelectedFields.ResetContent();

	// Since the Selected Fields list is now empty,
	// disable the Remove buttons
	m_RemoveOne.EnableWindow(FALSE);
	m_RemoveAll.EnableWindow(FALSE);
}
```

39. Test your program



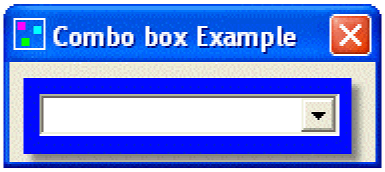
40. Return to MSVC.

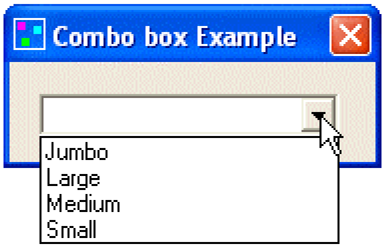
20.2 Combo Boxes

20.2.1 Overview

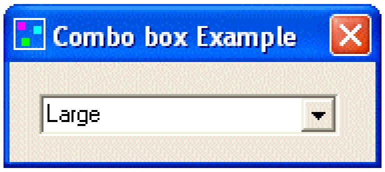
A combo box is a Windows control that holds a list of items. Each item can be a null-terminated string or it can be made of a bitmap and a string. A combo box shares a lot of characteristics with a list but there are more variances of a combo box.

As far as looks are concerned, there are two types of combo boxes. The most regularly used combo box is made of two sections. The main part is an edit box. On the right of the edit box, there is a button with a down pointing arrow:



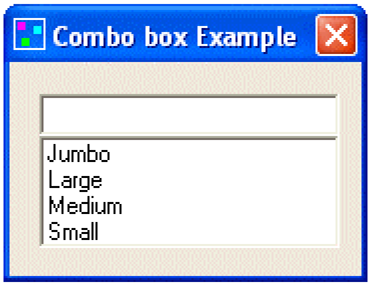


To use this type of combo box, the user would click the down pointing arrow button. This causes a list of items to display. If the user finds the desired item in the list, he or she can click it. The item clicked becomes the selection and it gets in the edit box side of the combo box:

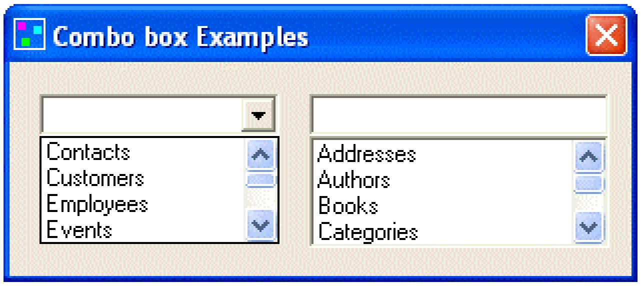


If the user does not find the desired item, he or she can click the down pointing arrow. In this case, no selection is made and the combo box remains as it was before the button was clicked, whether it had an item or not.

A combo box is called "Drop Down" or Drop List if it is made of an edit box and a down pointing arrow, as described above. Another type of combo box is referred to as Simple. This kind displays its items like a list box. The user does not have to click an arrow button to display the list. When a simple combo box displays, the user can locate the desired item in the list and click it. The selected item becomes the value of the edit side.



Like a list box or a group of radio buttons, a combo box is used to display a list of items to the user to select from. Like a series of radio buttons, a combo box allows the user to select only one item from the list. Like the list box control, if the number of items is higher than the allocated space can accommodate, the combo box is equipped with a vertical scroll bar, whether it is a drop type or a simple kind:



Over a list box, a combo box (especially the drop kinds) has the advantage of saving space as it can use only as much space as a combination of an edit box and a small button.

Practical Learning: Introduction to Combo Boxes

1. Start a new MFC Application and name it **Clarksville Ice Scream2**
2. Create it as a Dialog Based without the About Box. Set the Dialog Title to Clarksville Ice Scream
3. Click Finish

20.2.2 Combo Box Properties

To create a combo box, you can click the Combo Box button on the Controls toolbox and click the parent window. After adding a combo box, if you want to immediately create its strings, in the Properties window, click the **Data** field. To create the list of items, type the items separated by a semi-colon

By default, a newly added combo box is of drop down type. The kind of display is controlled by the **Type** combo box of the Properties window and its default value is **Dropdown**. This is programmatically equivalent to adding the **CBS_DROPDOWN** style. A drop down combo box allows the user to type a new value if the desired string is not in the list. To do this, the user can click the edit side of the control and start typing. If the user types a string that is longer than the width of the edit box part, the control would cause the computer to start making a beep sound, indicating that the user cannot type beyond the allocated length. If you want the user to be able to type a long string, check or set to True the **Auto HScroll** property. If you are programmatically creating the control, you can do this by adding the **CBS_AUTOHSCROLL** style.

If you do not want the user to be able to type a new string, set the **Type** property to **Drop List** or create the control with a **CBS_DROPDOWNLIST** style. With this type, whether the user clicks the edit part or the arrow button, the list would drop. If the user finds the desired value and clicks it, the list retracts and the new selection displays in the edit part of the control. If the user does not find the desired item in the list, he or she can click the edit box or the arrow button. In this case, the selection would not be changed. If you want the list part of the combo box to always display, like the list box control, set the **Type** property to **Simple** or create it with the **CBS_SIMPLE** style.

As described above, when the list of items is too long for the reserved rectangular area of the list side to display, the control gets automatically equipped with a vertical scroll bar. This is because its **Vertical Scroll** check box or property is automatically checked or set to True. If you allow or add this style, if the list is not too long and can be accomodated by the rectangle, no scroll bar would be displayed. If the list is too long and you set this property, then a vertical scroll bar would automatically appear on the control. If you insist on displaying a vertical scroll bar even if the rectangle is long enough to display all items, check the Disable No Scroll check box or set it to True. In this case, if the rectangle can accommodate all items, a disabled vertical scroll bar would appear on the control. y either checking the (or setting it to True) or creating the combo box with the **WS_HSCROLL** window style. Based on this convenience, unless you have a strong reason to do otherwise, you should always allow the vertical scroll bar.

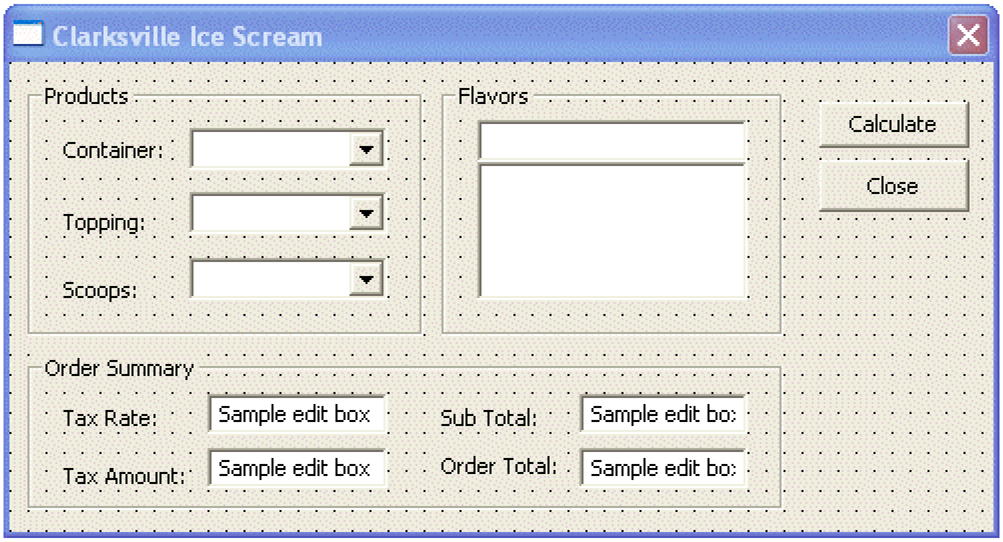
The combo boxes we have mentioned so far are created by their control, in which case the combo boxes are responsible for creating and updating their list. You can create a combo box that relies on its owner for all related operations. Such a combo box is referred to as owner draw. When an owner is responsible for drawing the items of a combo box, it can set the value of each item as it sees fit. For example, different items of the same list can use different fonts. They can display different types of items and they can even draw anything on their line. The ability to feature a combo box as owner draw or not is specified using the Owner Draw combo box of the Properties window. Its default value is No. To create an owner draw combo box where all items have the same height, set this property to the **Fixed** value. This is equivalent to adding the **CBS_OWNERDRAWFIXED** style to a dynamic combo box. On the other hand, if you want items to have different heights, set the Owner Draw value to Variable or create he control with a **CBS_OWNERDRAWVARIABLE** style.

Practical Learning: Adding Combo Boxes

1. From the Controls toolbox, click the Group Box button and draw a large rectangle on the left side of the dialog box

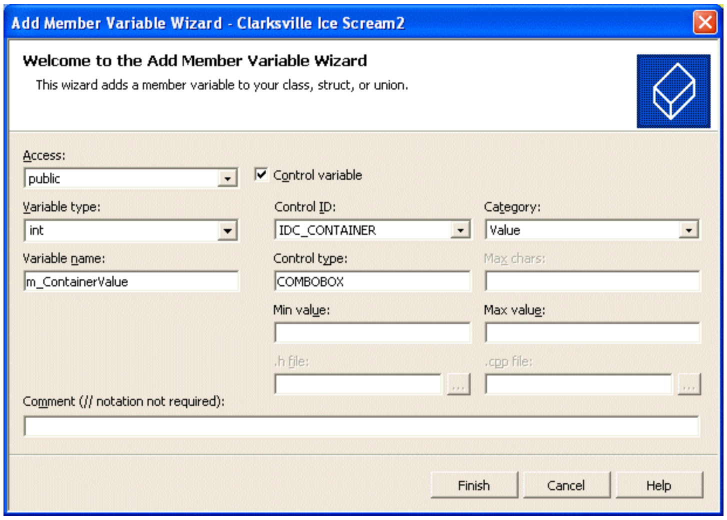
2. On the Controls toolbox, click the Combo Box button and click inside the previously added group box

3. Design the dialog box as follows:



Control	ID	Caption	Other Properties
Group Box		Products	
Static Text		Container:	
Combo Box	IDC_CONTAINER		Type: Drop List Data: Cone;Cup; Bowl Sort: False
Static Text		Topping:	
Combo Box	IDC_TOPPING		Data: None;Cookies;Peanuts;M&M Sort: False
Static Text		Scoops:	
Combo Box	IDC_SCOOPS		Data: One;Two;Three Sort: False
Group Box		Flavor	
Combo Box	IDC_FLAVORS		Type: Simple
Group Box		Order Summary	
Static Text		Tax Rate:	
Edit Box	IDC_TAX_RATE		Align Text: Right
Static Text		Sub Total:	
Edit Box	IDC_SUB_TOTAL		Align Text: Right
Static Text		Tax Amount:	
Edit Box	IDC_TAX_AMOUNT		Align Text: Right
Static Text		Order Total:	
Edit Box	IDC_ORDER_TOTAL		Align Text: Right
Button	IDC_CALCULATE_BTN	C&alculate	

4. To declare and associate a variable for a control, right-click the Container combo box and click Add Variable
5. In the Category combo box, select Value
6. In the Variable Type combo box, select int
7. In the Variable Name, type m_ContainerValue



Add Member Variable Wizard - Clarksville Ice Scream2

Welcome to the Add Member Variable Wizard

This wizard adds a member variable to your class, struct, or union.

Access: public

☑ Control variable

Variable type: int

Control ID: IDC_CONTAINER

Category: Value

Variable name: m_ContainerValue

Control type: COMBOBOX

Max chars:

Min value:

Max value:

.h file:

.cpp file:

Comment (// notation not required):

8. Click Finish
9. On the dialog box, right-click the Flavor combo box and click Add Variable
10. In the Category combo box, accept the Control selected. In the Variable Name, type m_Flavors and press Enter
11. Add the following variables for the other controls (all the Value Variables are CString type):

ID	Control Variable	Value Variable
IDC_TOPPING	m_Topping	
IDC_SCOOPS	m_Scoops	
IDC_TAX_RATE		m_TaxRate
IDC_SUB_TOTAL		m_SubTotal
IDC_TAX_AMOUNT		m_TaxAmount
IDC_ORDER_TOTAL		m_OrderTotal

12. Save All

20.2.3 Combo Box Methods

A combo box is based on the CComboBox class. Therefore, if you want to dynamically create this control, declare a variable or a pointer to CComboBox using its default constructor. To initialize the control, call its Create() method. Here is an example:

```
BOOL CExerciseDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        SetIcon(m_hIcon, TRUE);                    // Set big icon
        SetIcon(m_hIcon, FALSE);           // Set small icon
        // TODO: Add extra initialization here
        CComboBox *Majors = new CComboBox;

        Majors->Create(WS_CHILD | WS_VISIBLE |
                       WS_VSCROLL | CBS_DROPDOWNLIST,
                       CRect(10, 50, 100, 150), this, 0x1448);

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

After creating the control, probably the next action to take consists of creating its items. To add a string to a combo box, you can call the **CComboBox::AddString()** method that uses the same syntax as the **CListBox::AddString()** member function. Here is an example:

```
BOOL CExerciseDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        SetIcon(m_hIcon, TRUE);                 // Set big icon
        SetIcon(m_hIcon, FALSE);        // Set small icon

        // TODO: Add extra initialization here
        CComboBox *Majors = new CComboBox;

        Majors->Create(WS_CHILD | WS_VISIBLE |
                       WS_VSCROLL | CBS_DROPDOWNLIST,
                       CRect(10, 50, 100, 150), this, 0x1448);

        Majors->AddString("Accounting");
        Majors->AddString("Art Education");
        Majors->AddString("Finance");
        Majors->AddString("Biology");

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Most other methods are implemented as they are for the **CListBox** class.

Practical Learning: Using Combo Box Methods

1. To create the list of flavors, in the OnInitDialog() event of the dialog class, type the following:

```
CClarksvilleIceScream2Dlg::CClarksvilleIceScream2Dlg(CWnd* pParent /*=NULL*/)
    : CDialog(CClarksvilleIceScream2Dlg::IDD, pParent)
    , m_ContainerValue(0)
    , m_TaxRate(_T("0.00"))
    , m_SubTotal(_T("0.00"))
    , m_TaxAmount(_T("0.00"))
    , m_OrderTotal(_T("0.00"))
{
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

. . .
```

```
BOOL CClarksvilleIceScream2Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	m_Topping.SetCurSel(0);
	m_Scoops.SetCurSel(0);

	m_Flavors.AddString("French Vanilla");
	m_Flavors.AddString("Cream of Cocoa");
	m_Flavors.AddString("Chocolate Chip");
	m_Flavors.AddString("Cherry Coke");
	m_Flavors.AddString("Butter Pecan");
	m_Flavors.AddString("Chocolate Cooky");
	m_Flavors.AddString("Chunky Butter");
	m_Flavors.AddString("Vanilla Strawberry");
	m_Flavors.AddString("Macedoine");
	m_Flavors.AddString("Mint Chocolate");
	m_Flavors.SetCurSel(6);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

2. Save All

20.2.4 Combo Box Messages and Events

When you add or create a combo box, an amount of space is allocated and during the lifetime of your application, the combo box uses some memory to processing its assignments. If at one time there is not enough memory for these processings, the combo box sends an **ON_CBN_ERRSPACE** message.

Like any control, for the user to use the combo box, it must first have focus. This can be done by the user clicking its edit part, its list part (for a Simple combo box) or its down pointing arrow (for a drop type combo box). When the combo box receives focus, its sends the **ON_CBN_SETFOCUS** message.

Once the control has focus, if the user clicks or had clicked the edit side of the combo box that already had a selected item, if the user starts typing, which would modify the string of the item that was already selected, the combo box would send the **ON_CBN_EDITCHANGE** message (remember that the user can change the string only if the combo box was not created as Drop List). If the user finishes typing or changing the string, just before the altered string is validated, the combo box sends the **ON_CBN_EDITUPDATE** message.

To select an item from a drop type combo box, the user usually clicks the down pointing arrow button to display the control’s list. When the user clicks that button, just before the list displays, the control sends the **ON_CBN_DROPDOWN** message. After this click and this message, the list part of a drop combo box displays and the user must make a decision:

?? If the user finds the desired item in the list, he or she must let the combo box know. This is done by highlighiting it. To do this, the user can either click (with the mouse) or press the arrow keys (on the keyboard) to indicate his or her choice. When this highlighting occurs, the combo box sends the **ON_CBN_SELCHANGE** message, notifying the application that the combo box' selection may be changed soon

?? Once the user has found the desired item and has possibly highlighted it, if using the mouse, he or she can click to select it. If using the keyboard, after locating the item, the user can press Enter. Clicking the desired item or pressing Enter on the highlighted string means that the user has made a definite selection. This causes the control to send an **ON_CBN_SELENDOK** message, notifying the application that the user has made his or her decision.

?? If the user did not find the desired item, he or she may want to dismiss the combo box without making a selection. There are three main ways the user can invalidate a selection. If the user clicks another control or another application, if the list was displaying, it would retract and not selection would be made, event if the user had already highlighted an item. If the user clicks either the edit box part of the combo box for a Drop List type or the down pointing button, the selection is dismissed and if the list of a drop type was displaying, it would retract. If the user presses Esc, the selection would be dismissed. Any of these actions causes the selection to be dismissed or the user to cancel the selection action. This causes the combo box to send an **ON_CBN_SELENDCANCEL** message.

Once the user has clicked an item or pressed Enter to validate a selection, if the combo box was created not as Simple, the list part of the controls retracts to hide itself. At this time, the combo box sends an **ON_CBN_CLOSEUP** message.

If the user finishes using the combo box and moves to another control, the combo box sends an **ON_CBN_KILLFOCUS** message, notifying the application that it (the combo box) has lost focus.

20.3 Image Lists

20.3.1 Introduction

An image list is an array of pictures of the same size. The pictures are created as a single icon or bitmap and each icon or bitmap can be located using its index. The array is zero-based, meansing that the first picture has an index of 0. The second has an index of 1, etc.

An image list is not a traditional control. It does not display to the user who in fact is never aware of it. It is used to complement a control that needs a series of pictures for its own display.

Practical Learning: Introducing Image Lists

1. Start a new MFC Application and name it AirCraft1
2. Create it as a Single Document based on CView, without the initial toolbar, and change the Main Frame Caption to Air Craft Review

3. Access the PreCreateWindow() event of the CMainFrame class and change it as follows:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
      if( !CFrameWnd::PreCreateWindow(cs) )
              return FALSE;
      // TODO: Modify the Window class or styles here by modifying
      //  the CREATESTRUCT cs
      cs.style &= ~FWS_ADDTOTITLE;
      cs.cx = 420;
      cs.cy = 280;

      return TRUE;
}
```

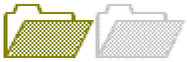
4. Test the application and return to MSVC

20.3.2 Image List Creation

In an MFC application, an image list is based on the **CImageList** class. This object is created in two main steps that do not necessarily follow each other. On one hand, you must have the pictures that will make the list. On the other hand, you must have a **CImageList** variable or pointer.

The easiest way is probaly to first create the picture. There are two kinds: masked or nonmasked. A nonmasked image list is designed as an array of pictures where all pictures have the same width and the same height but the pictures do not have to be square. Here is an example:

A masked image list contains two pictures of the same with and height. Unlike the unmasked image list, both pictures of the masked image normally represent the same illustration. The first picture is in color and the second would be monochrome. Here is an example:



To actually create an image list, declare a **CImageList** variable or pointer and call one of its **Create()** methods to initialize it. It is provided in various versions as follows:

```
BOOL Create(int cx, int cy, UINT nFlags, int nInitial, int nGrow);
BOOL Create(UINT nBitmapID, int cx, int nGrow, COLORREF crMask );
BOOL Create(LPCTSTR lpszBitmapID, int cx, int nGrow, COLORREF crMask );
BOOL Create(CImageList& imagelist1, int nImage1,
            CImageList& imagelist2, int nImage2, int dx, int dy);
BOOL Create(CImageList* pImageList);
```

The first version of this method allows you to describe the type of image list that will be created. This is done by specifying the width (cx) of each picture, the height (cy) of each picture, and a flag for the type of image list to create. The *nInitial* argument is the number of images that the image list will contain. The *nGrow* argument represents the number of images by which the image list can grow.

If you had designed an unmasked bitmap using the image editor and you want to initialize it, call the second or the third versions of the **Create()** method. The *nBitmapID* argument is the identifier of the bitmap. If you want to provide the string that contains the identifiers of the images, pass it as the *lpszBitmapID* argument. The *cx* value represents the width of each picture. The crMask is the color used to mask the transparency of the picture. Each pixel of the picture that matches this color will turn to black. Here is an example of using this Create() method:

```
BOOL CAnimation1Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	CImageList ImgList;

	ImgList.Create(IDB_IMGLIST, 48, 4, RGB(255, 55, 5));

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Besides, or instead of using, the Create() method, you can call **CImageList::Add()** to add a bitmap to the CImageList variable. Its syntaxes are:

```
int Add(CBitmap* pbmImage, CBitmap* pbmMask);
int Add(CBitmap* pbmImage, COLORREF crMask);
int Add(HICON hIcon );
```

The *pbmImage* argument represents the bitmap to be added, unless you want to add an icon, in which case you would use the third version. The *pbmMask* argument is the bitmap that will be used to mask the image list. You can use a color instead, in which case you would pass a COLORREF value as the second argument to the second version.

If you want to remove a picture from the image list, call the **CImageList::Remove()** method. Its syntax:

```
BOOL Remove(int nImage);
```

The *nImage* argument is the index of the picture to be removed. Instead of removing a picture, you can just replace it with another picture. This is done using the **CImageList::Replace()** method whose syntaxes are:

```
BOOL Replace(int nImage, CBitmap* pbmImage, CBitmap* pbmMask);
int Replace(int nImage, HICON hIcon);
```

Once an image list is ready, you can use it directly in an application or make it available to a control that can use it. One way you can use an image list is to display one or more of its pictures on a dialog box, a form, or a view. To do this, you would call the **CImageList::Draw()** method. Its syntax is:

```
BOOL Draw(CDC* pdc, int nImage, POINT pt, UINT nStyle);
```

The first argument, *pdc*, specifies the device context on which you are drawing. The *nImage* argument is the index of the picture you want to draw. The *pt* argument is a **POINT** or a **CPoint** value that specifies the location of the new picture. The *nStyle* argument is a flag that specifies how the picture will be drawn.

