Zero to Mastery

Microsoft Visual in C++

Zero to Mastery

Microsoft Visual in C++

Dr. RK Jain

• Shadab Saifi (***Illustrator***) • Ayaz Uddin (***Editor***)

An ISO 9001:2008 Certified Company

Vayu Education of India

2/25, Ansari Road, Darya Ganj, New Delhi-110 002

First Edition: 2022

DISCLAIMER

Errors, if any, are purely unintentional and readers are requested to communicate such errors to the publisher to avoid discrepancies in future.

Published by:

AN ISO 9001:2008 CERTIFIED COMPANY

VAYU EDUCATION OF INDIA

2/25, ANSARI ROAD, DARYA GANJ, NEW DELHI-110 002
PH.: 011-41564440, MOB. 09910115201

Preface

Microsoft Visual C++ (often abbreviated as MSVC) is a commercial Integrated Development Environment (IDE) product engineered by Microsoft for the C, C++, and C++/CLI programming languages. It has tools for developing and debugging C++ code, especially code written for the Microsoft Windows API, the DirectX API, and the Microsoft NET Framework.

Microsoft Visual C++ has always been one of the most comprehensive and sophisticated software development environments available. It has consistently provided a high level of programming power and convenience, while offering a diverse set of tools designed to suit almost every programming style. And Visual C++ version 6 adds significantly to the already impressive array of features. New features include easier application coding, building, and debugging; greater support for ActiveX and Internet technologies; additional database development options; and new application architectures and user-interface elements, such as Microsoft Internet Explorer 4 style controls.

Book is designed to cover the fundamental aspects of Visual C++ and thus covers the aspects in easy to understand language. Diagrammatic approach is being used to explain the concepts wherever possible. The books include significant number of multiple choice questions. You need not to know any programming language in order to use this book; it teaches you the basics of C++ followed by fundamentals of VC++.

Chapter 1 gives an introduction about Microsoft Visual C++. It talks about what is all included in VC++, its development environment.

Chapter 2 introduces class in C++ covering the fundamental features in C++, concept of constructors & destructors, THIS keyword, static members, inline functions etc.

Chapter 3 covers the concept of overloading in C++ including function and operator overloading.

Chapter 4 is about inheritance, polymorphism and virtual functions.

Chapter 5 you start using VC++ IDE where you learn the various options in the IDE including how to compile, debug and run the programs.

Chapter 6 you start using the GUI based development environment to make the programs. It teaches how the VC++ program works.

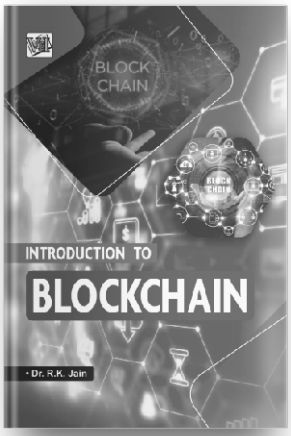
Chapter 7 covers the windows, dialog boxes and controls.

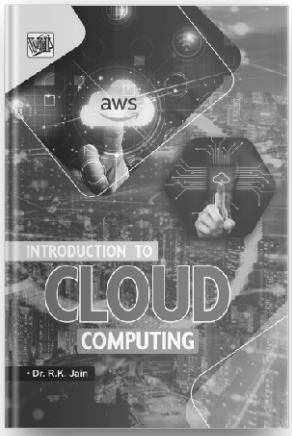
Chapter 8 covers Dialogs and Property Sheets in Visual C++.

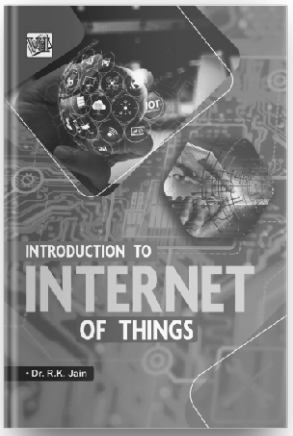
Contents

SPECIAL BONUS!

Want These 3 Bonus Books for free?







Get FREE, unlimited access to these and all of our new books by joining our community!

SCAN w/ your camera TO JOIN!