Practical Learning: Using an Image List

1. From the AirCraft folder that accompanies this book, import the following bitmaps and change their IDs as follows:

File	ID
AH64.bmp	IDB_AH64
AH64SIDE.bmp	IDB_AH64SIDE
AKIOWA.bmp	IDB_AKIOWA
COMANCHE.bmp	IDB_COMANCHE

2. In the header file of the view class, declare a private **CImageList** variable and name it **ImgList**
3. In the header file of the view class, declare another private **int** variable and name it **nImage**
4. In the constructor of the view class, initialize the image list and the nImage variable as follows:

```
CAirCraft1View::CAirCraft1View()
{
    // TODO: add construction code here
    ImgList.Create(400, 180, ILC_COLOR, 4, 1);

    CBitmap Bmp[4];

    Bmp[0].LoadBitmap(IDB_AH64);
    ImgList.Add(&Bmp[0], RGB(0, 0, 0));
    Bmp[1].LoadBitmap(IDB_AH64SIDE);
    ImgList.Add(&Bmp[1], RGB(0, 0, 0));
    Bmp[2].LoadBitmap(IDB_AKIOWA);
    ImgList.Add(&Bmp[2], RGB(0, 0, 0));
    Bmp[3].LoadBitmap(IDB_COMANCHE);
    ImgList.Add(&Bmp[3], RGB(0, 0, 0));

    nImage = 0;
}
```

5. In the OnPaint() event of the view class, display an image by calling the CImageList::Draw() method as follows:

```
void CAirCraft1View::OnDraw(CDC* pDC)
{
    CAirCraft1Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    ImgList.Draw(pDC, nImage, CPoint(255, 255, 255), ILD_NORMAL);
    // TODO: add draw code for native data here
}
```

6. Open the IDR_MAINFRAME menu. Under the Status Bar menu item of View, add a separator, followed by a new menu item captioned as &AH64 and press Enter

7. Right-click the new menu item and click Add Event Handler…

8. Accept the Message Type as COMMAND. In the Class List, click CAirCraft1View. Accept the name of the function. Then click Add And Edit and implement the event as follows:

```
void CAirCraft1View::OnViewAh64()
{
	// TODO: Add your command handler code here
	nImage = 0;
	Invalidate();
}
```

9. Again, under the View menu, create a new menu item with a caption as **A&H64 Side** and press Enter

10. Add a COMMAND Event Handler for the new menu item. Associate it with the view class and implement it as follows:

```
void CAirCraft1View::OnViewAh64side()
{
	// TODO: Add your command handler code here
	nImage = 1;
	Invalidate();
}
```

11. Again, under the View menu, create a new menu item with a caption as **A&kiowa** and press Enter

12. Add a COMMAND Event Handler for the new menu item. Associate it with the view class and implement it as follows:

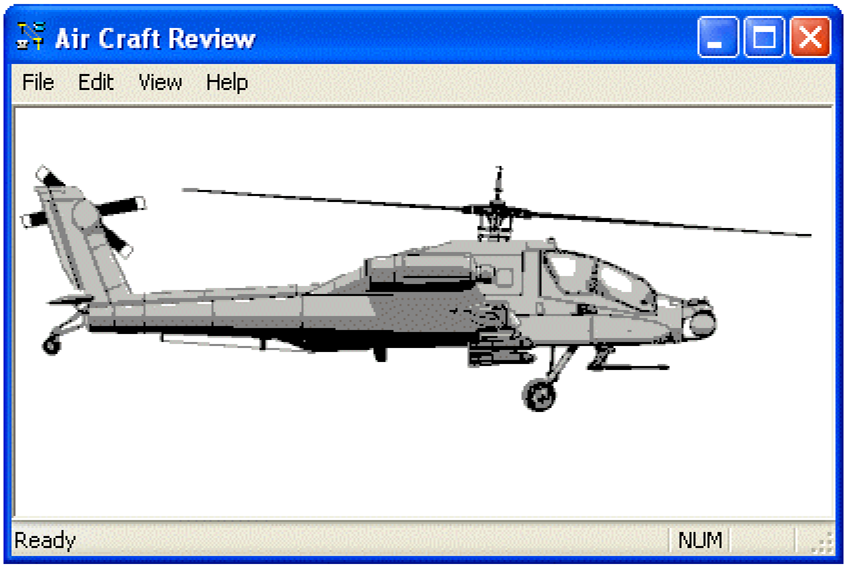
```
void CAirCraft1View::OnViewAkiowa()
{
	// TODO: Add your command handler code here
	nImage = 2;
	Invalidate();
}
```

13. Again, under the View menu, create a new menu item with a caption as **&Commanche** and press Enter

14. Add a COMMAND Event Handler for the new menu item. Associate it with the view class and implement it as follows:

```
void CAirCraft1View::OnViewCommanche()
{
	// TODO: Add your command handler code here
	nImage = 3;
	Invalidate();
}
```

15. Test the application



16. Close it and return to MSVC

Chapter 21: Tree and List Controls

✍ The Tree Control

✍ The Tree View

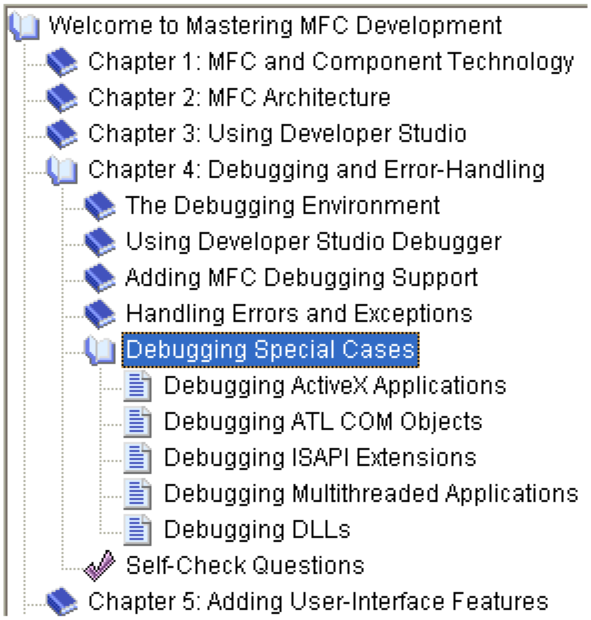
✍ The List Control

☞ The List View

21.1 The Tree Control

21.1.1 Overview

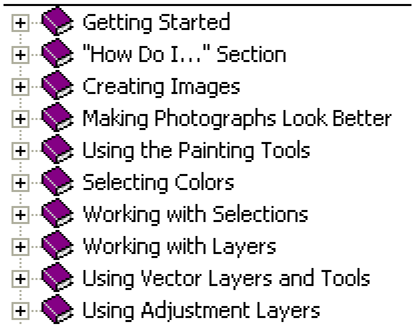
A tree control is an object that displays a hierarchical list of items arranged as a physical tree but a little upside down. The items display in a parent-child format to show those that belong to interrelated categories, such as parent to child and child to grandchild, etc; or folder to subfolder to file. Here is an example of a tree list:



- Welcome to Mastering MFC Development
  - Chapter 1: MFC and Component Technology
  - Chapter 2: MFC Architecture
  - Chapter 3: Using Developer Studio
  - Chapter 4: Debugging and Error-Handling
    - The Debugging Environment
    - Using Developer Studio Debugger
    - Adding MFC Debugging Support
    - Handling Errors and Exceptions
    - Debugging Special Cases
      - Debugging ActiveX Applications
      - Debugging ATL COM Objects
      - Debugging ISAPI Extensions
      - Debugging Multithreaded Applications
      - Debugging DLLs
    - Self-Check Questions
  - Chapter 5: Adding User-Interface Features

Figure 57: A Tree List With One Root

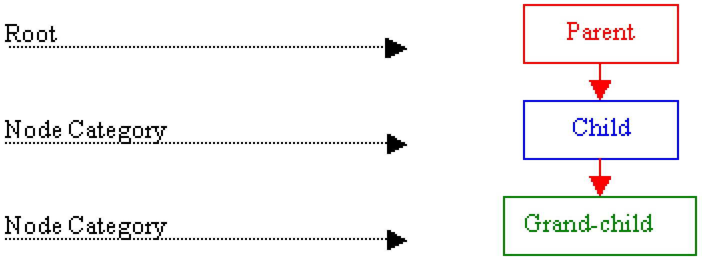
The starting item of the tree is sometimes called the root and represents the beginning of the tree. While most tree list have one root, it is not unusual to have a tree list that has many roots, as long as the tree creator judges it necessary. Here is an example:



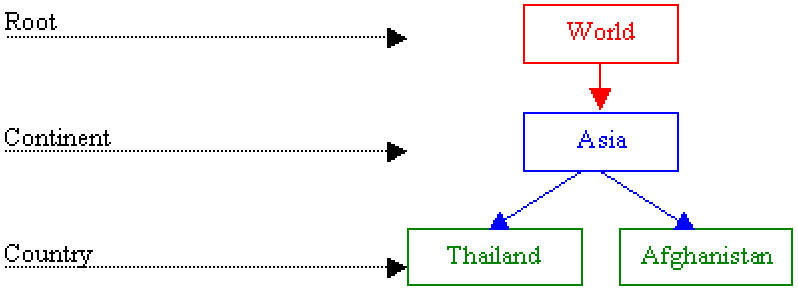
- Getting Started
- "How Do I..." Section
- Creating Images
- Making Photographs Look Better
- Using the Painting Tools
- Selecting Colors
- Working with Selections
- Working with Layers
- Using Vector Layers and Tools
- Using Adjustment Layers

Figure 58: A Tree List With Various Roots

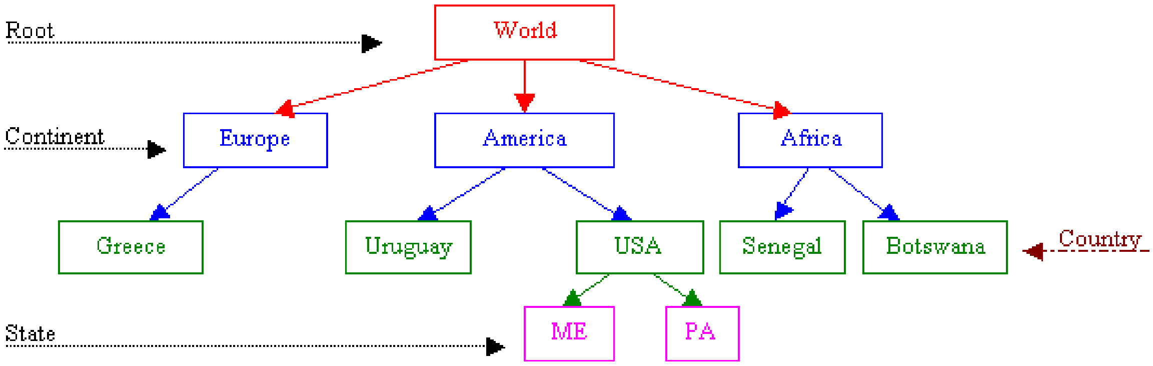
Each item, including the root, that belongs to the tree is referred to as a node. An item that belongs to, or depends on, another can also be called a leaf. In the following charts, the down arrow means, "has the following child or children":



A tree list is not limited to a one-to-one correspondence. Not only can an item have more than one dependency, but also a child can make the tree stop at any category. Categories of a tree list are organized by levels. The most used trees have one parent. Here is an example representing the world and some countries:



The children of a parent are recognized by their belonging to the same level but can have different behaviors; for example, while one child might have another child (or other children), an item on the same level does not necessarily abide by a set rule. Everything usually depends on the tree designer.



The concept of a tree list is implemented in the MFC library by the **CTreeCtrl**. To create a tree list on a dialog box or a form, at design time, on the Controls toolbox, click the Tree Control button and click the desired area on a dialog box or a form:

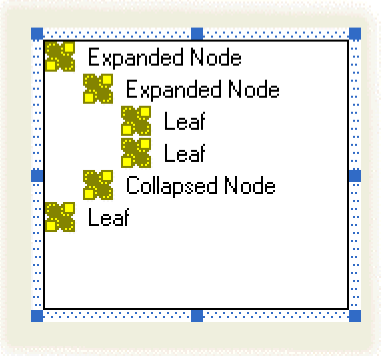


Figure 59: A Newly added Tree Control

Alternatively, to programmatically create a tree list, declare a variable or a pointer to **CTreeCtrl**. To initialize the control, call its Create() method. Here is an example:

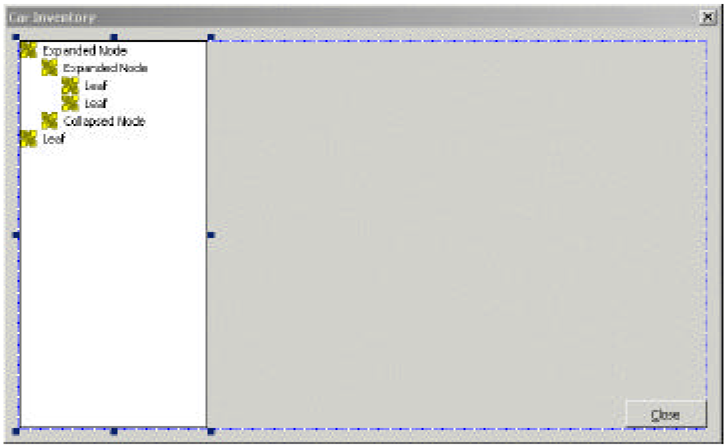
```
private:
	CTreeCtrl *TreeSoft;
};
```

```
CControlsDlg::CControlsDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CControlsDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CControlsDlg)
		// NOTE: the ClassWizard will add member initialization here
	//}}AFX_DATA_INIT
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

	TreeSoft = new CTreeCtrl;
}

CControlsDlg::~CControlsDlg()
{
	delete TreeSoft;
}

BOOL CControlsDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	TreeSoft->Create(WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
			CRect(10, 10, 240, 280), this, 0x1221);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

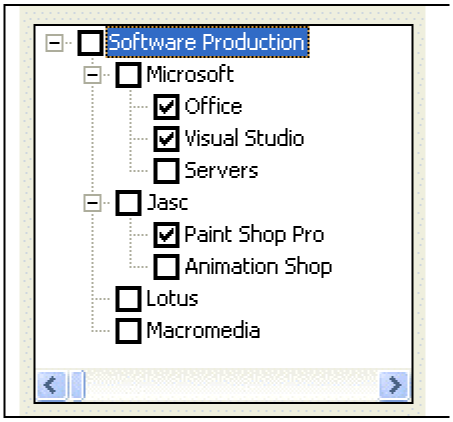
Practical Learning: Creating a Tree List

1. Create a new Dialog Based MFC Application named CarInventory2 without the About Box and set the Dialog Title to Car Inventory

2. Resize the dialog box to 430 x 230
3. On the dialog box, delete the TODO line and the OK button
4. Change the caption of the Cancel button to Close
5. From the Controls toolbox, click the Tree Control button and click on the left area of the dialog box
6. Change its ID to IDC_CAR_TREE
7. Set its location to 7, 7 and its dimensions to 114 x 216



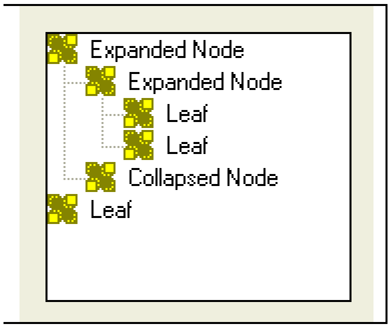
21.1.2 Tree List Properties



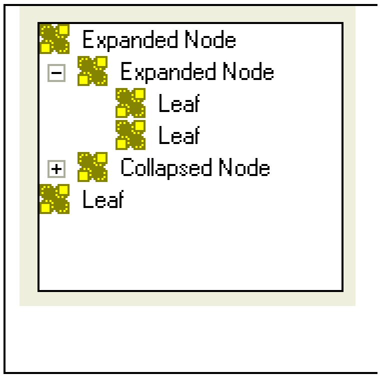
As mentioned already, a tree list is meant to display items in a list driven by a root and followed by one or more leaves. The items are mainly made of text. Optionally, to display a check box on the left side of the text, set the Check Boxes property to True or add the **TVS_CHECKBOXES** style (if you are using MSVC 6 and you had added the Tree Control button to the form or dialog box, open the resource file as text and manually add this style because it may not be available on the Properties window).

To guide the user with the tree items, the control uses tool tips. If you will need access to the information stored in tool tips, set the **Info Tip** property to True or add the **TVS_INFOTIP** style. If you do not want to display tool tips, set the **Tool Tips** property to False or create it with the **TVS_NOTOOLTIPS** style.

When a node has children or leaves, to show this, you may want to display lines connecting these relationships. To do this at design time, set the **Has Lines** proeprty to True:

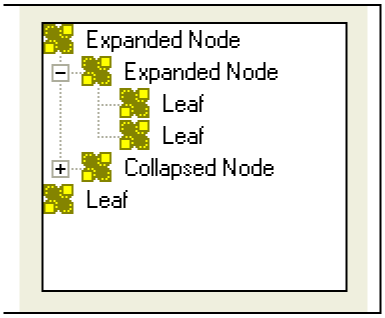


If you are programmatically creating the control and you want to display lines among related nodes, add the **TVS_HASLINES** style.

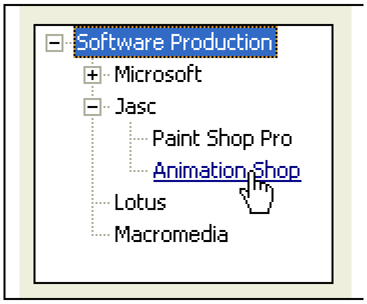


A node that has dependent children can display or hides them. To display its leaves, a node must be expanded. To hide its leaves, a node must collapse. These operations must be obvious to the user but something should indicate whether a node has dependent or not. This can be illustrated by a button that accompany such a node. To add these buttons to the control, at design time, set the **Has Buttons** property to True. This is equivalent to dynamically creating a tree list with the **TVS_HASBUTTONS** style:

Unless you have a reason to do otherwise, it is usually a good idea to combine both the **Has Buttons** (or **TVS_HASBUTTONS**) and the **Has Lines** (or **TVS_HASLINES**) styles:



To show which item is the root, or which items play the roles of roots, of the tree list, you can display a line from the root(s) to the child(ren). To do this, at design time, set the **Lines At Root** property to True or add the **TVS_LINESATROOT** style. The line from the root(s) to to the child(ren) can display only if the control has the **Has Lines** property set to True or the **TVS_HASLINES** style.



When using the list, the user typically selects an item by clicking it. If you want the mouse cursor to turn into a pointer finger and to underline the item when the mouse is over the node, set the **Track Select** property to True or create the control with the **TVS_TRACKSELECT** style.

Once the mouse pointer is on top of the desired item, the user can click to select it.

When the user clicks another control or another application, the node that was selected would lose its selection as the tree control would have lost focus. If you want the tree to always show the selected item even if the control loses focus, set its **Show Selection**

Always property to True or create the control with the **TVS_SHOWSELALWAYS** style.

Besides selecting an item, when a node has children, to expand it, the user can double-click the node or click its button if available. Simply selecting the node does not expand it. If you want the selected node to automatically expand without the user having to double-click or click its button, set its **Single Expand** property to True or create it with the **TVS_SINGLEEXPAND** style.

When the items of a tree control display or when the nodes expand, they may span beyond the allocated rectangle of the control. When this happens, a vertical and/or a horizontal scroll bars may automatically display. This is because, by default, the **Scroll** property is set to True. If you do not want any scroll bar, set the Scroll property to False or create the control with the **TVS_NOSCROLL** style.

Another operation the user can perform on a node consists of changing its text. If you want to allow this, set the control's Edit Labels to True or add the **TVS_EDITLABELS** style to it.

A user can be allowed to add items to the list by drag-n-drop operations. If you want to prevent this, set the Disable Drag Drop property to True or create the control with the **TVS_DISABLEDRAGDROP** style.

Practical Learning: Configuring a Tree Control

1. On the dialog box, click the tree control to select it. On the Properties window set the **Has Buttons**, the **Has Lines** and the **Lines At Root** properties to True
2. Also set the **Client Edge** and the **Modal Frame** properties to True
3. Add a **Control Variable** to the tree object and name it **m_CarTree**
4. Save All

21.1.3 Tree Controls Methods

After adding or creating a tree control, you may want to fill it with the necessary items. Each node of the control is an **HTREEITEM**. To create a new node, call the **CTreeCtrl::InsertItem()** method. It comes in various versions. One of them is:

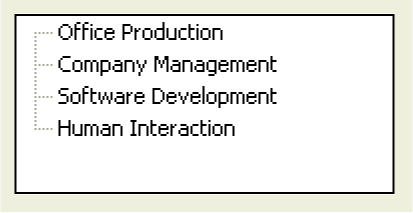
```
HTREEITEM InsertItem( LPCTSTR lpszItem, HTREEITEM hParent = TVI_ROOT,
HTREEITEM hInsertAfter = TVI_LAST );
```

The easiest way to add an item consists of calling the **InsertItem()** method with a null-terminated string as argument because this is the only required argument of this version. Here is an example:

```
BOOL CControlsDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	TreeSoft->Create(WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP |
                              TVS_HASLINES | TVS_HASBUTTONS | TVS_LINESATROOT |
                               TVS_SINGLEEXPAND | TVS_SHOWSELALWAYS |
                              TVS_TRACKSELECT,
                              CRect(10, 10, 200, 100), this, 0x1221);

        TreeSoft->InsertItem("Office Production");

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```

In this case, the item would appear as the root. You can add as many nodes like that and each would appear as a root:

```
BOOL CControlsDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        SetIcon(m_hIcon, TRUE);                 // Set big icon
        SetIcon(m_hIcon, FALSE);        // Set small icon

        // TODO: Add extra initialization here
        TreeSoft->Create(WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP |
                              TVS_HASLINES | TVS_HASBUTTONS | TVS_LINESATROOT |
                              TVS_SINGLEEXPAND | TVS_SHOWSELALWAYS |
                              TVS_TRACKSELECT,
                              CRect(10, 10, 200, 100), this, 0x1221);

        TreeSoft->InsertItem("Office Production");//, TVI_ROOT);
        TreeSoft->InsertItem("Company Management");
        TreeSoft->InsertItem("Software Development");
        TreeSoft->InsertItem("Human Interaction");

        return TRUE;  // return TRUE  unless you set the focus to a control
}
```



Office Production
Company Management
Software Development
Human Interaction

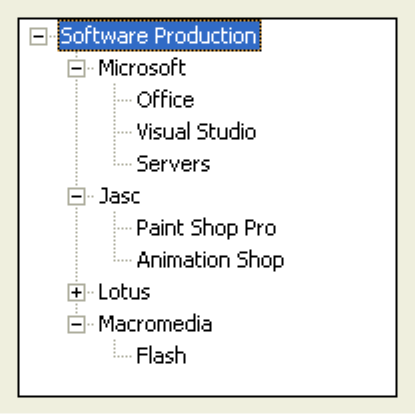
Figure 60: A Tree List With All Items As Roots

When calling this version of the **InsertItem()** method, if you do not pass the second argument, the node is created as root. This is because the root item has an HTREEITEM value of **TVI_ROOT**, which is passed as default. You can also pass the second argument as NULL, which would produce the same effect.