OR Visit

freebie.kartbucket.com

CHAPTER 1

INTRODUCTION TO VC++

1.1 WHAT IS MICROSOFT VISUAL C++?

Microsoft Visual C++ (often abbreviated as **MSVC**) is a commercial integrated development environment (IDE) product engineered by Microsoft for the C, C++, and C++/CLI programming languages. It has tools for developing and debugging C++ code, especially code written for the Microsoft Windows API, the DirectX API, and the Microsoft .NET Framework.

Visual C++ is one of the most widespread and important languages available today for developing applications for the Windows operating system. Developed and sold by Microsoft, Visual C++ is actually an entire development environment.

What this means is that Microsoft first took C++, which is a common, powerful programming language that can be used to write any kind of application for any kind of operating system.

They then devised a set of functions written in C++ that allow a programmer to control the Windows environment. For example, one function might draw a window on the screen while another might print text in that window. This set of functions is called the MFC, or Microsoft Foundation Class. (For the more advanced out there, the MFC functions wrap the Windows API functions, and hence make them easier to use and object-oriented as well).

Finally, they developed an application that allows a programmer to easily create code using these MFC functions. This application is what you actually buy and install on your computer.

Visual C++ is more complicated than programming in C++ on a text based system for three main reasons:

1.1.1 Windows Programs are Event-Driven

If you're used to programming in C++ on a Unix system, for instance, you're used to writing a "Main" function that controls the execution of your program. The main function starts at the top and moves to the bottom, executing each line of code in turn. This makes program execution very easy to follow.

Windows programs, on the other hand, are driven by events. If the user clicks the mouse in a certain place or selects a certain menu option, the program performs a certain task, etc. You can visualize an event-driven program as a collection of functions that exist in no particular order, and each function is executed by a particular event. This is incredibly confusing and frustrating because you don't know where a program starts or ends.

1.1.2 The MFC

The MFC is extremely complicated and large. Every little thing you do in Windows, like printing text, displaying an icon etc. requires you to research and learn obscure functions. Be prepared to use online help extensively.

1.1.3 Visual C++ is Object Oriented

If you're already a C++ programmer, you're used to object - oriented programming, but if you're a C programmer you may not be. Object orientation means that every function and variable in a program exist as part of organizational units called objects. For instance, a database program might contain an object called Record. The Record object might contain three variables called Name, Address and Phone Number, and a bunch of functions that allow users to enter and change these pieces of information. Each time a user adds a new record, a new instance of the record object is created. An object can contain child objects, which inherit all the properties of the parent object and add their own as well.

1.2 WHAT'S INCLUDED IN VISUAL C++ ?

1.2.1 VC++ Developer Studio

The Developer Studio is the core of the Visual C++ product. It is an integrated application that provides a complete set of programming tools. The Developer Studio includes a project manager for keeping track of your program source files and build options, a text editor for entering program source code and a set of resource editors for designing program resources, such as menus, dialog boxes, and icons. It also provides

programming *wizards* (*AppWizard* and *ClassWizard*), which help you to generate the basic source code for your programs, define C++ classes, handle Windows messages, and perform other tasks. You can build and execute your programs from within the Developer Studio, which automatically runs the optimizing compiler, the incremental linker, and any other required build tools. You can also debug programs using the integrated debugger, and you can view and manage program symbols and C++ classes using the ClassView window.

1.2.2 VC++ Runtime Libraries

The Visual C++ runtime libraries provide standard functions such as strcpy and sprintf, which you can call from either C or C++ programs. If you perform a custom installation of Visual C++, the Setup program lets you select the specific library version or versions that you want to copy to your hard disk (static, shared, or single-threaded). You can also opt to copy the runtime library source code.

1.2.3 VC++ MFC and Template Libraries

The Microsoft Foundation Classes (the MFC) is an extensive C++ class library designed for creating Windows GUI (graphical user interface) programs. The MFC simplifies writing these programs, and it provides many high-level features that can save you considerable coding effort. Although you can build Windows GUI programs in C or C++ *without* using the MFC.

You can also install the Microsoft Active Template Library (ATL), which is a set of template-based C++ classes that facilitate creating ActiveX controls and other types of COM (Component Object Model) objects. The ATL provides an *alternative* to using the MFC to create COM objects. Objects created using the ATL tend to be smaller and faster than those created using the MFC. However, the ATL doesn't provide the extensive set of built-in features or the ease of programming that the MFC offers.

1.2.4 VC++ Build Tools

This component of Visual C++ consists of the optimizing C/C++ compiler, the incremental linker, the resource compiler (for preparing program resources such as menus and dialog boxes), and the other tools required to generate 32-bit Windows programs. You generally run these tools *through* the Microsoft Developer Studio.