The **InsertItem()** method returns an **HTREEITEM** value. You can use this value as a parent to a leaf item. This is done by passing it as the second argument when creating a leaf. Here is an example:

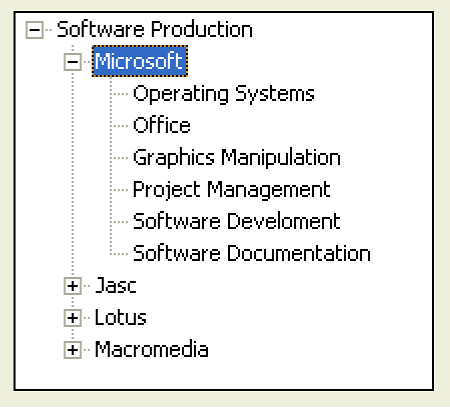
```
BOOL CControlsDlg::OnInitDialog()
{
        CDialog::OnInitDialog();
```

```
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	TreeSoft->Create(WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP |
		         TVS_HASLINES | TVS_HASBUTTONS | TVS_LINESATROOT |
		          TVS_SINGLEEXPAND | TVS_SHOWSELALWAYS |
		          TVS_TRACKSELECT,
		         CRect(10, 10, 200, 200), this, 0x1221);

	HTREEITEM hTree, hCompany;

	hTree = TreeSoft->InsertItem("Software Production", TVI_ROOT);
	hCompany = TreeSoft->InsertItem("Microsoft", hTree);
	TreeSoft->InsertItem("Office", hCompany);
	TreeSoft->InsertItem("Visual Studio", hCompany);
	TreeSoft->InsertItem("Servers", hCompany);
	hCompany = TreeSoft->InsertItem("Jasc", hTree);
	TreeSoft->InsertItem("Paint Shop Pro", hCompany);
	TreeSoft->InsertItem("Animation Shop", hCompany);
	hCompany = TreeSoft->InsertItem("Lotus", hTree);
	TreeSoft->InsertItem("Notes", hCompany);
	TreeSoft->InsertItem("Smart Office", hCompany);
	hCompany = TreeSoft->InsertItem("Macromedia", hTree);
	TreeSoft->InsertItem("Flash", hCompany);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```



When using the **InsertItem()** method as we have done so far, the items are added in the order of their appearance. Besides creating new nodes, the **InsertItem()** methods also allows you to control the order in which to insert the new item. The new leaf can be added as the first or the last child of a node. To do this, pass a third argument to this version of the **InsertItem()** method and give it the **TVI_FIRST** to be the first child or **TVI_LAST** to be the last child of the current parent node. Consider the following example:

```
BOOL CControlsDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	TreeSoft->Create(WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP |
	                 TVS_HASLINES | TVS_HASBUTTONS | TVS_LINESATROOT |
	                 TVS_SINGLEEXPAND | TVS_SHOWSELALWAYS |
	                 TVS_TRACKSELECT,
	                 CRect(10, 10, 220, 200), this, 0x1221);

	HTREEITEM hTree, hCompany;

	hTree = TreeSoft->InsertItem("Software Production", TVI_ROOT);
	hCompany = TreeSoft->InsertItem("Microsoft", hTree);
	TreeSoft->InsertItem("Office", hCompany);
	TreeSoft->InsertItem("Graphics Manipulation", hCompany, TVI_LAST);
	TreeSoft->InsertItem("Project Management", hCompany);
	TreeSoft->InsertItem("Software Develoment", hCompany);
	TreeSoft->InsertItem("Operating Systems", hCompany, TVI_FIRST);
	TreeSoft->InsertItem("Software Documentation", hCompany);

	hCompany = TreeSoft->InsertItem("Jasc", hTree);
	TreeSoft->InsertItem("Paint Shop Pro", hCompany);
	TreeSoft->InsertItem("Animation Shop", hCompany);
	hCompany = TreeSoft->InsertItem("Lotus", hTree);
	TreeSoft->InsertItem("Notes", hCompany);
	TreeSoft->InsertItem("Smart Office", hCompany);
	hCompany = TreeSoft->InsertItem("Macromedia", hTree);
	TreeSoft->InsertItem("Flash", hCompany);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

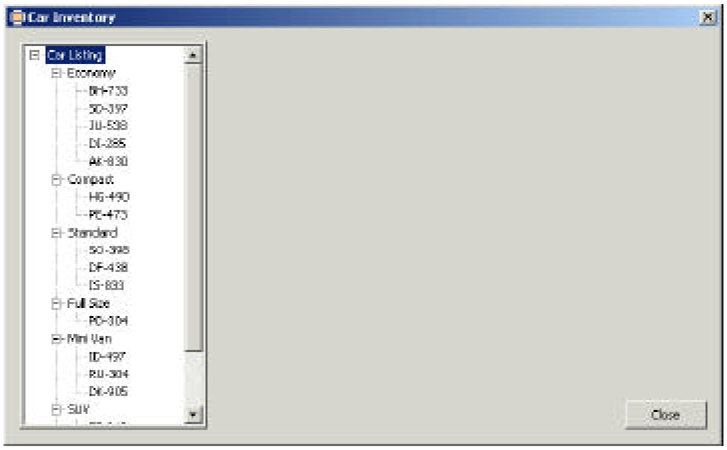


Practical Learning: Configuring a Tree Control

1. To create the items for the tree control, access the OnInitDialog() event of the CCarInventory2Dlg class and type the following:

```
BOOL CCarInventory2Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	HTREEITEM hItem, hCar;

	hItem = m_CarTree.InsertItem("Car Listing", TVI_ROOT);

	hCar  = m_CarTree.InsertItem("Economy", hItem);
	m_CarTree.InsertItem("BH-733", hCar);
	m_CarTree.InsertItem("SD-397", hCar);
	m_CarTree.InsertItem("JU-538", hCar);
	m_CarTree.InsertItem("DI-285", hCar);
	m_CarTree.InsertItem("AK-830", hCar);

	hCar  = m_CarTree.InsertItem("Compact", hItem);
	m_CarTree.InsertItem("HG-490", hCar);
	m_CarTree.InsertItem("PE-473", hCar);
	hCar  = m_CarTree.InsertItem("Standard", hItem);
	m_CarTree.InsertItem("SO-398", hCar);
	m_CarTree.InsertItem("DF-438", hCar);
	m_CarTree.InsertItem("IS-833", hCar);

	hCar  = m_CarTree.InsertItem("Full Size", hItem);
	m_CarTree.InsertItem("PD-304",  hCar);

	hCar  = m_CarTree.InsertItem("Mini Van", hItem);
	m_CarTree.InsertItem("ID-497", hCar);
	m_CarTree.InsertItem("RU-304", hCar);
	m_CarTree.InsertItem("DK-905", hCar);

	hCar  = m_CarTree.InsertItem("SUV", hItem);
	m_CarTree.InsertItem("FE-948", hCar);
	m_CarTree.InsertItem("AD-940", hCar);

	hCar  = m_CarTree.InsertItem("Truck", hItem);
	m_CarTree.InsertItem("HD-394", hCar);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

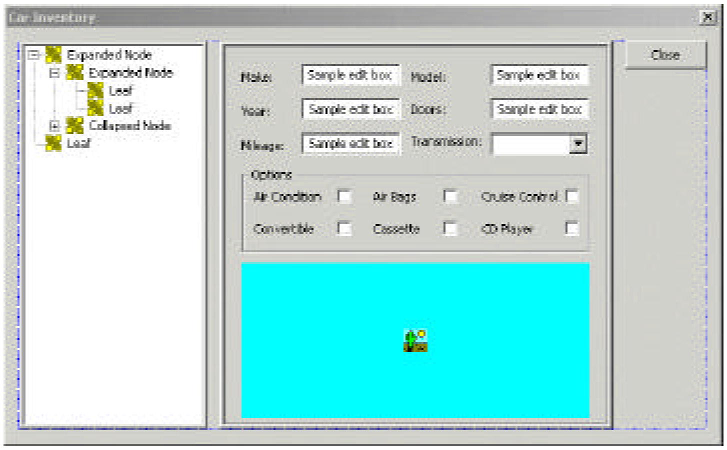
2. Execute the program to test the tree control



3. Close it and return to MSVC

4. Import the cars from the Cars folder that accompany this book and change their IDs accordingly to their file name. An example would be IDB_CIVIC for the civic.mdb

5. Design the rest of the dialog box with additional controls as follows:



Control	ID	Caption	Other Properties
Picture			238 x 216
Static Text		Make:	
Edit Box	IDC_MAKE		
Static Text		Model:	
Edit Box	IDC_MODEL		
Static Text		Year:	
Edit Box	IDC_YEAR		Align Text: Right Number: True
Static Text		Doors:	
Edit Box	IDC_DOORS		Align Text: Right Number: True
Static Text		Mileage:	
Edit Box	IDC_MILEAGE		Align Text: Right Number: True
Static Text		Transmission:	
Combo Box	IDC_TRANSMISSION		Type: Drop List Data: Automatic;Manual
Group Box		Options	
Check Box	IDC_AC	Air Condition	Left Text: True
Check Box	IDC_AIR_BAGS	Air Bags	Left Text: True
Check Box	IDC_CRUISE_CONTROL	Cruise Control	Left Text: True
Check Box	IDC_CONVERTIBLE	Convertible	Left Text: True
Check Box	IDC_CASSETTE	Cassette	Left Text: True
Check Box	IDC_CD_PLAYER	CD Player	Left Text: True
Picture	IDC_PICTURE		Type: Bitmap Image: IDB_ESCORT Center Image: True Dimensions: 210x 86

6. Add a variable to each non-static control as follows:

ID	Value Variable		Control Variable	
	Type	Name	Type	Name
IDC_MAKE	CString	m_Make		
IDC_MODEL	CString	m_Model		

IDC_YEAR	int	m_Year		
IDC_DOORS	int	m_Doors		
IDC_MILEAGE	long	m_Mileage		
IDC_TRANSMISSION	int	m_Transmission		
IDC_AC	BOOL	m _AC		
IDC_AIR_BAGS	BOOL	m _AirBags		
IDC_CRUISE_CONTROL	BOOL	m _CruiseControl		
IDC_CONVERTIBLE	BOOL	m _Convertible		
IDC_CASSETTE	BOOL	m _Cassette		
IDC_CD_PLAYER	BOOL	m _CDPlayer		
IDC_PICTURE	BOOL		CStatic	m_Picture

7. Save All

21.1.4 Tree Control Messages

Most messages of the tree controls are notification messages that are sent to its parent window. For example, the **NM_CLICK** message is sent to the dialog box or the form, that acts as the parent, that the tree control has been clicked. If the click was done with the right mouse button, the **NM_RCLICK** message is sent instead. In the same way, if the user double-clicks the control an **NM_DBLCLK** message is sent. If the user double-clicked with the right mouse button, the **NM_CDBLCLK** message is sent.

As mentioned already, the user has the ability to expand a node that has at least one child. When the user initiates an action that would expand a node, the tree control sends a **TVN_ITEMEXPANDING**. After the item has expanded, the control sends the **TVN_ITEMEXPANDED** message.

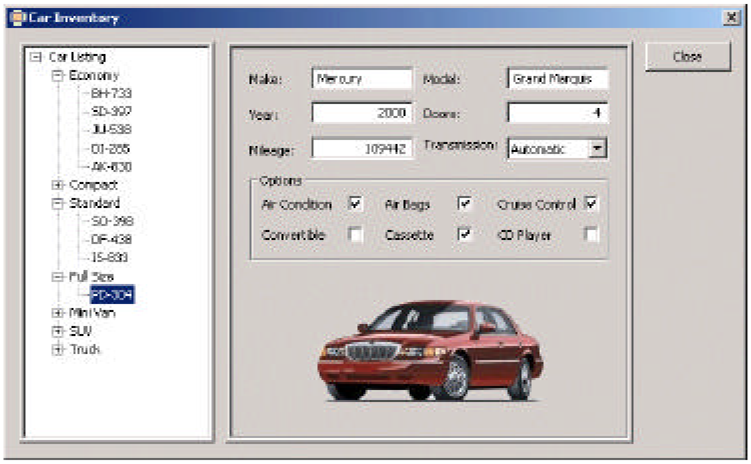
Practical Learning: Using Tree Control Messages

1. On the dialog box, right-click the tree control and click Add Event Handler…
2. In the Message Type list box, click **TVN_SELCHANGED**. In the Class List, make sure CCarInventory2Dlg is selected
3. In the Function Handle Name, change the name to **OnCarSelectedChange** and click Add And Edit
4. Implement the OnPaint and the OnCarSelectedChange events as follows:

```
void CCarInventory2Dlg::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // Change the car picture based on the selected tag
    CTreeCtrl *pTree = reinterpret_cast<CTreeCtrl
*>(GetDlgItem(IDC_CAR_TREE));
    HTREEITEM hTree = pTree->GetSelectedItem();

    CString ItemSelected;
    CBitmap Bmp;

    ItemSelected = pTree->GetItemText(hTree);

    if( ItemSelected == "BH-733" )
    {
            Bmp.LoadBitmap(IDB_FOCUS);
            m_Picture.SetBitmap(Bmp);
    }
    else if( ItemSelected == "HD-394" )
    {
            Bmp.LoadBitmap(IDB_EMPTY);
            m_Picture.SetBitmap(NULL);
    }
    else if( ItemSelected == "SD-397" )
    {
            Bmp.LoadBitmap(IDB_EMPTY);
            m_Picture.SetBitmap(NULL);
    }
    else if( ItemSelected == "PD-304" )
    {
            Bmp.LoadBitmap(IDB_GRANDMARQUIS);
            m_Picture.SetBitmap(Bmp);
    }
    else
    {
            Bmp.LoadBitmap(IDB_EMPTY);
            m_Picture.SetBitmap(NULL);
    }

    if (IsIconic())
    {
            SendMessage(WM_ICONERASEBKGND,
reinterpret_cast<WPARAM>(dc.GetSafeHdc()), 0);

            // Center icon in client rectangle
            int cxIcon = GetSystemMetrics(SM_CXICON);
            int cyIcon = GetSystemMetrics(SM_CYICON);
            CRect rect;
            GetClientRect(&rect);
            int x = (rect.Width() - cxIcon + 1) / 2;
            int y = (rect.Height() - cyIcon + 1) / 2;

            // Draw the icon
            dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
            CDialog::OnPaint();
    }
}

// The system calls this function to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CCarInventory2Dlg::OnQueryDragIcon()
{
    return static_cast<HCURSOR>(m_hIcon);
}

void CCarInventory2Dlg::OnCarSelectedChange(NMHDR *pNMHDR, LRESULT
*pResult)
{
    LPNMTREEVIEW pNMTreeView = reinterpret_cast<LPNMTREEVIEW>(pNMHDR);
    // TODO: Add your control notification handler code here
    *pResult = 0;

    CTreeCtrl *pTree = reinterpret_cast<CTreeCtrl
*>(GetDlgItem(IDC_CAR_TREE));
    HTREEITEM hTree = pTree->GetSelectedItem();

    CString ItemSelected;
    CBitmap Bmp;

    ItemSelected = pTree->GetItemText(hTree);

    if( ItemSelected == "BH-733" )
    {
            m_Make.Format("%s", "Ford");
            m_Model.Format("%s", "Focus");
            m_Year = 2000;
            m_Doors = 4;
            m_Mileage = 72804;
            m_Transmission  = 1;
            m_AC           = TRUE;
            m_AirBags       = TRUE;
            m_CruiseControl = FALSE;
            m_Convertible   = FALSE;
            m_Cassette      = FALSE;
            m_CDPlayer                  = FALSE;
    }
    else if( ItemSelected == "HD-394" )
    {
            m_Make.Format("%s", "Ford");
            m_Model.Format("%s", "F150");
            m_Year = 2002;
            m_Doors = 4;
            m_Mileage = 28845;
            m_Transmission = 0;
            m_AC           = TRUE;
            m_AirBags       = TRUE;
            m_CruiseControl = FALSE;
            m_Convertible   = FALSE;
            m_Cassette      = TRUE;
            m_CDPlayer             = FALSE;
    }
    else if( ItemSelected == "SD-397" )
    {
            m_Make.Format("%s", "Daewoo");
            m_Model.Format("%s", "Lanos");
            m_Year = 2000;
            m_Doors = 4;
            m_Mileage = 94508;
            m_Transmission  = 1;
            m_AC           = TRUE;
            m_AirBags       = TRUE;
            m_CruiseControl = FALSE;
            m_Convertible   = FALSE;
            m_Cassette      = TRUE;
            m_CDPlayer             = FALSE;
    }
    else if( ItemSelected == "PD-304" )
    {
            m_Make.Format("%s", "Mercury");
```

```
                m_Model.Format("%s", "Grand Marquis");
                m_Year           = 2000;
                m_Doors          = 4;
                m_Mileage        = 109442;
                m_Transmission  = 0;
                m_AC             = TRUE;
                m_AirBags        = TRUE;
                m_CruiseControl = TRUE;
                m_Convertible    = FALSE;
                m_Cassette       = TRUE;
                m_CDPlayer       = FALSE;
        }
        else
        {
                m_Make.Format("%s", "");
                m_Model.Format("%s", "");
                m_Year = 0;
                m_Doors = 0;
                m_Mileage = 0;
                m_Transmission  = 0;
                m_AC           = FALSE;
                m_AirBags        = FALSE;
                m_CruiseControl = FALSE;
                m_Convertible   = FALSE;
                m_Cassette      = FALSE;
                m_CDPlayer                  = FALSE;
        }

        CRect RectPicture;

        m_Picture.GetWindowRect(&RectPicture);
        ScreenToClient(&RectPicture);
        InvalidateRect(&RectPicture);

        UpdateData(FALSE);
}
```

5. Test the application



Make: Mercury Model: Grand Marquis
Year: 2000 Doors: 4
Mileage: 109442 Transmission: Automatic

Air Condition ☑ Air Bags ☑ Cruise Control ☑
Convertible □ Cassette ☑ CD Player □

6. Close it and return to MSVC

21.1.5 Tree Control With Bitmaps or Icons

Bitmaps can be used to enhanced the display of items on a tree control. Each tree item can be configured to display or not to display a small picture on its left. To do this, you can declare a CImageList variable and add pictures to it. Once the image list is ready, you can call the **CTreeCtrl::SetImageList()** method. Its syntax is:

```
CImageList* SetImageList(CImageList * pImageList, int nImageListType);
```

The first argument, *pImageList*, is a pointer to a CImageList variable. The *nImageListType* argument specifies the type of image list that will be used. The possible values are:

Value Type	Description
LVSIL_NORMAL	The image list is made of large icons
LVSIL_SMALL	The image list is made of small icons
LVSIL_STATE	The image list is made of state images

To specify the pictures used for a tree item, call one of the following versions of the **CTreeCtrl:: InsertItem()** methods:

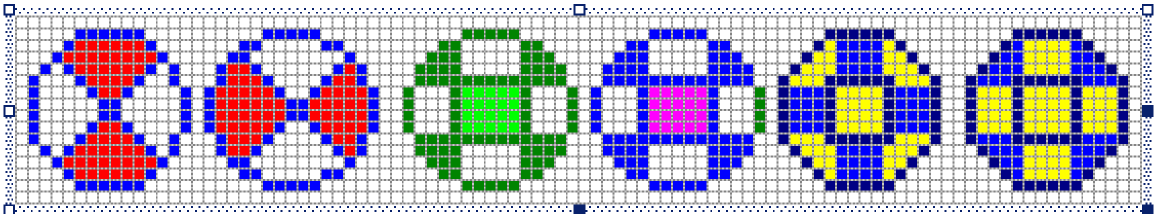
```
HTREEITEM InsertItem(UINT nMask, LPCTSTR lpszItem, int nImage,
                     int nSelectedImage, UINT nState,
                     UINT nStateMask, LPARAM lParam,
                     HTREEITEM hParent, HTREEITEM hInsertAfter);
HTREEITEM InsertItem(LPCTSTR lpszItem, int nImage, int nSelectedImage,
                     HTREEITEM hParent = TVI_ROOT,
                     HTREEITEM hInsertAfter = TVI_LAST);
```

The *nMask* argument specifies the type of value used to set on the list item. As seen already, the *lpszItem* is the text that will be displayed for the current item.

The value of *nImage* is the index of the image used for the item being inserted from the image list. The *nSelectedImage* value is the index of the image that will display when the inserted item is selected or has focus.

Practical Learning: Using an Image List on a Tree Control

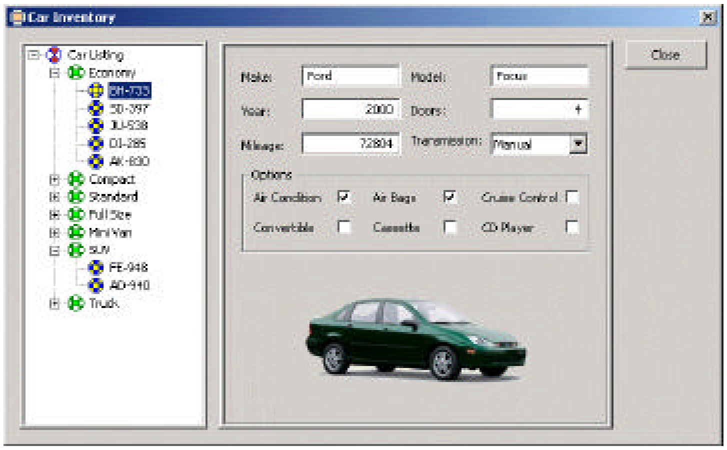
1. Display the Add Resource dialog box and double-click Bitmap
2. On the Properties, change its ID to IDB_IMGTREE
3. Design is as follows:



4. In the header file of the dialog, declare a private **CImageList** variable and name it **ImgList**
5. To assign the pictures to the tree items, change the OnInitDialog() event of the dialog box as follows:

```
BOOL CCarInventory2Dlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Set the icon for this dialog.  The framework does this automatically
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	// TODO: Add extra initialization here
	HTREEITEM hItem, hCar;

	m_Picture.SetBitmap(NULL);
	ImgList.Create(IDB_IMGTREE, 16, 6, RGB(0, 0, 0));
	m_CarTree.SetImageList(&ImgList, TVSIL_NORMAL);

	hItem = m_CarTree.InsertItem("Car Listing", 0, 1, TVI_ROOT);
	hCar  = m_CarTree.InsertItem("Economy", 2, 3, hItem);
	m_CarTree.InsertItem("BH-733", 4, 5, hCar); //
	m_CarTree.InsertItem("SD-397", 4, 5, hCar); //
	m_CarTree.InsertItem("JU-538", 4, 5, hCar);
	m_CarTree.InsertItem("DI-285", 4, 5, hCar);
	m_CarTree.InsertItem("AK-830", 4, 5, hCar);

	hCar  = m_CarTree.InsertItem("Compact", 2, 3, hItem);
	m_CarTree.InsertItem("HG-490", 4, 5, hCar);
	m_CarTree.InsertItem("PE-473", 4, 5, hCar);
	hCar  = m_CarTree.InsertItem("Standard", 2, 3, hItem);
	m_CarTree.InsertItem("SO-398", 4, 5, hCar);
	m_CarTree.InsertItem("DF-438", 4, 5, hCar);
	m_CarTree.InsertItem("IS-833", 4, 5, hCar);

	hCar  = m_CarTree.InsertItem("Full Size", 2, 3, hItem);
	m_CarTree.InsertItem("PD-304", 4, 5,  hCar);

	hCar  = m_CarTree.InsertItem("Mini Van", 2, 3, hItem);
	m_CarTree.InsertItem("ID-497", 4, 5, hCar);
	m_CarTree.InsertItem("RU-304", 4, 5, hCar);
	m_CarTree.InsertItem("DK-905", 4, 5, hCar);

	hCar  = m_CarTree.InsertItem("SUV", 2, 3, hItem);
	m_CarTree.InsertItem("FE-948", 4, 5, hCar);
	m_CarTree.InsertItem("AD-940", 4, 5, hCar);

	hCar  = m_CarTree.InsertItem("Truck", 2, 3, hItem);
	m_CarTree.InsertItem("HD-394", 4, 5, hCar);

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

6. Test the application



Make: Ford
Model: Focus
Year: 2000
Doors: 4
Mileage: 72804
Transmission: Manual

Air Condition ☑ Air Bags ☑ Cruise Control □
Convertible □ Cassette □ CD Player □

7. Close it and return to MSVC

21.2 The Tree View

21.2.1 Overview

A tree view is a frame-based application whose view uses the characteristics of a tree control. To create such an application, you can work from scratch and derive a class from **CTreeView**. Alternatively, you can use the MFC Application wizard to create the application.

As a tree control is based on the **CTreeCtrl** class, a tree view application uses the Document/View Architecture to implement its behavior rather than using a dialog box. This class simply implements the tree control on a application and provides all the functionality of the CTreeCtrl class.

21.2.2 Tree View Implementation

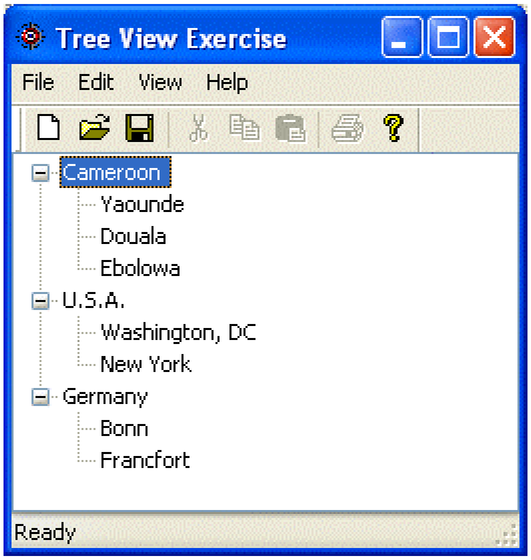
To create a tree view application, you can create a form-based application, place a tree control on it, and do anything reviewed above. Alternatively, you can derive your own class from CTreeView. Instead, the MFC provides a faster and better means of creating a tree view, using the MFC Application wizard for which you would select CTreeView as the Base Class.

We saw that a tree control can be programmatically created by calling the CTreeCtrl::Create() method. This allows you to define the style used on the control. When using the tree view, you can specify the initial style in the **PreCreateWindow()** event of the view class. Here is an example:

```
BOOL CExoTV2View::PreCreateWindow(CREATESTRUCT& cs)
{
	cs.style |= TVS_HASLINES | TVS_HASBUTTONS |
			TVS_LINESATROOT | TVS_EDITLABELS;

	return CTreeView::PreCreateWindow(cs);
}
```

After creating the view, you should initiale it. This is usually done in the **OnInitialUpdate()** event of the view class. In order to take advantage of the features of the list control, you should obtain a reference to the **CTreeCtrl** class that controls the tree view. Once you have accessed that reference, you can do anything you would do with a **CTreeCtrl** variable or pointer. Here is an example of creating a feww items of a tree view:

```
void CExerciseView::OnInitialUpdate()
{
        CTreeView::OnInitialUpdate();

        CTreeCtrl&  trCtrl = GetTreeCtrl();

        HTREEITEM hItem;

        hItem = trCtrl.InsertItem( "Cameroon", 0, 2 );
        trCtrl.InsertItem( "Yaounde", 1, 3, hItem );
        trCtrl.InsertItem( "Douala", 1, 3, hItem );
        trCtrl.InsertItem( "Ebolowa", 1, 3, hItem );
        hItem = trCtrl.InsertItem( "U.S.A.", 0, 2 );
        trCtrl.InsertItem( "Washington, DC", 1, 3, hItem );
        trCtrl.InsertItem( "New York", 1, 3, hItem );
        hItem = trCtrl.InsertItem( "Germany", 0, 2 );
        trCtrl.InsertItem( "Bonn", 1, 3, hItem );
        trCtrl.InsertItem( "Francfort", 1, 3, hItem );
}
```



```
Cameroon
    Yaounde
    Douala
    Ebolowa
U.S.A.
    Washington, DC
    New York
Germany
    Bonn
    Francfort
```

21.3 The List Control

21.3.1 Overview

A list control consists of using one of four views to display a list of items. The list is typically equipped with icons that indicate what view is displaying. There are four views used to display items:

Icons: The control displays a list of items using icons with a 32x32 pixels size of icons. This is the preferred view when the main idea consists of giving an overview of the items

Small Icons: Like the other next two views, it uses 16x16 pixel icons to display a simplified list of the items. Once more, no detail is provided about the items of this list. The list is organized in disparate columns with some on top of others. If the list is supposed to be sorted, the alphabetical arrangement is organized from left to right.

List: This list, using small icons, is also organized in columns; this time, the columns are arranged so that the first column gets filled before starting the second. If the list is sorted, the sorting is arranged in a top-down manner.

Report: This view displays arranged columns of items and provides as many details as the list developer had arranged it.

21.3.2 List Control Creation

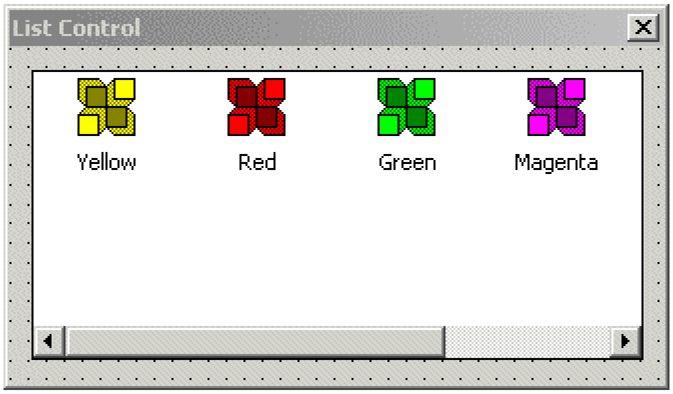
A list control is implemented in the MFC library by the **CListCtrl** class. At design time, to create a list control, on the Controls toolbox, click the List Control button and click the desired area on a dialog box or a form. Normally, you should expand its dimensions beyond the default assigned because a list control is usually used to display its items on a wide rectangle.

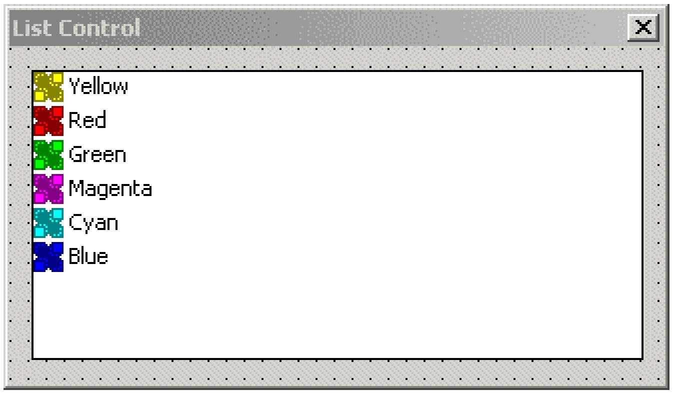
To programmatically create a list control, declare a variable or a pointer to **CListCtrl**. To initialiaze the control, call its **Create()** method. Here is an example:

```
BOOL CPictureDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	CListCtrl *lstCtrl = new CListCtrl;

	lstCtrl->Create(WS_CHILD | WS_VISIBLE,
	                 CRect(10, 10, 320, 280), this, 0x285);

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

As mentioned already, a list control can display its items in one of four views. To specify the desired view at design time, on the Properties window, select a value from the **View** combo box. The default value is **Icon**. The possible values are:

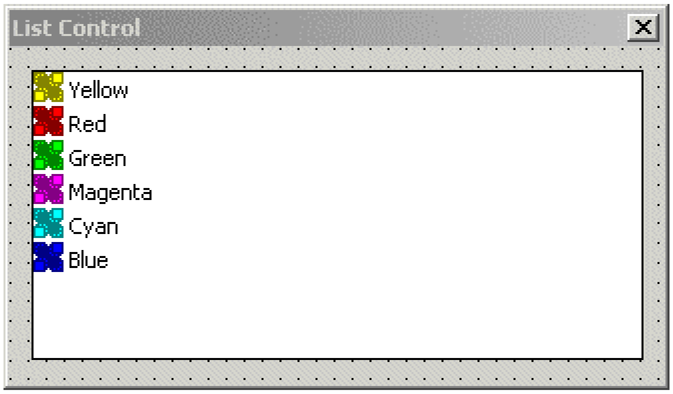
Icon: To get this value when programmatically creating the control, add the **LVS_ICON** style:

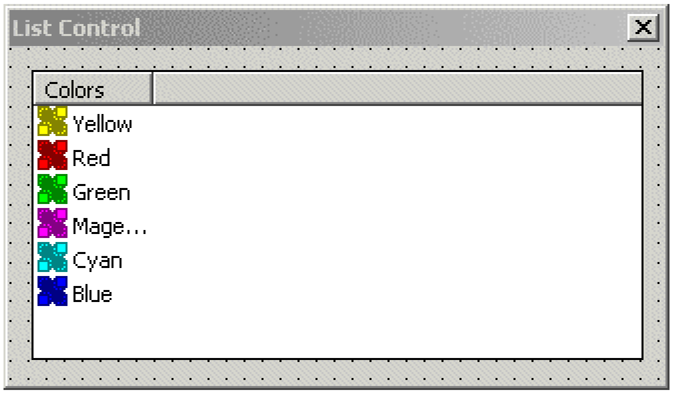




Small Icon: This is the same as adding the **LVS_SMALLICON** style to a dynamic list control

List: You can get the same result when creating the control with code by adding the **LVS_LIST** style:





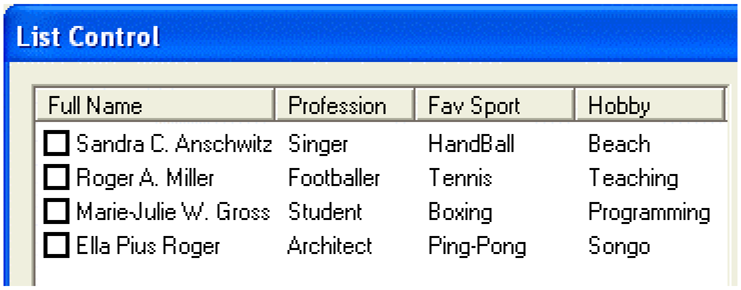
Report: This view displays the items in explicit columns. It is the same as adding the **LVS_REPORT** style

Besides the regular styles, the Win32 library provides extended styles for a list control. To apply an extended style, call the **CListCtrl::SetExtendedStyle()** method. Its syntax is:

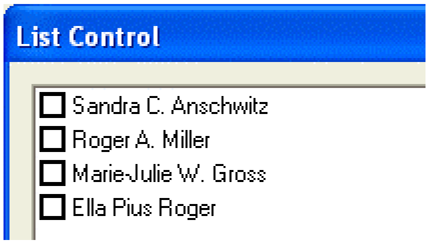
```
DWORD SetExtendedStyle(DWORD dwNewStyle);
```

When calling this method, pass the desired extended style or a combination of these styles as argument. Some of the values are:

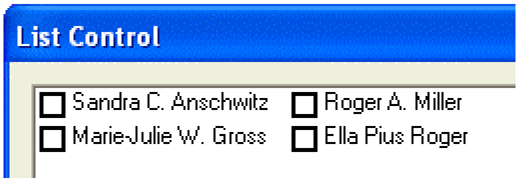
LVS_EX_CHECKBOXES : The items of the control will display a check box on their left side:



Full Name	Profession	Fav Sport	Hobby
☐ Sandra C. Anschwitz	Singer	HandBall	Beach
☐ Roger A. Miller	Footballer	Tennis	Teaching
☐ Marie-Julie W. Gross	Student	Boxing	Programming
☐ Ella Pius Roger	Architect	Ping-Pong	Songo

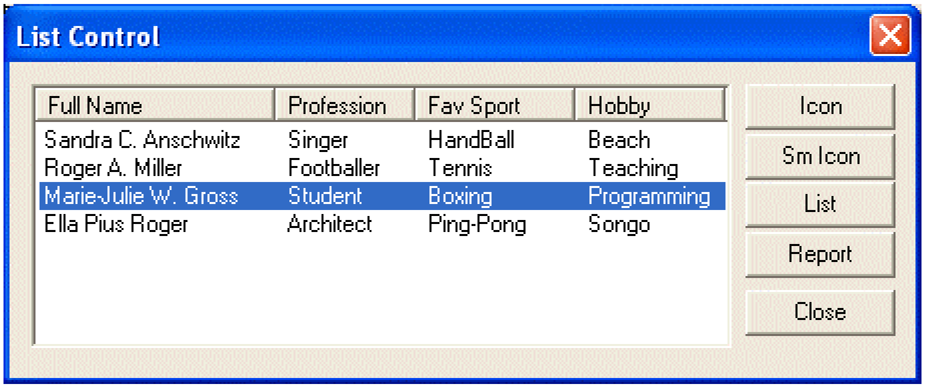


☐ Sandra C. Anschwitz
☐ Roger A. Miller
☐ Marie-Julie W. Gross
☐ Ella Pius Roger



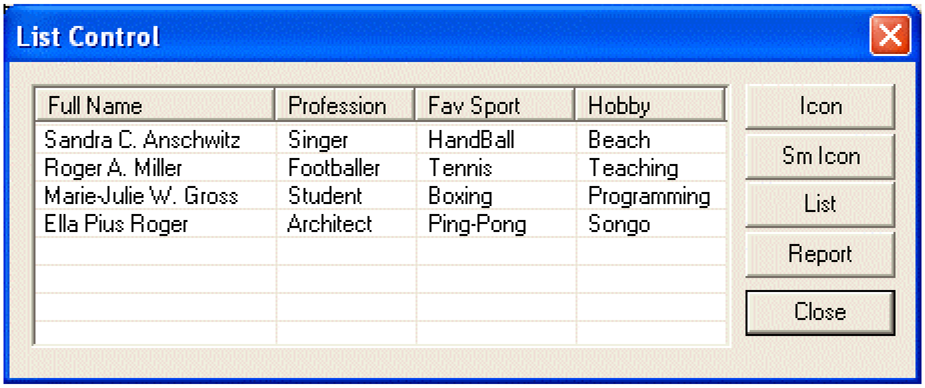
☐ Sandra C. Anschwitz ☐ Roger A. Miller
☐ Marie-Julie W. Gross ☐ Ella Pius Roger

LVS_EX_FULLROWSELECT: This style allows the whole row of a Report view to be selected instead of just the item:



Full Name	Profession	Fav Sport	Hobby
Sandra C. Anschwitz	Singer	HandBall	Beach
Roger A. Miller	Footballer	Tennis	Teaching
Marie-Julie W. Gross	Student	Boxing	Programming
Ella Pius Roger	Architect	Ping-Pong	Songo

LVS_EX_GRIDLINES: The control's items in Report view will display with horizontal grid lines that separate items and vertical grid lines that separate columns items or categories:

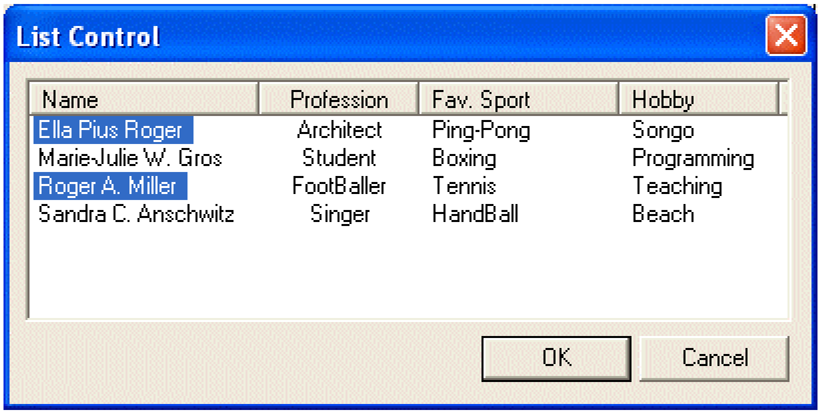


Full Name	Profession	Fav Sport	Hobby
Sandra C. Anschwitz	Singer	HandBall	Beach
Roger A. Miller	Footballer	Tennis	Teaching
Marie-Julie W. Gross	Student	Boxing	Programming
Ella Pius Roger	Architect	Ping-Pong	Songo

LVS_EX_TRACKSELECT: When this style is set, if the user positions the mouse on an item for a few seconds without clicking, the item would be automatically selected.

The items of a list control can display only within the control, if there are too many of them or the total width of the items is larger than the control can display, it would be equipped with either a vertical scroll bar, a horizontal scroll bar, or both. If you want to prevent scroll bars from displaying even if the list's items go beyond the allocated rectangle, set the **No Scroll** property to True or create the control with the **LVS_NOSCROLL** style.

Once the list has been created, the user can select an item by clicking it. To select more than one item, the user can press and hold either Ctrl for random selection or Shift for range selection. Here is an example of a random selection:



Name	Profession	Fav. Sport	Hobby
Ella Pius Roger	Architect	Ping-Pong	Songo
Marie-Julie W. Gros	Student	Boxing	Programming
Roger A. Miller	FootBaller	Tennis	Teaching
Sandra C. Anschwitz	Singer	HandBall	Beach

If you do not want the user to be able to select more than one item at a time, set the **Single Selection** property to True or create the control with the **LVS_SINGLESEL** style.

Any item that is selected is highlighted. When the user clicks another control or another application, you can decide whether you want the item(s) selected to still show its (their) selection. State. This characteristic is controlled at design time by the **Show Selection Always** property. By default, it is set to False, meaning that when the control looses focus or its parent application is deactivated, an item that is selected would not show it. Otherwise, you can set this property to True to indicate the selected item even when the control is not active. This property is available through the **LVS_SHOWSELALWAYS** style.

When creating the list, its items are sorted in alphabetical order using the items text as reference. Even if you add items later on, they are inserted in the appropriate order. This sorting feature is controlled at design time by the **Sort** combo box box. By default, the items of a list control are sorted in alphabetical order using the **Ascending** value or the **LVS_SORTASCENDING** style. If you want items to be sorted in reverse alphabetical order, set this property to Descending or create the control with the **LVS_SORTDESCENDING** style.

21.3.3 Items of a List Control

After visually adding or dynamically creating a list control, the next action you probably would take is to populate the control with the desired items. This can be taken care of by calling the **CListCtrl::InsertItem()** method. One of its syntaxes is as follows:

```
int InsertItem(const LVITEM* pItem );
```

This version requires an **LVITEM** pointer as argument. The **LVITEM** structure is defined as follows:

```
typedef struct _LVITEM {
    UINT mask;
    int iItem;
    int iSubItem;
    UINT state;
    UINT stateMask;
    LPTSTR pszText;
    int cchTextMax;
    int iImage;
    LPARAM lParam;
#if (_WIN32_IE >= 0x0300)
    int iIndent;
#endif
} LVITEM, FAR *LPLVITEM;
```

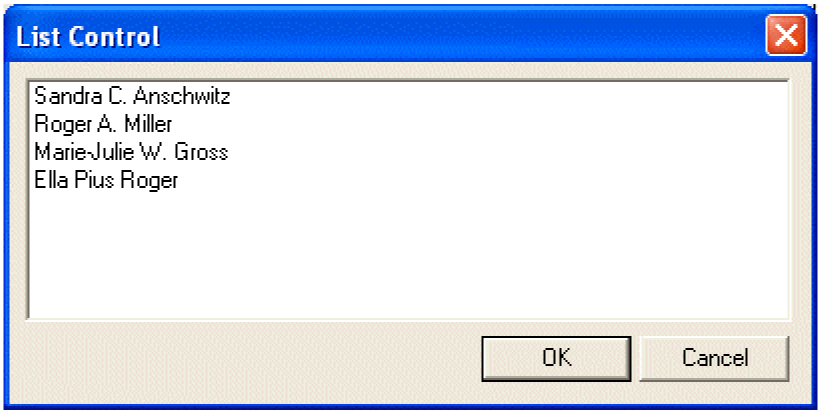
The *mask* member variable is used to specify the types of values you want to set for the current item.

The value of *iItem* specifies the index of the item that is being changed. The first item would have an index of 0. The second would be 1, etc. The *iSubItem* member variable is the index of the sub item for the current value. If the current item will be the leader, the *iSubItem* is stored in a 0-based array. If it is a sub item, then it is stored in a 1-based array.

The *pszText* member variable is the string that will display as the item. You can specify the length of this text by assigning a value to the *cchTextMask* variable.

After initializing the **LVITEM** variable, you can pass it to the **InsertItem()** method to add it as a new item to the list. Here is an example that creates items and displays as a List view:

```
BOOL COthersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	LVITEM lvItem;
	lvItem.mask = LVIF_TEXT;
	lvItem.iItem = 0;
	lvItem.iSubItem = 0;
	lvItem.pszText = "Sandra C. Anschwitz";
	m_List.InsertItem(&lvItem);

	lvItem.mask = LVIF_TEXT;
	lvItem.iItem = 1;
	lvItem.iSubItem = 0;
	lvItem.pszText = "Roger A. Miller";
	m_List.InsertItem(&lvItem);

	lvItem.mask = LVIF_TEXT;
	lvItem.iItem = 2;
	lvItem.iSubItem = 0;
	lvItem.pszText = "Marie-Julie W. Gross";
	m_List.InsertItem(&lvItem);

	lvItem.mask = LVIF_TEXT;
	lvItem.iItem = 3;
	lvItem.iSubItem = 0;
	lvItem.pszText = "Ella Pius Roger";
	m_List.InsertItem(&lvItem);

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



Sandra C. Anschwitz
Roger A. Miller
Marie-Julie W. Gross
Ella Pius Roger

The *state* member variable of the **LVITEM** structure specifies what to do with the new item. For example, once the item has been added, you may want to prepare it for deletion prior to a cut-and-paste operation, in which case you would give it a value of **LVIS_CUT**. If the item is involved in a drag-and-drop operation, you can assign it a state value of **LVIS_DROPHILIGHTED**. To give focus to the item, set its state value to **LVIS_FOCUSED**. An item with an **LVIS_SELECTED** state value will be selected.

Besides the above version of the **CListCtrl::InsertItem()** method, the **CListCtrl** class provides this other version:

int InsertItem(int *nItem*, LPCTSTR *lpszItem*);

This is a good simplification of the earlier version. The *nItem* argument is the index of the new item to de added. Like the **LVITEM**::*iItem* member variable, the value of this argument is 0 if the item will be the leader. The *lpszItem* value is the string that will be used to lead the current item.

21.3.4 The Report View

Whether you use the first or the second version, the **InsertItem()** method allows you to create the item that will display for the Icon, the Small Icon, or the List views of the control. If you plan to display the list in Report view (or to allow the user to transition from various views) and you want to provide more information for each item, you must “create” a report of information for each item.

Among the possible views available for a list control, one of them can display columns. This view is called the report view. This view is not required for a list view but it is the only one that provides more detailed information about the items of the list. If you plan to display that view on your list control, then you should create columns. (Alternatively, you can omit creating columns and instead provide headers of columns separately, which can be done using the **CHeaderCtrl** class. Otherwise, the list control provides the means of creating columns for its report view.)

To create the column(s) of a list control, you can use the **CListCtrl::InsertColumn()** method. One of its syntaxes is:

```
int InsertColumn(int nCol, const LVCOLUMN* pColumn);
```

The *nCol* argument is the index of the column that this call will create.

The *pColumn* argument is an **LVCOLUMN** pointer. This structure is defined as follows:

```
typedef struct _LVCOLUMN {
   UINT mask;
   int fmt;
   int cx;
   LPTSTR pszText;
   int cchTextMax;
   int iSubItem;
#if (_WIN32_IE >= 0x0300)
   int iImage;
   int iOrder;
#endif
} LVCOLUMN, FAR *LPLVCOLUMN;
```

The *mask* member variable is used to specify what attribute of the column you want to define with this **LVCOLUMN** variable.

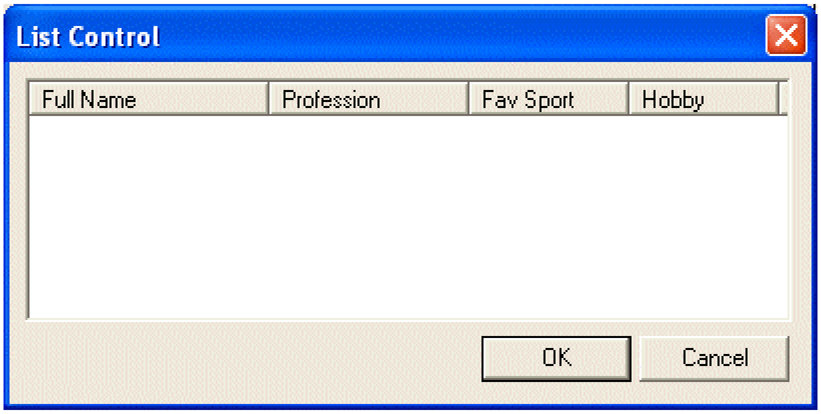
The *fmt* member variable formats the text of the column. For example, it can be used to align the text of the column to the left (the default) (**LVCFMT_LEFT**), the center (**LVCFMT_CENTER**), or the right (**LVCFMT_RIGHT**). If you do not set a value for this member variable, the text will be aligned to the left. If you plan to set a value for this variable, then add the **LVCF_FMT** value for the *mask* member variable.

The *cx* variable is used to specify the width occupied by the text of the column. If you do not set a value for this property, the column would initially appear so narrow its text would not display. Therefore, unless you have a good reason to omit it, you should always specify the value of this variable. If you plan to set a value for this property, then add the **LVCF_WIDTH** value to the *mask* member variable.

The *pszText* is the string that will appear as the text of the column. Just like all the other member variables, this one is not required but, besides the rectangle that limits the column header, this member is probably the most important characteristic of a column because it informs the user as to what this column is used for. The string of this variable can be provided as a null-terminated value. Like all other strings used in an MFC application, this string can also be a value of a String Table item. It can also be retrieved from an array of strings. To set a value for this member variable, add the **LVCF_TEXT** value to the *mask* variable. The length of this string can be specified by assigning a value to the *cchTextMax* member variable.

After initializing the **LVCOLUMN** variable, pass it as the **CListCtrl::InsertColumn()** second argument. Here is an example:

```
BOOL COthersDlg::OnInitDialog()
{
        CDialog::OnInitDialog();

        // TODO: Add extra initialization here
        LVCOLUMN lvColumn;
        int nCol;

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
        lvColumn.fmt  = LVCFMT_LEFT;
        lvColumn.cx   = 120;
        lvColumn.pszText = "Full Name";
        nCol = m_List.InsertColumn(0, &lvColumn);

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
        lvColumn.fmt  = LVCFMT_LEFT;
        lvColumn.cx   = 100;
        lvColumn.pszText = "Profession";
        m_List.InsertColumn(1, &lvColumn);

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
        lvColumn.fmt  = LVCFMT_LEFT;
        lvColumn.cx   = 80;
        lvColumn.pszText = "Fav Sport";
        m_List.InsertColumn(2, &lvColumn);

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
        lvColumn.fmt  = LVCFMT_LEFT;
        lvColumn.cx   = 75;
        lvColumn.pszText = "Hobby";
        m_List.InsertColumn(3, &lvColumn);


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



The *iOrder* member variable is used to identify the column addressed by the **LVCOLUMN** variable.

Besides, or instead of, the above version of the **CListCtrl::InsertColumn()** method, you can use the following version to create columns for the control:

```
int InsertColumn(int nCol, LPCTSTR lpszColumnHeading, int nFormat = LVCFMT_LEFT,
                 int nWidth = -1, int nSubItem = -1);
```

This version simplifies the first a little bit. The *nCol* argument is the index of the column that will be configured. The second argument, *lpszColumnHeading*, is the string that will be displayed on the column header. It follows the same rules as the **LVCOLUMN**::*pszText* member variable.

The optional *nFormat* argument is used to specify the horizontal alignment of the *lpszColumnHeading* text. It can be set to **LVCFMT_LEFT** for left alignment (the default), **LVCFMT_CENTER** for center alignment, or **LVCFMT_RIGHT** for right alignment. If you do not specify this argument, the text would be aligned to the left. The *nWidth* ardument is used to set the width of the column header in pixels. If you do not want to specify this argument, pass it at –1. The *nSubItem* is used to set the index of the sub item used on the current column.

With the columns configured, you must provide a string that will be displayed under a particular column header for an item. To do this, you must first specify which item will use the information you are going to add. The **InsertColumn()** method returns an integer that is the index of the new item. You can use this returned value to identify the column whose information you are adding. Then to specify a string for each column of the current item, call the **CListCtrl::SetItemText()** method. Its syntax is:

```
BOOL SetItemText(int nItem, int nSubItem, LPTSTR lpszText);
```

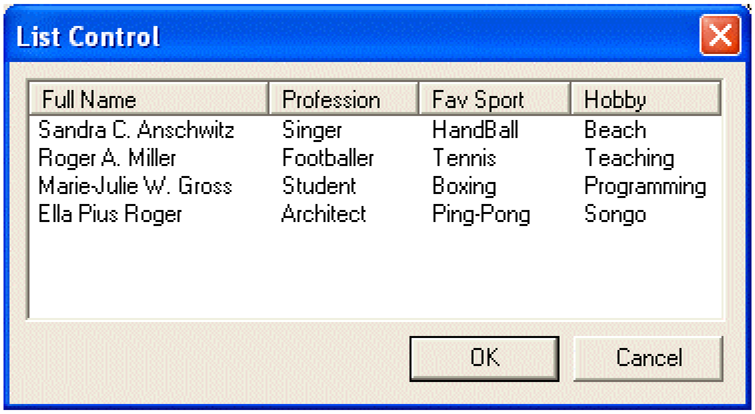
The *nItem* argument is the index of the column whose information you are adding. It can be the return value of a previously called **InsertColumn().** The pieces of information for each item are stored in a 0-based array. The index of the current sub item is specified using the *nSubItem* argument. The *lpszText* is the actual text that will display under the column for the current item.

Here is an example:

```
BOOL COthersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
```

```
// TODO: Add extra initialization here
LVCOLUMN lvColumn;

lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
lvColumn.fmt  = LVCFMT_LEFT;
lvColumn.cx   = 120;
lvColumn.pszText = "Full Name";
m_List.InsertColumn(0, &lvColumn);

lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
lvColumn.fmt  = LVCFMT_LEFT;
lvColumn.cx   = 75;
lvColumn.pszText = "Profession";
m_List.InsertColumn(1, &lvColumn);

lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
lvColumn.fmt  = LVCFMT_LEFT;
lvColumn.cx   = 80;
lvColumn.pszText = "Fav Sport";
m_List.InsertColumn(2, &lvColumn);

lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
lvColumn.fmt  = LVCFMT_LEFT;
lvColumn.cx   = 75;
lvColumn.pszText = "Hobby";
m_List.InsertColumn(3, &lvColumn);

LVITEM lvItem;
int nItem;

lvItem.mask = LVIF_TEXT;
lvItem.iItem = 0;
lvItem.iSubItem = 0;
lvItem.pszText = "Sandra C. Anschwitz";
nItem = m_List.InsertItem(&lvItem);

m_List.SetItemText(nItem, 1, "Singer");
m_List.SetItemText(nItem, 2, "HandBall");
m_List.SetItemText(nItem, 3, "Beach");

lvItem.mask = LVIF_TEXT;
lvItem.iItem = 1;
lvItem.iSubItem = 0;
lvItem.pszText = "Roger A. Miller";
nItem = m_List.InsertItem(&lvItem);

m_List.SetItemText(nItem, 1, "Footballer");
m_List.SetItemText(nItem, 2, "Tennis");
m_List.SetItemText(nItem, 3, "Teaching");

lvItem.mask = LVIF_TEXT;
lvItem.iItem = 2;
lvItem.iSubItem = 0;
lvItem.pszText = "Marie-Julie W. Gross";
nItem = m_List.InsertItem(&lvItem);

m_List.SetItemText(nItem, 1, "Student");
m_List.SetItemText(nItem, 2, "Boxing");
```

```
        m_List.SetItemText(nItem, 3, "Programming");

        lvItem.mask = LVIF_TEXT;
        lvItem.iItem = 3;
        lvItem.iSubItem = 0;
        lvItem.pszText = "Ella Pius Roger";
        nItem = m_List.InsertItem(&lvItem);

        m_List.SetItemText(nItem, 1, "Architect");
        m_List.SetItemText(nItem, 2, "Ping-Pong");
        m_List.SetItemText(nItem, 3, "Songo");

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



Full Name	Profession	Fav Sport	Hobby
Sandra C. Anschwitz	Singer	HandBall	Beach
Roger A. Miller	Footballer	Tennis	Teaching
Marie-Julie W. Gross	Student	Boxing	Programming
Ella Pius Roger	Architect	Ping-Pong	Songo

21.3.5 Views Transition

You can create a list control that displays its items in a single view or you can allow the user to change from one view to another. As mentioned already, at design time or when programmatically creating the list control, you can set the initial view using either the View combo box to select a view's value or by adding one of the view styles. If you want to display only that initial view, you can stop there. Otherwise, you can provide a means of changing views.

Because the view displayed on a list control is part of its style, in order to programmatically change its view mode, you can first retrieve the control's style using the **GetWindowLong()** function. The **GetWindowLong()** function only retrieves the current style of the control. You may need to check it first before changing it. This can be done by ANDing the value of the **GetWindowLong()** function with the **LVS_TYPEMASK** constant. After checking the view of the control, you can then change its style by calling the **SetWindowLong()** function. Here is an example:

```
void COthersDlg::OnIconBtn()
{
        // TODO: Add your control notification handler code here
        LONG mListStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE);
        mListStyle &= ~LVS_TYPEMASK;
        mListStyle |= LVS_ICON;
        SetWindowLong(m_List.m_hWnd, GWL_STYLE, mListStyle);
}
```

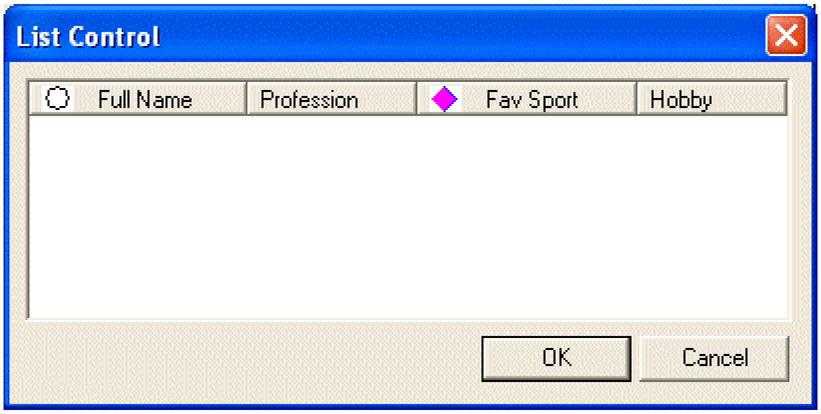
```
void COthersDlg::OnSmallIconBtn()
{
	// TODO: Add your control notification handler code here
	LONG mListStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE);
	mListStyle &= ~LVS_TYPEMASK;
	mListStyle |= LVS_SMALLICON;
	SetWindowLong(m_List.m_hWnd, GWL_STYLE, mListStyle);
}

void COthersDlg::OnListBtn()
{
	// TODO: Add your control notification handler code here
	LONG mListStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE);
	mListStyle &= ~LVS_TYPEMASK;
	mListStyle |= LVS_LIST;
	SetWindowLong(m_List.m_hWnd, GWL_STYLE, mListStyle);
}

void COthersDlg::OnReportBtn()
{
	// TODO: Add your control notification handler code here
	LONG mListStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE);
	mListStyle &= ~LVS_TYPEMASK;
	mListStyle |= LVS_REPORT;
	SetWindowLong(m_List.m_hWnd, GWL_STYLE, mListStyle);
}
```

21.3.6 List Control and Icons

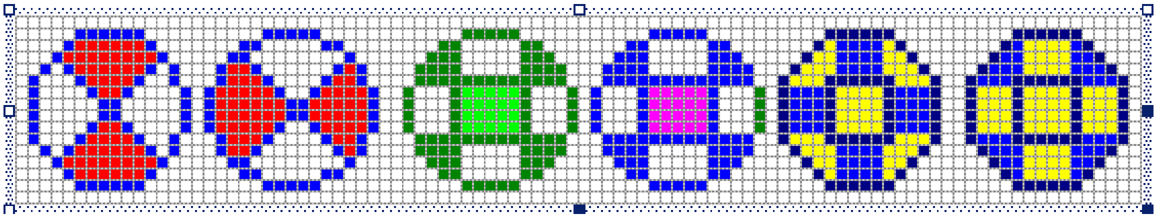
A list control can be configured to display pictures that accompany either the columns, the list items, or both. If you want to display a bitmap on the column, you should declare and initialize a **CImageList** variable. Then call the **CListCtrl::SetImageList()** method and pass it as argument. If you plan to do this, and if you are using the first version of the CListCtrl::InsertColumn() method that takes an LVCOLUMN pointer as argument, then add the **LVCF_IMAGE** value to the *mask* variable and add the **LVCFMT_IMAGE** value to the *fmt* variable. To specify the image that will display on the column header, assign its index to the value of the *iImage* variable. Here is an example:

```
BOOL COthersDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// TODO: Add extra initialization here
	LVCOLUMN lvColumn;

	CImageList *ImgHeaders = new CImageList;

	ImgHeaders->Create(16, 16, ILC_MASK, 1, 1);
	ImgHeaders->Add(AfxGetApp()->LoadIcon(IDI_UP));
	ImgHeaders->Add(AfxGetApp()->LoadIcon(IDI_LOSANGE));

	m_List.SetImageList(ImgHeaders, LVSIL_SMALL);

	lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH | LVCF_IMAGE;
	lvColumn.fmt  = LVCFMT_LEFT | LVCFMT_IMAGE;
	lvColumn.cx   = 120;
	lvColumn.pszText = "Full Name";
```

```
        lvColumn.iImage = 0;
        m_List.InsertColumn(0, &lvColumn);

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
        lvColumn.fmt  = LVCFMT_LEFT;
        lvColumn.cx   = 100;
        lvColumn.pszText = "Profession";
        m_List.InsertColumn(1, &lvColumn);

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH | LVCF_IMAGE;
        lvColumn.fmt  = LVCFMT_LEFT | LVCFMT_IMAGE;
        lvColumn.iImage = 1;
        lvColumn.cx   = 80;
        lvColumn.pszText = "Fav Sport";
        m_List.InsertColumn(2, &lvColumn);

        lvColumn.mask = LVCF_FMT | LVCF_TEXT | LVCF_WIDTH;
        lvColumn.fmt  = LVCFMT_LEFT;
        lvColumn.cx   = 75;
        lvColumn.pszText = "Hobby";
        m_List.InsertColumn(3, &lvColumn);

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



To use bitmaps or icons on a list control’s items, you should first create bitmaps or icons. If you do not plan to use the Report view and you want to use bitmaps, you can create a long bitmap that will made of small pictures of the same size. Each picture will be used for each item. Normally, each picture should have a size of 16x16 or less. An example would be:



This bitmap is made of 6 pictures of the same dimensions. If you do not plan to use the Report view and you plan to use icons, create each icon with a 16x16 size.

If you plan to display the control's items in Report view and you want to use bitmaps, you can create a long bitmap that will made of small pictures of the same size and each picture will be used for each item. Each picture should have a size of 32x32.

If you plan to display the control's items in Report view and other views, if you want to use bitmaps, you should create two long bitmaps. One would be made of pictures that are 16x16 size. Such pictures would be used for the Small Icon, the List, and the Report views. You should also create the second bitmap made of pictures of 32x32 size. These pictures would be used for the List view.

After creating the bitmap and/or icons, you should declare a variable or a pointer to **CImageList** class and initialize it using the **CImageList::Create()** method. To make the image list available to the list control, call the **CListCtrl::SetImageList()** method. Its syntax is:

```
CImageList* SetImageList(CImageList* pImageList, int nImageList);
```

The *pImageList* argument is a **CImageList** variable or pointer previously initialized. The *nImageList* is a flag that specifies the type of image list used. It can have one of the following values:

Value	Description
LVSIL_NORMAL	The image list is made of large bitmap or icons, typically 32x32
LVSIL_SMALL	The image list is made of small bitmap or icons, typically 16x16
LVSIL_STATE	The image list is made of pictures that will be used as mask

To associate a picture with a list item, you can use one of the following versions of the **CListCtrl::InsertItem()** methods:

```
int InsertItem(int nItem, LPCTSTR lpszItem, int nImage );
int InsertItem(UINT nMask, int nItem, LPCTSTR lpszItem, UINT nState, UINT nStateMask,
               int nImage, LPARAM lParam );
```

The *nImage* argument is the index of the image used for the item.

The *nMask* argument is the same as the **LVITEM**::*mask* member variable introduced earlier. It is simply used to specify what information you need to specify on the item. The nState argument is the same as the **LVITEM**::*state* member variable introduced earlier and used to specify what to do with the item, whether to select it, give it focus, getting it ready for a cut-and-paste operation, or highlighted for a drag-and-drop operation. The *nStateMask* argument is used in conjunction with the *nState* argument. It is used to specify what exact type of information, defined by the *state* flag, that will be changed or retrieved on the item.

The *lParam* argument, as well as the *lParam* member variable of the **TVITEM** structure are used to perform a specific operation on the item, such as involving it in a sort algorithm or finding an item.

21.4 The List View

21.4.1 Overview

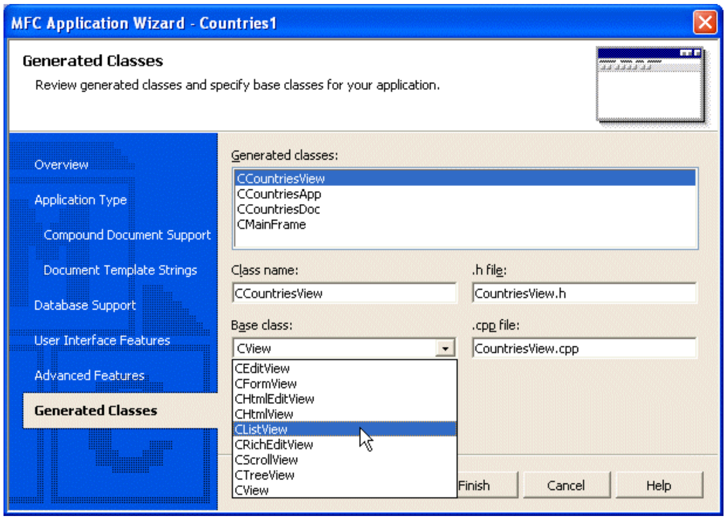
The list control we have used so far is a regular control added to a dialog box or a form. The Document/View Architecture of the MFC allows you to create a broader frame-based application whose view is an implementation of the list control. This is called the List View.

The list view application is a program based on the CListView class. There are various ways you can create such an application. You can work from scratch. In this case, you would derive a class from CListView. Alternatively, you can create a form-based application, then add a list control that uses part or its whole client area. Fortunately, Visual C++ provides a good starting point to create a list view application. This is done by using the MFC Application wizard.

To create a list view, you can start a new MFC Application and, on the Based Class, select CListView. If you open the header file of the view class, you would notice that it is based on CListView.

Practical Learning: Creating a List View-Based Application

1. Start a new MFC Application and name it Countries1
2. Set the Application Type to Single Document
3. Change the Main Frame Caption to Countries Listing
4. Remove the Printing And Print Preview option
5. Set the Base Class to CListView


MFC Application Wizard - Countries1

Generated Classes

Review generated classes and specify base classes for your application.

Generated classes:
CCountriesView
CCountriesApp
CCountriesDoc
CMainFrame

Class name: CCountriesView

.h file: CountriesView.h

Base class: CView

.cpp file: CountriesView.cpp

CEditView
CFormView
CHtmlEditView
CHtmlView
CListView
CRichEditView
CScrollView
CTreeView
CView

6. Click Finish

21.4.2 List View Implementation

The list view is simply a technique of implementing a list control on a frame-based application using the Document/View architecture. All of the functionality emanates from the **CListCtrl** class. Therefore, in order to do anything, you should obtain a reference to the underlying **CListCtrl** already available by deriving a class from **CListView**.

Creating a class based on **CListView**only makes the list control available. You must still initialize it, specify the necessary style(s) or extended style(s), and configure it as you see fit using the apprpriate events.

The best place to initialize the list view is probably the **CListView::OnInitialUpdate()** event. Here is an example:

```
void CExoLVView::OnInitialUpdate()
{
	CListView::OnInitialUpdate();

	CListCtrl& lCtrl = GetListCtrl();
}
```

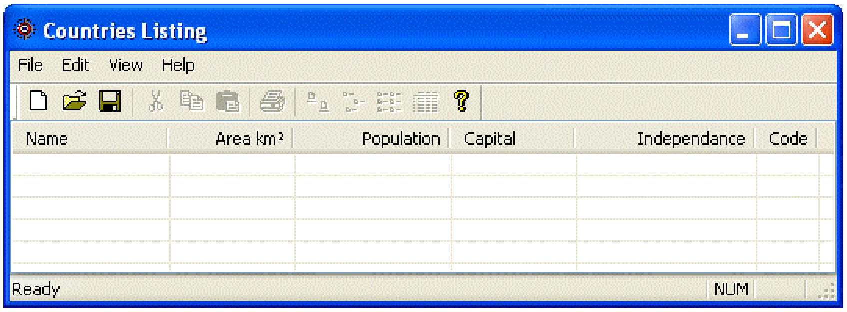
Once you have a reference to the **ClistCtrl** that controls the view, you can use everything else we covered earlier with regards to the list control. For example, to create columns, call any of the **CListCtrl::InsertColumn()** methods. To create the items that compose the list, call any of the **CListCtrl::InsertItem()** methods and use the **CListCtrl::SetItemText()** method to create sub items. The bitmaps or icons are used in the same way.

Practical Learning: Configuring a List View

1. To create columns for the list view and display it in report mode, change the OnInitialUpdate() event of the view class as follows:

```
void CCountriesView::OnInitialUpdate()
{
    CListView::OnInitialUpdate();

    // TODO: You may populate your ListView with items by directly accessing
    //  its list control through a call to GetListCtrl().
    CListCtrl& lCtrl = GetListCtrl();

    lCtrl.InsertColumn(0, "Name", LVCFMT_LEFT, 120);
    lCtrl.InsertColumn(1, "Area km\262", LVCFMT_CENTER, 80);
    lCtrl.InsertColumn(2, "Population", LVCFMT_LEFT, 120);
    lCtrl.InsertColumn(3, "Capital", LVCFMT_LEFT, 100);
    lCtrl.InsertColumn(4, "Code", LVCFMT_CENTER, 40);
    lCtrl.InsertColumn(5, "Independance", LVCFMT_CENTER, 88);

    lCtrl.SetExtendedStyle(LVS_EX_GRIDLINES | LVS_EX_FULLROWSELECT);

    ModifyStyle(0, LVS_REPORT);
}
```

2. Test the application:

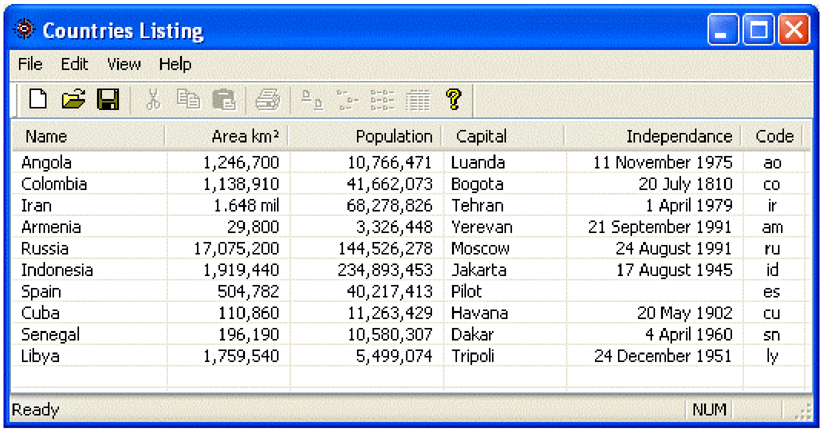


3. To populate the list, change the OnInitialUpdate() event of the view class as follows:

```
void CCountriesView::OnInitialUpdate()
{
    CListView::OnInitialUpdate();

    // TODO: You may populate your ListView with items by directly accessing
    //  its list control through a call to GetListCtrl().
    CListCtrl& lCtrl = GetListCtrl();

    lCtrl.InsertColumn(0, "Name", LVCFMT_LEFT, 100);
    lCtrl.InsertColumn(1, "Area km\262", LVCFMT_RIGHT, 80);
    lCtrl.InsertColumn(2, "Population", LVCFMT_RIGHT, 100);
    lCtrl.InsertColumn(3, "Capital", LVCFMT_LEFT, 80);
    lCtrl.InsertColumn(4, "Independance", LVCFMT_RIGHT, 115);
    lCtrl.InsertColumn(5, "Code", LVCFMT_CENTER, 40);

    lCtrl.SetExtendedStyle(LVS_EX_GRIDLINES | LVS_EX_FULLROWSELECT);

    ModifyStyle(0, LVS_REPORT);

    int nItem;

    nItem = lCtrl.InsertItem(0, "Libya");
    lCtrl.SetItemText(nItem, 1, "1,759,540"); lCtrl.SetItemText(nItem, 2, "5,499,074");
    lCtrl.SetItemText(nItem, 3, "Tripoli");   lCtrl.SetItemText(nItem, 4, "24 December 1951");
    lCtrl.SetItemText(nItem, 5, "ly");

    nItem = lCtrl.InsertItem(0, "Senegal");
    lCtrl.SetItemText(nItem, 1, "196,190"); lCtrl.SetItemText(nItem, 2, "10,580,307");
    lCtrl.SetItemText(nItem, 3, "Dakar");   lCtrl.SetItemText(nItem, 4, "4 April 1960");
    lCtrl.SetItemText(nItem, 5, "sn");

    nItem = lCtrl.InsertItem(0, "Cuba");
    lCtrl.SetItemText(nItem, 1, "110,860"); lCtrl.SetItemText(nItem, 2, "11,263,429");
    lCtrl.SetItemText(nItem, 3, "Havana");  lCtrl.SetItemText(nItem, 4, "20 May 1902");
    lCtrl.SetItemText(nItem, 5, "cu");

    nItem = lCtrl.InsertItem(0, "Spain");
    lCtrl.SetItemText(nItem, 1, "504,782"); lCtrl.SetItemText(nItem, 2, "40,217,413");
    lCtrl.SetItemText(nItem, 3, "Pilot");   lCtrl.SetItemText(nItem, 4, "");
    lCtrl.SetItemText(nItem, 5, "es");

    nItem = lCtrl.InsertItem(0, "Indonesia");
    lCtrl.SetItemText(nItem, 1, "1,919,440"); lCtrl.SetItemText(nItem, 2, "234,893,453");
    lCtrl.SetItemText(nItem, 3, "Jakarta");   lCtrl.SetItemText(nItem, 4, "17 August 1945");
    lCtrl.SetItemText(nItem, 5, "id");

    nItem = lCtrl.InsertItem(0, "Russia");
```

```
        lCtrl.SetItemText(nItem, 1, "17,075,200"); lCtrl.SetItemText(nItem, 2, "144,526,278");
        lCtrl.SetItemText(nItem, 3, "Moscow");     lCtrl.SetItemText(nItem, 4, "24 August 1991");
        lCtrl.SetItemText(nItem, 5, "ru");

        nItem = lCtrl.InsertItem(0, "Armenia");
        lCtrl.SetItemText(nItem, 1, "29,800");        lCtrl.SetItemText(nItem, 2, "3,326,448");
        lCtrl.SetItemText(nItem, 3, "Yerevan"); lCtrl.SetItemText(nItem, 4, "21 September 1991");
        lCtrl.SetItemText(nItem, 5, "am");

        nItem = lCtrl.InsertItem(0, "Iran");
        lCtrl.SetItemText(nItem, 1, "1.648 mil"); lCtrl.SetItemText(nItem, 2, "68,278,826");
        lCtrl.SetItemText(nItem, 3, "Tehran");    lCtrl.SetItemText(nItem, 4, "1 April 1979");
        lCtrl.SetItemText(nItem, 5, "ir");

        nItem = lCtrl.InsertItem(0, "Colombia");
        lCtrl.SetItemText(nItem, 1, "1,138,910"); lCtrl.SetItemText(nItem, 2, "41,662,073");
        lCtrl.SetItemText(nItem, 3, "Bogota");    lCtrl.SetItemText(nItem, 4, "20 July 1810");
        lCtrl.SetItemText(nItem, 5, "co");

        nItem = lCtrl.InsertItem(0, "Angola");
        lCtrl.SetItemText(nItem, 1, "1,246,700"); lCtrl.SetItemText(nItem, 2, "10,766,471");
        lCtrl.SetItemText(nItem, 3, "Luanda ");   lCtrl.SetItemText(nItem, 4, "11 November 1975");
        lCtrl.SetItemText(nItem, 5, "ao");
}
```

4. Test the application



Name	Area km²	Population	Capital	Independance	Code
Angola	1,246,700	10,766,471	Luanda	11 November 1975	ao
Colombia	1,138,910	41,662,073	Bogota	20 July 1810	co
Iran	1.648 mil	68,278,826	Tehran	1 April 1979	ir
Armenia	29,800	3,326,448	Yerevan	21 September 1991	am
Russia	17,075,200	144,526,278	Moscow	24 August 1991	ru
Indonesia	1,919,440	234,893,453	Jakarta	17 August 1945	id
Spain	504,782	40,217,413	Pilot		es
Cuba	110,860	11,263,429	Havana	20 May 1902	cu
Senegal	196,190	10,580,307	Dakar	4 April 1960	sn
Libya	1,759,540	5,499,074	Tripoli	24 December 1951	ly

5. Close it and return to MSVC

6. To prepare the view for display transition, in the header file of the view, declare two member functions as follows:

```
public:
        DWORD GetViewType(void);
        BOOL SetViewType(DWORD dwViewType);
};
```

7. In the source file of the view, implement the member functions as follows:

```
DWORD CCountriesView::GetViewType(void)
{
        return (GetStyle() & LVS_TYPEMASK);
}

BOOL CCountriesView::SetViewType(DWORD dwViewType)
{
	return (ModifyStyle(LVS_TYPEMASK, dwViewType & LVS_TYPEMASK));
}
```

8. From the Resource View tab, display the IDR_MAINFRAME menu. In the View category, add a separator in the first empty item
9. Click the next empty field and, in the Properties window, click the ID field. Then click its arrow to select **ID_VIEW_LARGEICON**. Set its Caption to **Large &Icons**
10. In the same way, set the ID of the next item to **ID_VIEW_SMALLICICON** and set its Caption to **S&mall Icons**
11. In the same way, set the ID of the next item to **ID_VIEW_LIST** and set its Caption to **&List**
12. In the same way, set the ID of the next item to **ID_VIEW_DETAILS** and set its Caption to **&Details**
13. To allow the user to change views, right-click the Large Icons menu item and click Add Event Handler…
14. In the Message Type list, accept the COMMAND message. In the Class List, click the view class. In the Function Handler Name, change the name to OnViewLargeIcon
15. Click Add And Edit and implement the message as follows:

```
void CCountriesView::OnViewLargeIcon()
{
	// TODO: Add your command handler code here
	if( GetViewType() != LVS_ICON )
		SetViewType(LVS_ICON);
}
```

16. In the same way, Add An Event Handler for the Small Icons menu item. Associate its COMMAND message with the view class. Change its name to **OnViewSmallIcon**
17. Implement it as follows:

```
void CCountriesView::OnViewSmallicon()
{
	// TODO: Add your command handler code here
	if( GetViewType() != LVS_SMALLICON )
		SetViewType(LVS_SMALLICON);
}
```

18. In the same way, Add An Event Handler for the List menu item. Associate its COMMAND message with the view class and implement it as follows:

```
void CCountriesView::OnViewList()
{
	// TODO: Add your command handler code here
	if( GetViewType() != LVS_LIST )
		SetViewType(LVS_LIST);
}
```

19. In the same way, Add An Event Handler for the List menu item. Associate its COMMAND message with the view class and implement it as follows:

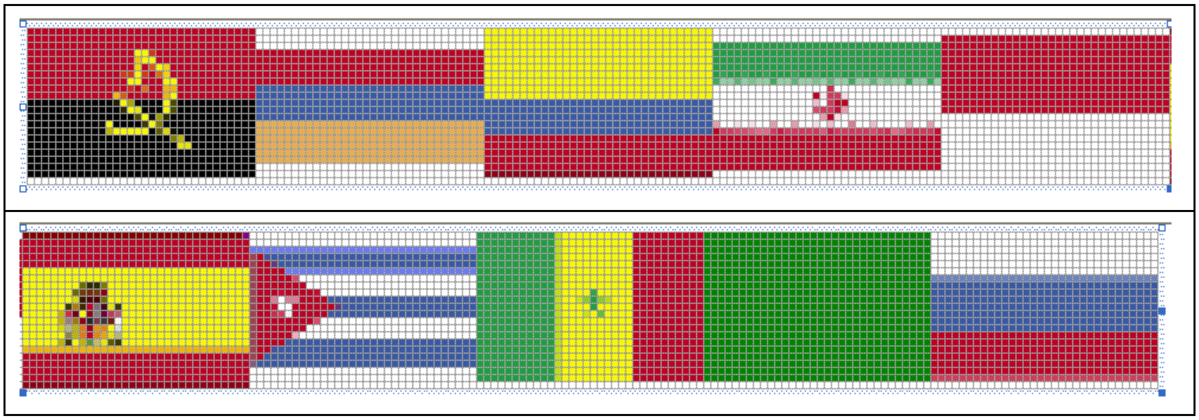
```
void CCountriesView::OnViewDetails()
{
	// TODO: Add your command handler code here
	if( GetViewType() != LVS_REPORT )
		SetViewType(LVS_REPORT);
}
```

20. Test the application and return to MSVC

21. Display the Add Resource dialog box and double-click Bitmap

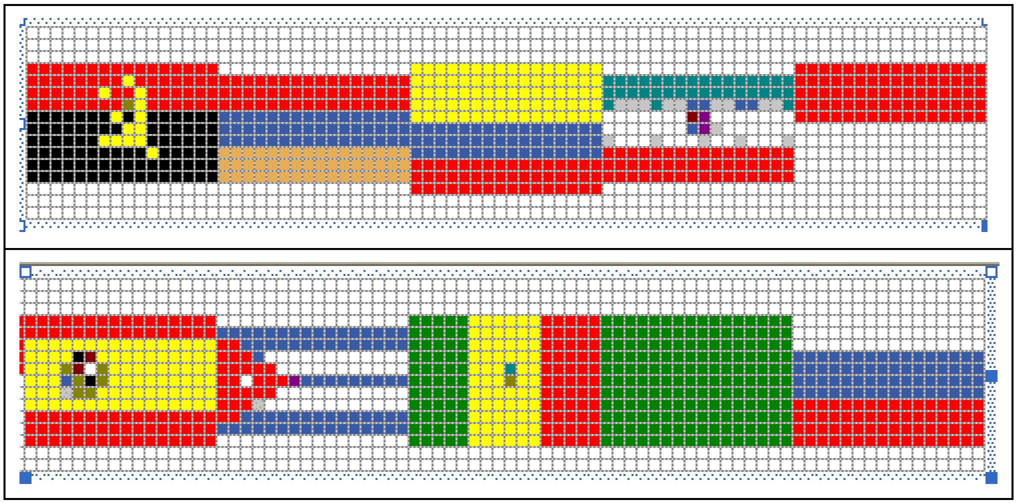
22. On the Properties window, change the ID of the bitmap to IDB_LARGE and change its dimensions 320x32

23. Design the bitmap as follows or import the Large.bmp file from the resources that accompany this book:



24. Add another bitmap. Change its ID to IDB_SMALL and change its dimensions 160x32

25. Design the bitmap as follows or import the small.bmp bitmap it from the resources that accompany this book:



26. In the header file of the view class, declare two CImageList variables named m_Large and m_Small

```
protected:
	CImageList m_Large;
	CImageList m_Small;

// Generated message map functions
protected:
	DECLARE_MESSAGE_MAP()
};
```

27. To use the new bitmaps, change the OnInitialUpdate() event of the view class as follows:

```
void CCountriesView::OnInitialUpdate()
{
	CListView::OnInitialUpdate();

	// TODO: You may populate your ListView with items by directly accessing
	//  its list control through a call to GetListCtrl().
	CListCtrl& lCtrl = GetListCtrl();

	lCtrl.InsertColumn(0, "Name", LVCFMT_LEFT, 100);
	lCtrl.InsertColumn(1, "Area km\262", LVCFMT_RIGHT, 80);
	lCtrl.InsertColumn(2, "Population", LVCFMT_RIGHT, 100);
	lCtrl.InsertColumn(3, "Capital", LVCFMT_LEFT, 80);
	lCtrl.InsertColumn(4, "Independance", LVCFMT_RIGHT, 115);
	lCtrl.InsertColumn(5, "Code", LVCFMT_CENTER, 40);

	lCtrl.SetExtendedStyle(LVS_EX_GRIDLINES | LVS_EX_FULLROWSELECT);

	ModifyStyle(0, LVS_REPORT);

	m_Small.Create(IDB_SMALL, 16, 0, RGB(226, 174, 87));
	m_Large.Create(IDB_LARGE, 32, 0, RGB(192, 192, 192));

	lCtrl.SetImageList(&m_Small, LVSIL_SMALL);
	lCtrl.SetImageList(&m_Large, LVSIL_NORMAL);

	int nItem;

	nItem = lCtrl.InsertItem(0, "Libya", 8);
	lCtrl.SetItemText(nItem, 1, "1,759,540"); lCtrl.SetItemText(nItem, 2, "5,499,074");
	lCtrl.SetItemText(nItem, 3, "Tripoli");   lCtrl.SetItemText(nItem, 4, "24 December 1951");
	lCtrl.SetItemText(nItem, 5, "ly");

	nItem = lCtrl.InsertItem(0, "Senegal", 7);
	lCtrl.SetItemText(nItem, 1, "196,190"); lCtrl.SetItemText(nItem, 2, "10,580,307");
	lCtrl.SetItemText(nItem, 3, "Dakar");   lCtrl.SetItemText(nItem, 4, "4 April 1960");
	lCtrl.SetItemText(nItem, 5, "sn");

	nItem = lCtrl.InsertItem(0, "Cuba", 6);
	lCtrl.SetItemText(nItem, 1, "110,860"); lCtrl.SetItemText(nItem, 2, "11,263,429");
	lCtrl.SetItemText(nItem, 3, "Havana");  lCtrl.SetItemText(nItem, 4, "20 May 1902");
	lCtrl.SetItemText(nItem, 5, "cu");

	nItem = lCtrl.InsertItem(0, "Spain", 5);
	lCtrl.SetItemText(nItem, 1, "504,782"); lCtrl.SetItemText(nItem, 2, "40,217,413");
	lCtrl.SetItemText(nItem, 3, "Pilot");   lCtrl.SetItemText(nItem, 4, "");
	lCtrl.SetItemText(nItem, 5, "es");

	nItem = lCtrl.InsertItem(0, "Indonesia", 4);
	lCtrl.SetItemText(nItem, 1, "1,919,440"); lCtrl.SetItemText(nItem, 2, "234,893,453");
	lCtrl.SetItemText(nItem, 3, "Jakarta");   lCtrl.SetItemText(nItem, 4, "17 August 1945");
	lCtrl.SetItemText(nItem, 5, "id");

	nItem = lCtrl.InsertItem(0, "Russia", 9);
	lCtrl.SetItemText(nItem, 1, "17,075,200"); lCtrl.SetItemText(nItem, 2, "144,526,278");
	lCtrl.SetItemText(nItem, 3, "Moscow");      lCtrl.SetItemText(nItem, 4, "24 August 1991");
	lCtrl.SetItemText(nItem, 5, "ru");

	nItem = lCtrl.InsertItem(0, "Armenia", 1);
	lCtrl.SetItemText(nItem, 1, "29,800");          lCtrl.SetItemText(nItem, 2, "3,326,448");
	lCtrl.SetItemText(nItem, 3, "Yerevan"); lCtrl.SetItemText(nItem, 4, "21 September 1991");
	lCtrl.SetItemText(nItem, 5, "am");

	nItem = lCtrl.InsertItem(0, "Iran", 3);
	lCtrl.SetItemText(nItem, 1, "1.648 mil"); lCtrl.SetItemText(nItem, 2, "68,278,826");
	lCtrl.SetItemText(nItem, 3, "Tehran");    lCtrl.SetItemText(nItem, 4, "1 April 1979");
	lCtrl.SetItemText(nItem, 5, "ir");

	nItem = lCtrl.InsertItem(0, "Colombia", 2);
	lCtrl.SetItemText(nItem, 1, "1,138,910"); lCtrl.SetItemText(nItem, 2, "41,662,073");
	lCtrl.SetItemText(nItem, 3, "Bogota");    lCtrl.SetItemText(nItem, 4, "20 July 1810");
	lCtrl.SetItemText(nItem, 5, "co");

	nItem = lCtrl.InsertItem(0, "Angola", 0);
	lCtrl.SetItemText(nItem, 1, "1,246,700"); lCtrl.SetItemText(nItem, 2, "10,766,471");
	lCtrl.SetItemText(nItem, 3, "Luanda ");   lCtrl.SetItemText(nItem, 4, "11 November 1975");
	lCtrl.SetItemText(nItem, 5, "ao");
}
```

28. Test the application and return to MSVC

Chapter 22: Custom Libraries

✍ The Tree Control

✍ The Tree View

✍ The List Control

☞ The List View

22.1 Introduction to Libraries

22.1.1 Overview

A library is a group of functions, classes, or other resources that can be made available to programs that need already implemented entities without the need to know how these functions, classes, or resources were created or how they function. Although a library can appear as a complete program, in the strict sense, it is not. For example, it cannot be executed by a casual user and it cannot execute its own program. A library makes it easy for a programmer to use a function or functions, a class or classes, a resource or resources created by another person or company and trust that this external source is reliable and efficient. Although the issue or creating or implementing a library may appear intimidating, it is absolutely not. There are only two difficult aspects of a library: 1)Why do you need a library? And what do you want to put in a library? As you see, these two "difficult" aspects we mention have nothing to do with programming. We can also state that we do not have an answer to either of these questions: it will be up to you. What we will do here is to show HOW to create a library, not why. We will also show how to include things in a library. We will not define WHY you should put this function and not that one in a library because, from our experience, it always depends on the programmer, the project, or a group of people working on an application.

22.1.2 Libraries Characteristics

A library is created and functions like a normal regular program, using functions or other resources and communicating with other programs. To implement its functionality, a library contains functions that other programs would need to complete their functionality. At the same time, a library may use some functions that other programs would not need. For this reason, there are two types of functions you will create or include in your libraries. An internal function is one used only by the library itself: the program that use the library, also called the clients of the library, will not need access to these functions. External functions are those that can be accessed by the clients of the library.

There are two broad categories of libraries you will deal with in your programs: static libraries and dynamic libraries. The process of creating each makes the biggest difference, not necessarily their functionalities. Even so, there are various techniques used to create each category (once again, we insist on stating that the process of creating a library will be the least difficult of your tasks).

Microsoft Visual C++ 6 already brought various options for creating or configuring a library. Visual C++ .Net has gone even further by providing a lot more choices and improvements. The types available are static libraries, Win32 dynamic link libraries (DLLs) and MFC DLLs.

22.2 General Static Libraries

22.2.1 Introduction

A static library is a file that contains functions, classes, or resources that an external program can use to complement its functionality. To use a library, the programmer has to create a link to it. The project can be a console application, a Win32 or an MFC application. The library file has the **lib** extension.

22.2.2 Creation of a Static Library

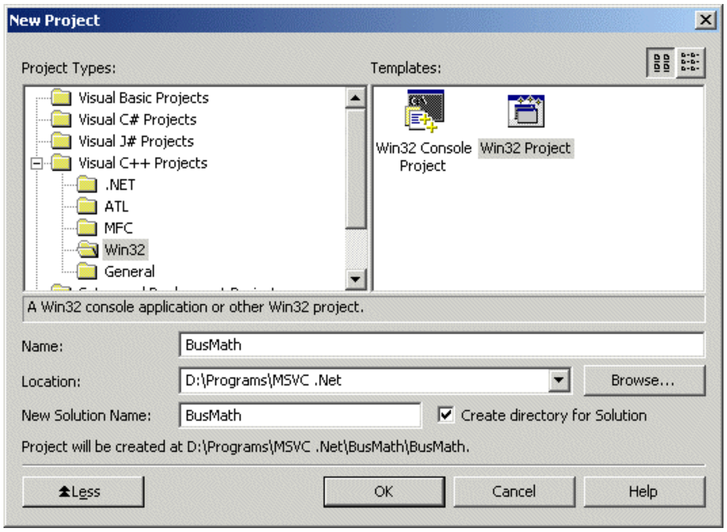
To create a static library, you can open the New Project dialog box and select Win32 Project as the type of application. In the Win32 Application Wizard, select the Static Library radio button in the Application Type section. The following options are also available:

- If you do not check the Precompiled Header option, an empty project would be created for you
- If you check the Precompiled Header option, the wizard would generate a StdAfx.h header file and a StdAfx.cpp source file for the project. This allows you to include common header files in StdAfx.h and omit those common header files in every file of the project. When necessary, the compiler would retrieve the needed header file from StdAfx.h

After laying the foundation of a static library, you can add functions, classes, and/or resources that will be part of the library.

Practical Learning: Creating a Static Library

1. Start Microsoft Visual Studio if necessary and display the New Project dialog box
2. In the Project Types list, make sure you select the Visual C++ Projects node. In the Templates list, click Win32 Project. In the Name edit box, replace the content with BusMath



A Win32 console application or other Win32 project.

Name: BusMath

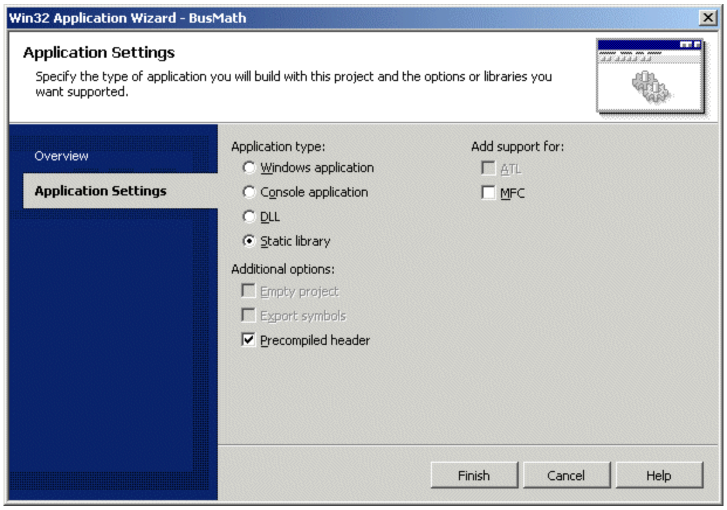
Location: D:\Programs\MSVC .Net

New Solution Name: BusMath

☑ Create directory for Solution

Project will be created at D:\Programs\MSVC .Net\BusMath\BusMath.

3. Click OK
4. In the Win32 Application Wizard, click Application Settings. In the Application Type section, click Static Library



Win32 Application Wizard - BusMath

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:

○ Windows application
○ Console application
○ DLL
● Static library

Additional options:

□ Empty project
□ Export symbols
☑ Precompiled header

Add support for:

□ ATL
□ MFC

5. Click Finish

6. To add a header file, on the main menu, click Project -> Add New Item… In the Add New Item – BusMath dialog box, in the Templates list, click Header File (.h). Replace the content of the Name edit box with **bmcalc** and click Open

7. In the empty file, declare the following functions:

```
#ifndef _BUSINESSMATH_H_
#define _BUSINESSMATH_H_

double Min(const double *Numbers, const int Count);
double Max(const double *Numbers, const int Count);
double Sum(const double *Numbers, const int Count);
double Average(const double *Numbers, const int Count);
long   GreatestCommonDivisor(long Nbr1, long Nbr2);

#endif          // _BUSINESSMATH_H_
```

8. To add a source file, on the main menu, click Project -> Add New Item… In the Templates lis of the Add New Item – BusMath dialog box, click Source File (.cpp). Replace the content of the Name edit box with bmcalc and click Open

9. Implement the functions as follows:

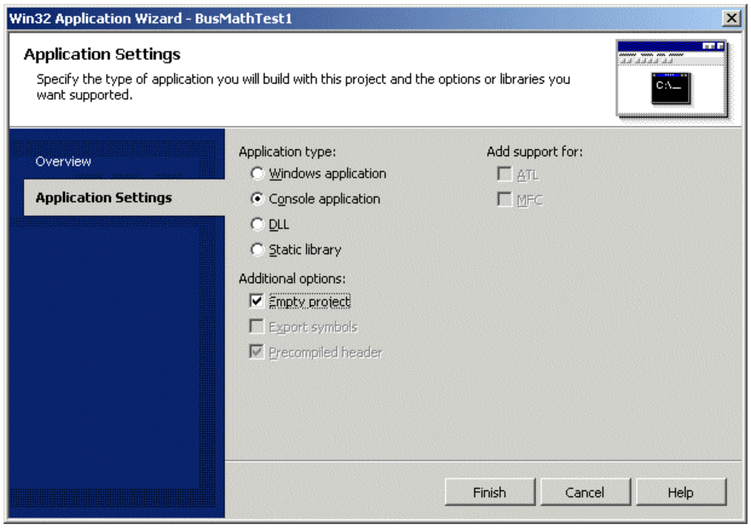
```
#include "StdAfx.h"
#include "bmcalc.h"

double Min(const double *Nbr, const int Total)
{
    double Minimum = Nbr[0];

    for(int i = 0; i < Total; i++)
            if( Minimum > Nbr[i] )
                    Minimum = Nbr[i];
    return Minimum;
}

double Max(const double *Nbr, const int Total)
{
    double Maximum = Nbr[0];

    for(int i = 0; i < Total; i++)
            if( Maximum < Nbr[i] )
```

```
                    Maximum = Nbr[i];
    return Maximum;
}

double Sum(const double *Nbr, const int Total)
{
    double S = 0;

    for(int i = 0; i < Total; i++)
            S += Nbr[i];

    return S;
}

double Average(const double *Nbr, const int Total)
{
    double avg, S = 0;

    for(int i = 0; i < Total; i++)
            S += Nbr[i];
    avg = S / Total;

    return avg;
}

long   GreatestCommonDivisor(long Nbr1, long Nbr2)
{
    while( true )
    {
            Nbr1 = Nbr1 % Nbr2;
            if( Nbr1 == 0 )
                    return Nbr2;

            Nbr2 = Nbr2 % Nbr1;
            if( Nbr2 == 0 )
                    return Nbr1;
    }
}
```

10. To create the library, on the main menu, click Build -> Build BusMath

11. To prepare a test for a console application, display the New Project dialog box. In the Project Type, click Visual C++ Projects and, in the Templates list, click Win32 Project. In the Name edit box, type **BusMathTest1** and press Enter

12. In the Win32 Application Wizard, click Application Settings. In the Application Type section, click the Windows Application radio button. In the Additional Options section, click the Empty Project check box



Win32 Application Wizard - BusMathTest1

Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:

○ Windows application

● Console application

○ DLL

○ Static library

Additional options:

☑ Empty project

□ Export symbols

☑ Precompiled header

Add support for:

□ ATL

□ MFC

13. Click Finish

14. Create a new C++ source file (Project -> Add New Item… -> C++ File (.cpp)) and Name it **Exercise**

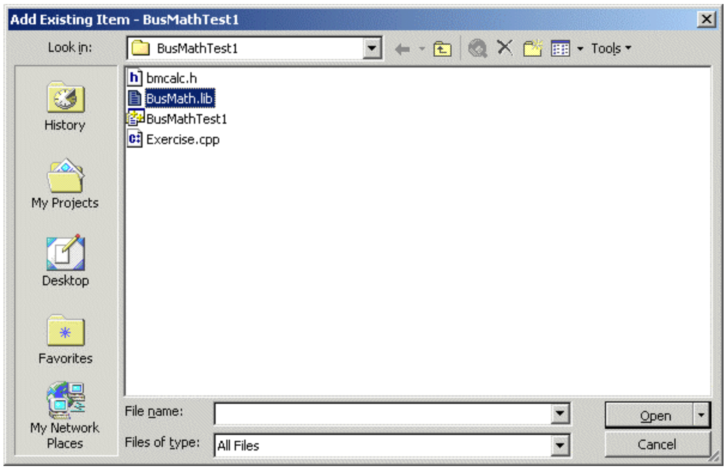
15. Open Windows Explorer or My Computer

16. Locate the folder that contains the above project: BusMath. Open its Debug folder and copy the **BusMath.lib** file. Display the contents of the BusMathTest1 folder and open the BusMathTest1 folder inside of it. Notice that it contains Exercise.cpp. Paste the BusMath.lib file in the same fo lder that contains Exercise.cpp

17. From the BusMath project, copy the **bmcalc.h** header file. Still in Windows Explorer or My Computer, paste bmcalc.h it in the same folder that contains BusMath.lib and Exercise.cpp

18. Back in Visual Studio, to add the library to the current project, on the main menu, click Project -> Add Existing Item. In the Files of Type combo box, select All Files

19. In the list of files, click BusMath.lib



Add Existing Item - BusMathTest1

Look in: BusMathTest1

bmcalc.h
BusMath.lib
BusMathTest1
Exercise.cpp

File name:

Files of type: All Files

20. Click Open

21. Type the following in the empty Exercise.cpp file

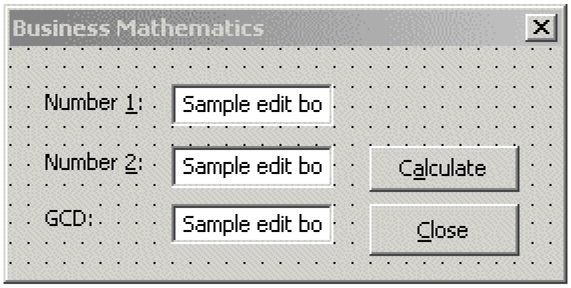
```
#include <iostream>
#include "bmcalc.h"
using namespace std;
```

```
int main()
{
	double Numbers[] = { 12.55, 94.68, 8.18, 60.37, 104.502, 75.05 };
	int Count = sizeof(Numbers) / sizeof(double);

	double Total = Sum(Numbers, Count);
	double Avg   = Average(Numbers, Count);
	double Low    = Min(Numbers, Count);
	double High  = Max(Numbers, Count);

	cout << "Characteristics of the list";
	cout << "\nMinimum: " << Low;
	cout << "\nMaximum: " << High;
	cout << "\nTotal:   " << Total;
	cout << "\nAverage: " << Avg << endl;

	return 0;
}
```

22. Test the application:

```
Characteristics of the list
Minimum: 8.18
Maximum: 104.502
Total:   355.332
Average: 59.222
Press any key to continue
```

23. Close it and retur to MSVC
24. To test the library with a GUI application, display the New Project dialog box. In the Project Types list, make sure Visual C++ Projects is selected. In the Templates list, click MFC Application
25. In the Name edit box, type BusMathTest2 and press Enter
26. Set the Application Type as Dialog Based. Set the Dialog Title to Business Mathematics and remove the About Box check box. Click Finish
27. In Windows Explorer or My Computer, one after the other, copy th BusMath.lib and bmcalc.h files from the BusMath project to the BusMathTest2 folder inside the main BusMathTest2 folder
28. Delete the OK button and design the dialog box as follows:



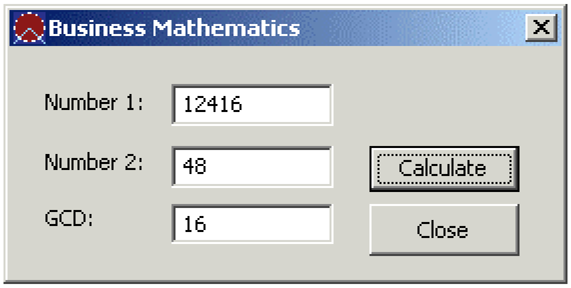
Control	ID	Caption
Static Text		Number &1:
Edit Control	IDC_NUMBER1	
Static Text		Number &2:
Edit Control	IDC_NUMBER2	

Button	IDC_CALCULATE	C&alculate
Static Text		GCD:
Edit Control	IDC_GCD	
Button	IDCANCEL	&Close

29. Add a Value variable for the IDC_NUMBER1 control and name it m_Number1
30. Add a Value variable for the IDC_NUMBER2 control and name it m_Number2
31. Add a Value variable for the IDC_GCD control and name it m_GCD
32. To add the library to the current project, on the main menu, click Project -> Add Existing Item… and select BusMath.lib
33. On the dialog box, double-click the Calculate button and implement it OnClick event as follows:

```
// BusMathTest2Dlg.cpp : implementation file
//

#include "stdafx.h"
#include "BusMathTest2.h"
#include "BusMathTest2Dlg.h"
#include ".\busmathtest2dlg.h"
#include "bmcalc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CBusMathTest2Dlg dialog

CBusMathTest2Dlg::CBusMathTest2Dlg(CWnd* pParent /*=NULL*/)
	: CDialog(CBusMathTest2Dlg::IDD, pParent)
	, m_Number1(_T("0"))
	, m_Number2(_T("0"))
	, m_GCD(_T("0"))
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

. . .

void CBusMathTest2Dlg::OnBnClickedCalculate()
{
	// TODO: Add your control notification handler code here
	int Nbr1, Nbr2, Result;

	UpdateData(TRUE);

	Nbr1 = atoi(m_Number1);
	Nbr2 = atoi(m_Number2);
	Result = GreatestCommonDivisor(Nbr1, Nbr2);

	m_GCD.Format("%d", Result);

	UpdateData(FALSE);
}
```

34. Test the application. Here is an example:



Number 1: 12416

Number 2: 48

GCD: 16

35. Close it and return to MSVC

22.3 MFC Static Libraries

22.3.1 Introduction

An MFC static library is a library file that includes MFC functions or classes. Such a library natively supports MFC strings (such as **CString**, **CTime**, etc), lists, and any other C/C++ or MFC global functions. Because this library is an MFC type, it can be used only by an MFC-based application.

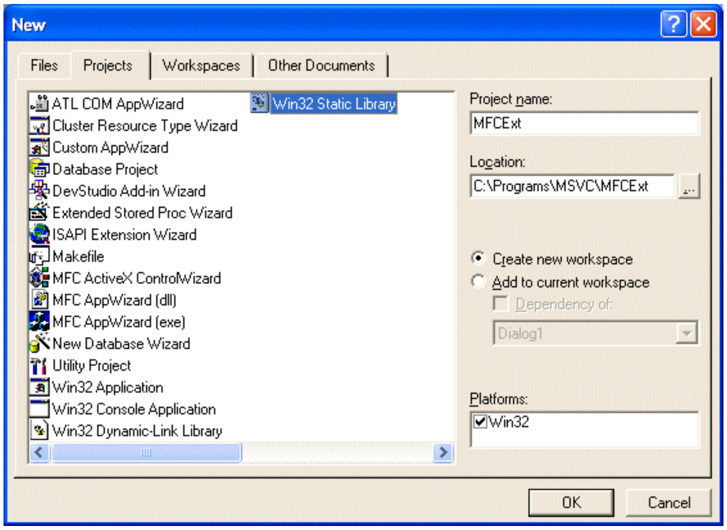
22.3.2 Creation of an MFC Static Library

To create an MFC static library, from the New Project dialog box, select Win32 Project. Then, in the Win32 Application Wizard, select the Static Library radio button in the Application Type section and, in the Add Support For section, click the MFC check box. The following two options are available:

- If you do not select the Precompiled Header radio button, an empty project would be created
- If you select the Precompiled Header radio button, the wizard will generate a StdAfx.h header file and a StdAfx.cpp source file for the project. The wizard will also include the afx.h and the afxwin.h files in the StdAfx.h header file. You can still add any necessary header file in StdAfx.h

Practical Learning: Creating a Static MFC Library

1. To start a new project, on the main menu, click File -> New…
2. In the New dialog box, click the Projects property page
3. In the list, click Win32 Static Library
4. In the Project Name edit box, type **MFCExt**



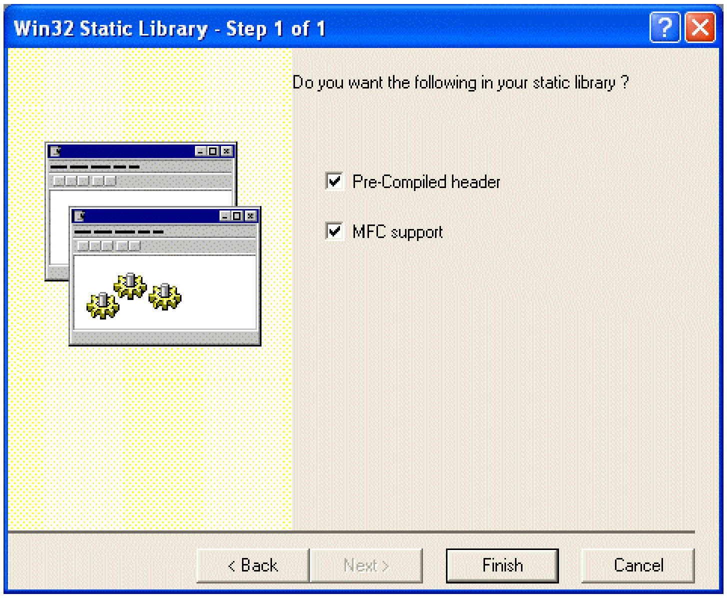
Project name: MFCExt

Location: C:\Programs\MSVC\MFCExt

● Create new workspace
○ Add to current workspace
□ Dependency of: Dialog1

Platforms: ☑ Win32

5. Click OK



Do you want the following in your static library ?

☑ Pre-Compiled header

☑ MFC support

6. In the Win32 Static Library – Step 1 of 1, click both check boxes and click Finish then click OK

7. To add a new header file, on the main menu, click File -> New…

8. In the Files property page of the New dialog box, click C/C++ Header File. In the File Name edit box, type **mfcextent** and click OK

9. In the empty file, type the following:

```
#ifndef MFCEXT_H_
#define MFCEXT_H_

namespace MFCExtensions
{
    BOOL     IsNatural(const CString Str);
    BOOL     IsNumeric(const CString Str);
    int      StringToInt(const CString Str);
    double   StringToFloat(const CString Str);
}
```

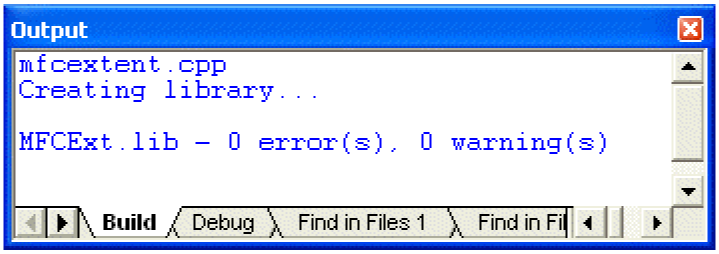
```
#endif          // MFCEXT_H
```

10. To add a new source file, on the main menu, click File -> New… In the Files property page of the New dialog box, click C++ Source File. In the File Name edit box, type **mfcextent** and click OK

11. In the empty file, type the following:

```
#include "stdafx.h"
#include " mfcextent.h"

namespace MFCExtensions
{
    BOOL IsNatural(const CString Str)
    {
            // Check each character of the string
            // If a character at a certain position is not a digit,
            // then the string is not a valid natural number
            for(int i = 0; i < Str.GetLength(); i++)
            {
                    if( Str[i] < '0' || Str[i] > '9' )
                            return FALSE;
            }

            return TRUE;
    }

    BOOL IsNumeric(const CString Str)
    {
            // Make a copy of the original string to test it
            CString WithoutSeparator = Str;
            // Prepare to test for a natural number
            // First remove the decimal separator, if any
            WithoutSeparator.Replace(".", "");

            // If this number were natural, test it
            // If it is not even a natural number, then it can't be valid
            if( IsNatural(WithoutSeparator) == FALSE )
                    return FALSE; // Return Invalid Number

            // Set a counter to 0 to counter the number of decimal separators
            int NumberOfSeparators = 0;

            // Check each charcter in the original string
            for(int i = 0; i < Str.GetLength(); i++)
            {
                    // If you find a decimal separator, count it
                    if( Str[i] == '.' )
                            NumberOfSeparators++;
            }

            // After checking the string and counting the decimal separators
            // If there is more than one decimal separator,
            // then this cannot be a valid number
            if( NumberOfSeparators > 1 )
                    return FALSE;      // Return Invalid Number
            else    // Otherwise, this appears to be a valid decimal number
                    return TRUE;
    }
    int StringToInt(const CString Str)
    {
            // First check to see if this is a valid natural number
            BOOL IsValid = IsNatural(Str);

            // If this number is valid, then convert it
            if( IsValid == TRUE )
                    return atoi(Str);
            else
            {
                    // Return 0 to be nice
                    return 0;
            }
    }

    double  StringToFloat(const CString Str)
    {
            // First check to see if this is a valid number
            BOOL IsValid = IsNumeric(Str);

            // If this number is valid, then convert it
            if( IsValid == TRUE )
                    return atof(Str);
            else
            {
                    // Return 0 to be nice
                    return 0;
            }
    }
}
```

12. To create the library, on the main menu, click Build -> Build MFCExt.lib
    The Output window will indicate that a file with .lib extension was created



```
mfcextent.cpp
Creating library...

MFCExt.lib - 0 error(s), 0 warning(s)
```

22.3.3 MFC Static Library Test

Like the regular static library we saw above, you can use an MFC static library either on a console or a graphical application. The main difference is that the application must be able use MFC. Therefore, when creating the application, specify that you will use MFC either In A Shared DLL or In A Static DLL.

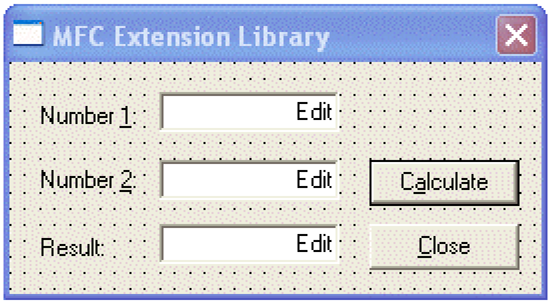
Practical Learning: Testing an MFC Static Library

1. To start a new application, on the main menu, click File -> New…
2. In the Projects tab of the New dialog box, click MFC AppWizard. In the Project Name edit box, type MFCExtTest and click OK

3. Set the project type to Dialog Based and click Next

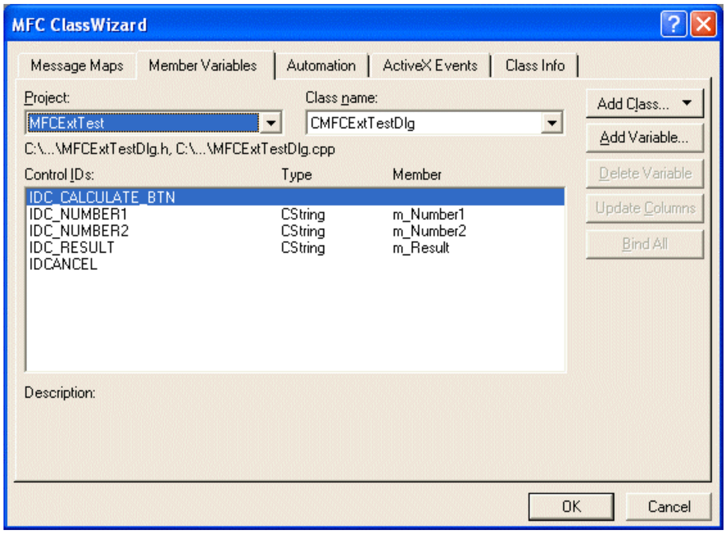
4. Remove the check box of the About Box. Set the Title to **MFC Extension Library** then click Finish and click OK

5. Design the dialog box as follows:



Control	ID	Caption	Addition Properties
Static Text		Number &1:	
Edit Control	IDC_NUMBER1		Align Text: Right
Static Text		Number &2:	
Edit Control	IDC_NUMBER2		Align Text: Right
Button	IDC_CALCULATE	C&alculate	Default Button: True
Static Text		Result:	
Edit Control	IDC_RESULT		Align Text: Right
Button	IDCANCEL	&Close	

6. Press Ctrl + W to access the ClassWizard. In the Member Variables tab, Add a Variable for each edit box as follows:



Project: MFCExtTest

Class name: CMFCExtTestDlg

C:\...\MFCExtTestDlg.h, C:\...\MFCExtTestDlg.cpp

Control IDs:	Type	Member
IDC_CALCULATE_BTN		
IDC_NUMBER1	CString	m_Number1
IDC_NUMBER2	CString	m_Number2
IDC_RESULT	CString	m_Result
IDCANCEL		

7. Using Windows Explorer or My Computer, copy the MFCExt.lib and the mfcextent.h files from the folder of the previously created library and paste them in the folder of the current project (the lib and the h files are in different folder but you should paste them in the main folder of the current project)

8. To include the library in the current project, on the main menu, click Project -> Add To Project -> Files…

9. Change the Files Of Type to Library Files (.lib). Select the MFCExt.lib file and click OK

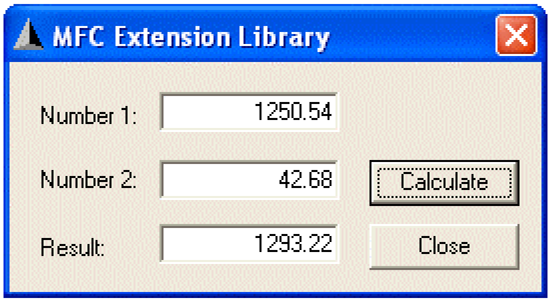
10. Display the dialog box. Double -click the Calculate button. Accept the suggested name of the function and click OK

11. Implement the event as follows:

```
// MFCExtTestDlg.cpp : implementation file
//

#include "stdafx.h"
#include "MFCExtTest.h"
#include "MFCExtTestDlg.h"

#include "mfcextent.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMFCExtTestDlg dialog

CMFCExtTestDlg::CMFCExtTestDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CMFCExtTestDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CMFCExtTestDlg)
    m_Number1 = _T("0.00");
    m_Number2 = _T("0.00");
    m_Result = _T("0.00");
    //}}AFX_DATA_INIT
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

. . .

void CMFCExtTestDlg::OnCalculateBtn()
{
    // TODO: Add your control notification handler code here
    double Number1, Number2, Result;

    UpdateData();

    // Evaluate the content of each operand edit box
    if( MFCExtensions::IsNumeric(m_Number1) == TRUE )
    {
            Number1 = MFCExtensions::StringToFloat(m_Number1);
    }
    else
    {
            AfxMessageBox("Invalid Number");
            Number1 = 0.00;
    }

    if( MFCExtensions::IsNumeric(m_Number2) == TRUE )
    {
            Number2 = MFCExtensions::StringToFloat(m_Number2);
    }
    else
    {
```

```
                AfxMessageBox("Invalid Number");
                Number2 = 0.00;
        }

        Result = Number1 + Number2;

        m_Result.Format("%.2f", Result);
        UpdateData(FALSE);
}
```

12. Test the application



22.4 Win32 DLL

22.4.1 Introduction

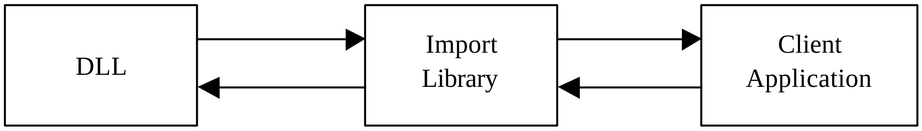
A Win32 DLL is a library that can be made available to programs that run on a Microsoft Windows computer. As a normal library, it is made of functions and/or other resources grouped in a file. The DLL abbreviation stands for Dynamic Link Library. This means that, as opposed to a static library, a DLL allows the programmer to decide on when and how other applications will be linked to this type of library. For example, a DLL allows difference applications to use its library as they see fit and as necessary. In fact, applications created on different programming environments can use functions or resources stored in one particular DLL. For this reason, an application dynamically links to the library.

You create a DLL like any other application, as a project. This project must contain at least one file. From your experience, you probably know that each C++ program contains the **main()** function and each Win32 application contains a **WinMain()** function. These are referred to as entry-points because the compiler always looks for these functions as the starting point of an application. In the same way, a Win32 DLL must contain an entry point function. The most regularly used function as the entry point is called **DllMain**. Unlike a C++ program that uses **main()** and unlike a Win32 application that uses **WinMain()**, a DLL can have a different function as the entry-point but it must have an entry point. When building your DLL, you will need to communicate your entry-point to the compiler.

When creating the DLL, you must provide a way for external applications to access the functions or resources included in the DLL. There are two main ways you will do this: using a statement that indicates that a particular function will be exported, or by creating an additional specially configured file that will accompany the DLL.

22.4.2 Fundamentals of a DLL

We have mentioned that a DLL is created as a project that contains at least one source file and this source file should present an entry-point. After creating the DLL, you will build and distribute it so other programs can use it. When building it, you must create a library file that will accompany the DLL. This library file will be used by other programs to import what is available in the DLL:



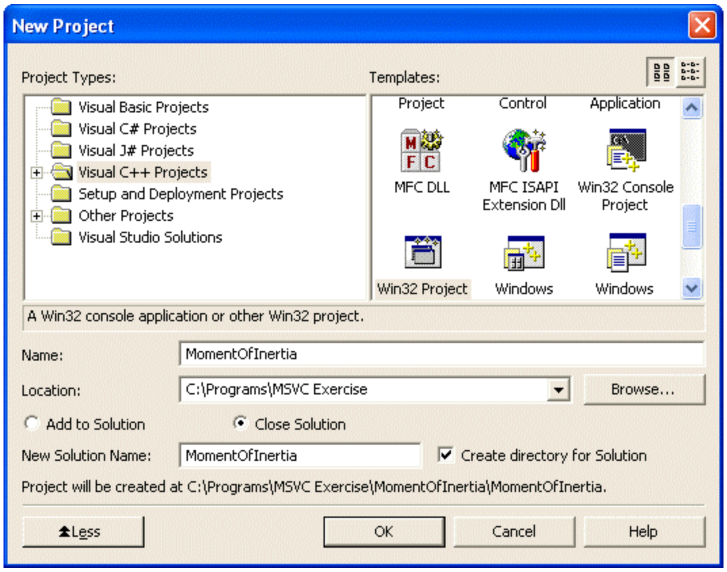
When the import library is created, it contains information about where each available function is included in the DLL and can locate it. When an application needs to use a function contained in the DLL, it presents its request to the import library. The import library checks the DLL for that function. If the function exists, the client program can use it. If it does not, the library communicates this to the application and the application presents an error.

Normally, the import library is created when building the DLL but you must provide a mechanism for the compiler to create it. There are two main ways you do this. You can use a calling convention that allows the compiler to detect what function will be accessible to the clients of the DLL. In this case, each one of these functions will start with the **_declspec** modifier. When creating the DLL, the **_declspec** modifier must take as argument the **dllexport** keyword.

The functions do not have to be included in the main file that contains the entry-point; it may simply be convenient to do it that way.

Practical Learning: Creating a DLL

1. To start a new project, on the main menu, click File -> Project…
2. In the New Project dialog box, in the Project Type list, select Visual C++. In the Templates list, click Win32 Project
3. In the Name edit box, type MomentOfInertia



A Win32 console application or other Win32 project.

Name: MomentOfInertia

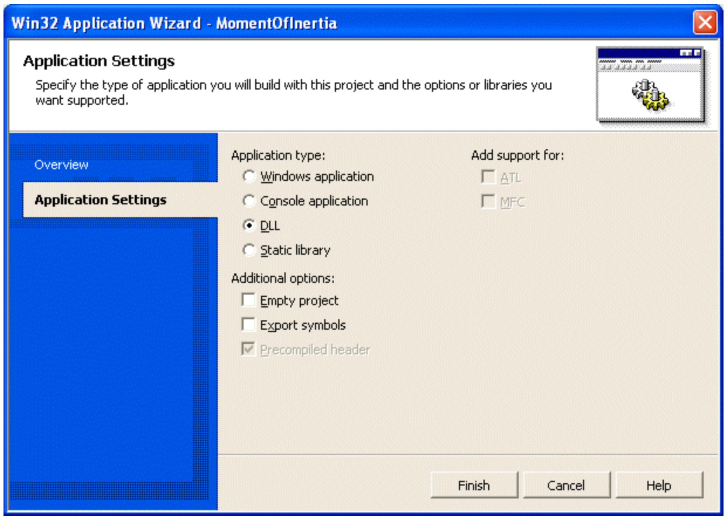
Location: C:\Programs\MSVC Exercise

○ Add to Solution ● Close Solution

New Solution Name: MomentOfInertia ☑ Create directory for Solution

Project will be created at C:\Programs\MSVC Exercise\MomentOfInertia\MomentOfInertia.

4. Click OK
5. In the Win32 Application Wizard, in the left frame, click Application Settings. In the Application Type section, click the DLL radio button



Application Settings

Specify the type of application you will build with this project and the options or libraries you want supported.

Application type:

○ Windows application
○ Console application
● DLL
○ Static library

Additional options:

□ Empty project
□ Export symbols
☑ Precompiled header

Add support for:

□ ATL
□ MFC

6. Click Finish
7. Change the contents of the MomentOfInertia.cpp file as follows:

```
// MomentOfInertia.cpp : Defines the entry point for the DLL application.
//

#include "stdafx.h"
// Calculation of the moment of inertia
// Rectangle
extern "C" _declspec(dllexport)double MOIRectangle(double b, double h);
// Semi-Circle
extern "C" _declspec(dllexport)double MOISemiCircle(double r);
// Triangle
extern "C" _declspec(dllexport)double MOITriangle(double b, double h, int);

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                LPVOID lpReserved
                                          )
{
   return TRUE;
}

double MOIRectangle(double b, double h)
{
    return b * h * h * h / 3;
}

double MOISemiCircle(double r)
{
    const double PI = 3.14159;

    return r * r * r * r * PI / 8;
}

double MOITriangle(double b, double h, int)
{
    return b * h * h * h / 12;
}
```

8. To create the DLL, on the main menu, click Build -> Build MomentOfInertia
9. Open Windows Explorer and open the Debug folder of the current project. Notice that a file with dll extension and another file with lib extension have been created

22.4.3 Win32 DLL Test

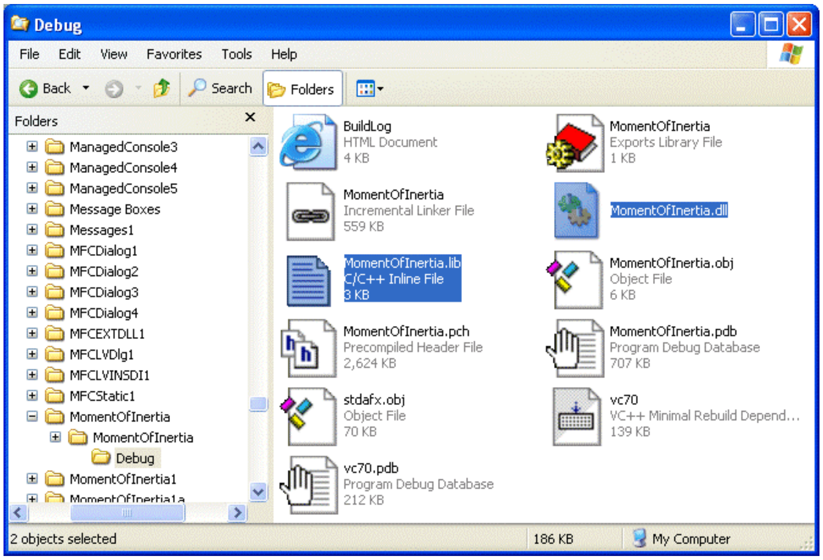
Naturally you should test your DLL to make sure it behaves as expected, before distributing it to other applications. For an application to be able to use the DLL, it must be able to locate its dll and its lib files. The simplest way to do this is to simply copy and paste these files in the client project. Some other times, the DLL and its library will need to reside in the system directory. To make the DLL available to an application, you must import the DLL into the application. Although there are various ways to do this, the simplest consists of adding it to the project by using the main menu where you would click Project -> Add Existing Item…, and selecting the lib file.

We mentioned that, when creating the DLL, you should use the **_declspec(dllexport)** modifier for each function that will be accessed outside of the DLL. In the project that is using the DLL, each function that will be accessed must be declared using the **_declspec(dllimport)** modifier.

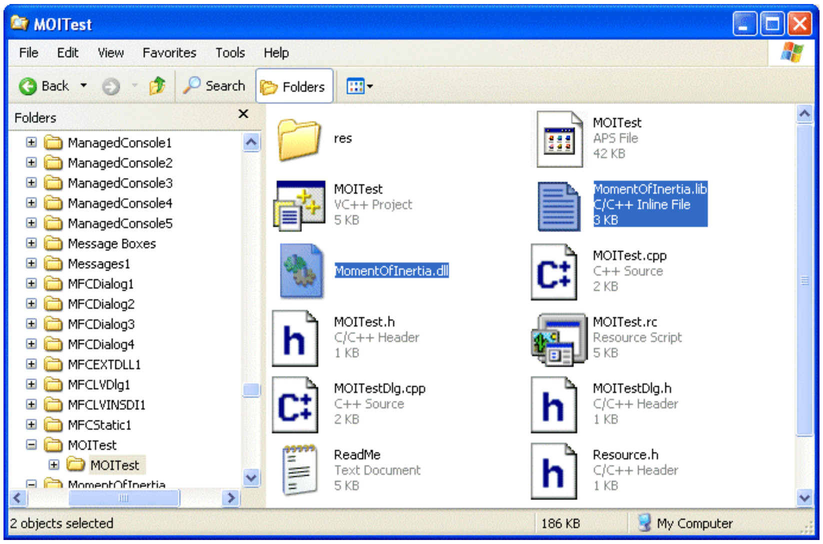
Practical Learning: Testing a DLL

1. To start a new project, display the New Project dialog box
2. In the Project Types, click Visual C++ Projects. In the Templates, click MFC Application
3. In the Name edit box, type **MOITest** and press Enter
4. In the MFC Application Wizard, create the application as Dialog Based without an About Box and set the **Dialog Title** to **Moment Of Inertia DLL Test**
5. Click Finish

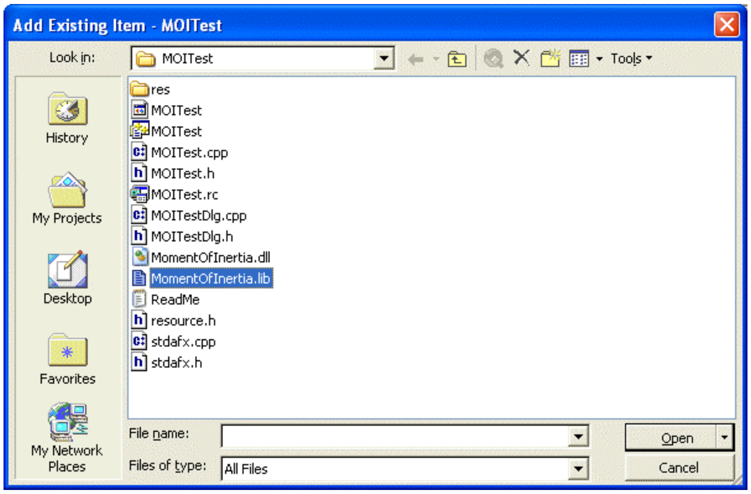
6. Open Windows Explorer or My Computer. Locate the folder that contains the MomentOfInertia project
7. Select and copy both the MomentOfInertia.dll and the MomentOfInertia.lib files


Debug
File Edit View Favorites Tools Help
Back Search Folders
Folders
ManagedConsole3
ManagedConsole4
ManagedConsole5
Message Boxes
Messages1
MFCDialog1
MFCDialog2
MFCDialog3
MFCDialog4
MFCEXTDLL1
MFCLVDlg1
MFCLVINSDI1
MFCStatic1
MomentOfInertia
MomentOfInertia
Debug
MomentOfInertia1
MomentOfInertia1a
BuildLog HTML Document 4 KB
MomentOfInertia Exports Library File 1 KB
MomentOfInertia Incremental Linker File 559 KB
MomentOfInertia.dll
MomentOfInertia.lib C/C++ Inline File 3 KB
MomentOfInertia.obj Object File 6 KB
MomentOfInertia.pch Precompiled Header File 2,624 KB
MomentOfInertia.pdb Program Debug Database 707 KB
stdafx.obj Object File 70 KB
vc70 VC++ Minimal Rebuild Depend... 139 KB
vc70.pdb Program Debug Database 212 KB
2 objects selected
186 KB
My Computer

8. Locate the folder that contains the current new project then paste the dll and the lib files:



9. Back in MSVC, on the main menu, click Project -> Add Existing Item…
10. In the Add Existing Item dialog box, change the Files of Type to All Files
11. In the list of files, click MomentOfInertia.lib

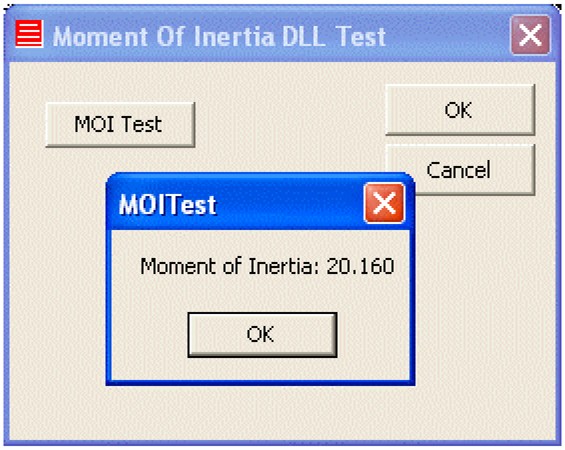


12. Click Open
13. Delete the TODO line on the dialog box. Add a new **Button**. Change its **Caption** to **MOI Test**
14. Double-click the new button to initiate its **BN_CLICKED** message
15. Implement it as follows:

```
// MOITestDlg.cpp : implementation file
//

#include "stdafx.h"
#include "MOITest.h"
#include "MOITestDlg.h"
#include ".\moitestdlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// Calculation of the moment of inertia
// Rectangle
extern "C" _declspec(dllimport)double MOIRectangle(double b, double h);
// Semi-Circle
extern "C" _declspec(dllimport)double MOISemiCircle(double r);
// Triangle
extern "C" _declspec(dllimport)double MOITriangle(double b, double h, int);
// CMOITestDlg dialog

CMOITestDlg::CMOITestDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CMOITestDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

. . .

void CMOITestDlg::OnBnClickedButton1()
{
	// TODO: Add your control notification handler code here
	double MOI = MOIRectangle(3.25, 2.65);
	char Sentence[40];
```

```
        sprintf(Sentence, "Moment of Inertia: %.3f", MOI);
        MessageBox(Sentence);
}
```

16. Test the application



Moment of Inertia: 20.160

17. Close the dialog box and return to MSVC

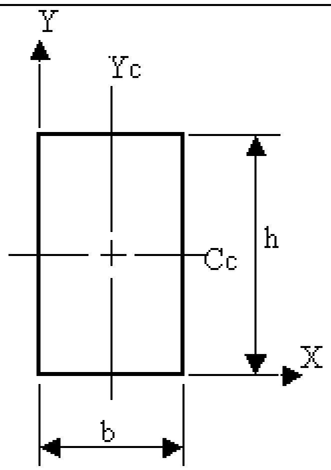
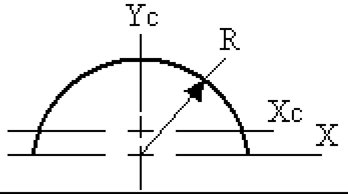
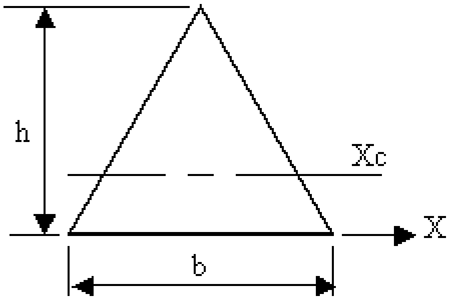
22.5 DLL Module-Definition Files

22.5.1 Definition File Fundamentals

We mentioned that a dll must provide a means of importing its functions and making them available to client applications. We learned above how to help the compiler create the import library by preceding at least one function with the **_declspec(dllexport)** modifier. Microsoft Windows allows another technique. Instead of preceding your functions with a modifier, you can instead add another object called the Module-Definition file.

A definition file is a text file that has the extension **def**. It must contain at least two sections. The first section is made of one line. It starts with the LIBRARY word followed by the name of the DLL. It is important that the name you specify be the same name as the DLL that will be made available to other applications. The second section starts with the EXPORTS word and contains a list of the functions that will be exported.

In the following exercise, we will create a DLL that can be used to calculate the moment of inertia using the following illustrations and formulas:

Rectangle		$I_x = \frac{bh^3}{3}$ $I_y = \frac{hb^3}{3}$ $I_{xc} = \frac{bh^3}{12}$ $I_{yc} = \frac{hb^3}{12}$
Semi-Circle		$I_{xc} = 0.110R^4$ $I_{yc} = I_x = \frac{\pi R^4}{8}$
Triangle		$I_{xc} = \frac{bh^3}{36}$ $I_x = \frac{bh^3}{12}$

Practical Learning: Creation a Definition DLL

1. To start a new application, open the New Project dialog box and, in the Project Types list, click Visual C+ Projects
2. In the Templates list, click Win32 Project. In the Name box, type MomentInertia and press Enter
3. In the Win32 Application Wizard, click Application Settings. Click the DLL radio button and click Finish
4. To add a header file, on the main menu, click Project -> Add New Item…
5. In the Add New Item dialog box, click Header File (.h). In the Name edit box, type MomentOfInertia and press Enter
6. In the empty file, type the following:

```
// MomentOfInertia.cpp : Defines the entry point for the DLL application.
//
#pragma once
```

```
// Calculation of the moment of inertia
// Rectangle
double MOIRectX(double b, double h);
double MOIRectY(double b, double h);
double MOIRectXC(double b, double h);
double MOIRectYC(double b, double h);

// Circle
double MOICircleXC(double r);
double MOICircleYC(double r);

// Semi-Circle
double MOISemiCircleX(double r);
double MOISemiCircleXC(double r);
double MOISemiCircleYC(double r);

// Triangle
double MOITriangleX(double b, double h, int);
double MOITriangleXC(double b, double h, int);
```

7. To add the accompanying source file, display the Add New Item dialog box and select C++ File (.cpp). In the Name edit box, type MomentOfInertia and press Enter

8. In the empty file, type the following:

```
#include "stdafx.h"
#include "MomentOfInertia.h"

const double PI = 3.14159;

// Rectangle
double MOIRectX(double b, double h)
{
    return b * h * h * h / 3;
}
double MOIRectY(double b, double h)
{
    return h * b * b * b / 3;
}
double MOIRectXC(double b, double h)
{
    return b * h * h * h / 12;
}
double MOIRectYC(double b, double h)
{
    return h * b * b * b / 12;
}

// Circle
double MOICircleXC(double r)
{
    return PI * r * r * r * r / 4;
}
double MOICircleYC(double r)
{
    return PI * r * r * r * r / 4;
}

// Semi-Circle
```

```
double MOISemiCircleX(double r)
{
    return r * r * r * r * PI / 8;
}
double MOISemiCircleXC(double r)
{
    return 0.110 * r * r * r * r;
}
double MOISemiCircleYC(double r)
{
    return MOISemiCircleX(r);
}

// Triangle
double MOITriangleX(double b, double h, int)
{
    return b * h * h * h / 12;
}
double MOITriangleXC(double b, double h, int)
{
    return b * h * h * h / 36;
}
```

9. To add a definition file, display the Add New Item dialog box again. In the Templates list, click Module-Definition File (.def)
10. In the Name edit box, type MomentInertia and press Enter
11. Change the file as follows:

```
LIBRARY       MomentInertia

EXPORTS
    MOIRectX
    MOIRectY
    MOIRectXC
    MOIRectYC

    MOICircleXC
    MOICircleYC

    MOISemiCircleX
    MOISemiCircleXC
    MOISemiCircleYC;

    MOITriangleX
    MOITriangleXC
```

12. To create the DLL, on the main menu, click Build -> Build MomentInertia

22.5.2 Usage of a Definition File DLL

Although we mentioned that you can create a definition file to have an import library, you do not have to use it, although you can. The technique we used above allows the compiler to know that you need an import library. This means that there are at least two different ways you can make use of a definition file (as we stated already, the most difficult aspect of DLL is based on decisions, not how to create a library).

Once again, you can use the dll and lib files in the client applications of your DLL, as we did already. If you do not want to use the definition file, you must provide a mechanism for a client application to locate the function that is included in your DLL. This time also, there are different ways you can do this. When a client application wants to use a function, it can declare the function in the file that will call the function. The declaration is done as a normal C++ function declaration. The function must be listed as it appears in the DLL. This is why you should (honestly you must) always provide documentation for your DLL: other people or companies should spend time predicting or guessing what your DLL is used for or what it contains. In the following example, a function called Number() is called from main(). The function is only declared but it is not defined because this was already taken care of in a DLL:

```
#include <iostream>
using namespace std;

double Number();

int main()
{
	double Nbr = Number();

	cout << "Number: " << Nbr << endl;
	return 0;
}
```

Another technique consists of including the header file that contains the function definitions from the DLL. This means that, besides the dll and the lib files, if you distribute your library, you must also distribute the header file (provided you are not distributing the def file, although you can distribute both).

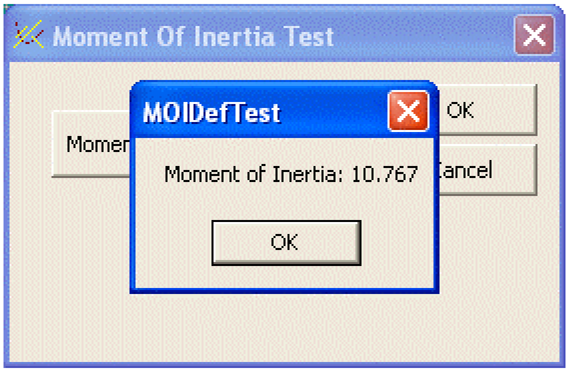
Practical Learning: Using a Module Definition DLL

1. To start a new application, display the New Project dialog box and select MFC application
2. In the Name edit box, type **MOIDefTest** and press Enter
3. In the MFC Application Wizard, create the application as Dialog Based without an About Box and set the **Dialog Title** to DLL Definition File Test
4. Click Finish
5. Open Windows Explorer or My Computer. Locate the folder that contains the MomentInertia project
6. Select and copy both the MomentInertia.dll and the MomentInertia.lib files
7. Locate the folder that contains the current new project then paste the dll and the lib files
8. Back in MSVC, on the main menu, click Project -> Add Existing Item…
9. In the Add Existing Item dialog box, change the Files of Type to All Files
10. In the list of files, click MomentInertia.lib
11. Click Open
12. Delete the TODO line on the dialog box. Add a new button. Change its Caption to **Moment Of Inertia**
13. Double-click the new button to initiate its **BN_CLICKED** message

14. Implement it as follows:

```
// MOIDefTestDlg.cpp : implementation file
//

#include "stdafx.h"
#include "MOIDefTest.h"
#include "MOIDefTestDlg.h"
#include ".\moideftestdlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

double MOIRectX(double b, double h);

// CMOIDefTestDlg dialog

CMOIDefTestDlg::CMOIDefTestDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CMOIDefTestDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

. . .

void CMOIDefTestDlg::OnBnClickedButton1()
{
	// TODO: Add your control notification handler code here
	double MOI = MOIRectX(3.25, 2.15);
	char Sentence[40];

	sprintf(Sentence, "Moment of Inertia: %.3f", MOI);
	MessageBox(Sentence);
}
```

15. Test the application



Moment of Inertia: 10.767

16. Close the dialog box and return to MSVC
17. To use the header file, return to Windows Explorer or My Computer
18. Locate the folder that contains the MomentInertia project and copy the MomentOfInertia.h header file
19. Paste the MomentOfInertia.h file in the folder that contains the current project
20. Change the above file as follows:

```
#include "stdafx.h"
```

```
#include "MOIDefTest.h"
#include "MOIDefTestDlg.h"
#include ".\moideftestdlg.h"
#include "MomentOfInertia.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CMOIDefTestDlg dialog

CMOIDefTestDlg::CMOIDefTestDlg(CWnd* pParent /*=NULL*/)
      : CDialog(CMOIDefTestDlg::IDD, pParent)
{
      m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

21. Test the application. Then close it and return to MSVC

Index