1.2.5 ActiveX

This component installs ActiveX controls that you can add to the Windows programs you create using the Microsoft Foundation Classes library. ActiveX controls are reusable software components that can perform a wide variety of tasks.

1.2.6 Data Access

The Data Access component includes database drivers, controls, and other tools that are used by Visual C++, and that allow you to develop Windows database programs. Although database programming isn't covered in this book, you must select those Data Access subcomponents that are initially selected because they form an essential part of Visual C++ (if you deselect any of them, Setup displays a warning).

1.2.7 Enterprise Tools

This component consists of the following enterprise tools:

- Microsoft Visual SourceSafe 6.0 Client
- Application Performance Explorer
- Repository
- Visual Component Manager
- Self-installing .exe Redistributable Files
- Visual Basic Enterprise Components
- VC++ Enterprise Tools
- Microsoft Visual Modeler
- Visual Studio Analyzer

1.2.8 Graphics

This component consists of graphics elements (metafiles, bitmaps, cursors and icons) as well as video clips that you can add to your programs.

1.2.9 Tools

The tools component of Visual C++ comprises the following supplemental development tools:

- API Text Viewer
- MS Info
- MFC Trace Utility
- Spy++
- Win 32 SDK Tools
- OLE/Com Object Viewer
- ActiveX Control Test Container
- VC Error Lookup

1.3 THE VISUAL C++ DEVELOPMENT ENVIRONMENT

Before you begin your quick tour around the Visual C++ development environment, you should start Visual C++ on your computer so that you can see firsthand how each of the areas are arranged and how you can change and alter that arrangement yourself.

After Developer Studio (the Microsoft Visual development environment) starts, you see a window that looks like Figure 1.1. Each of the areas has a specific purpose in the Developer Studio environment. You can rearrange these areas to customize the Developer Studio environment so that it suits your particular development needs.

1.3.1 The Workspace

When you start Visual C++ for the first time, an area on the left side of Developer Studio looks like it is taking up a lot of real estate and providing little to show for it. This area is known as the workspace, and it is your key to navigating the various pieces and parts of your development projects. The workspace allows you to view the parts of your application in three different ways:

- Class View allows you to navigate and manipulate your source code on a C++ class level.
- Resource View allows you to find and edit each of the various resources in your application, including dialog window designs, icons and menus.
- File View allows you to view and navigate all the files that make up your application.

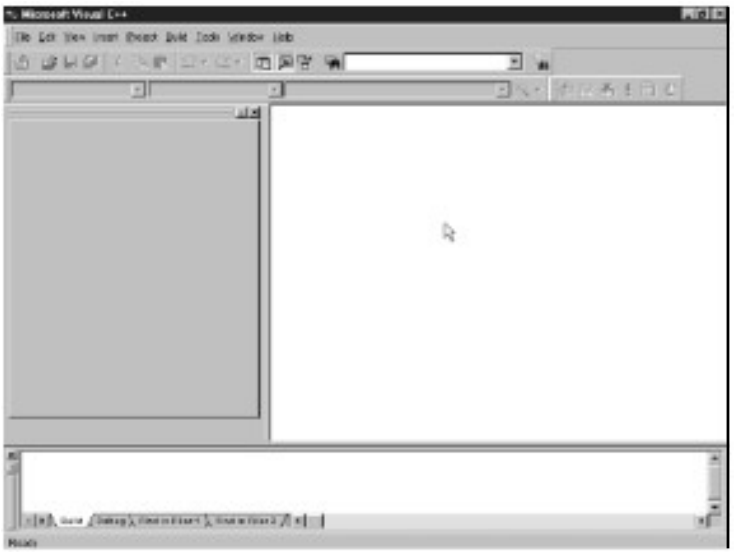


Figure 1.1: The visual C++ opening screen

1.3.2 The Output Pane

The Output pane might not be visible when you start Visual C++ for the first time. After you compile your first application, it appears at the bottom of the Developer Studio environment and remains open until you choose to close it. The Output pane is where Developer Studio provides any information that it needs to give you; where you see all the compiler progress statements, warnings, and error messages; and where the Visual C++ debugger displays all the variables with their current values as you step through your code. After you close the Output pane, it reopens itself when Visual C++ has any message that it needs to display for you.

1.3.3 The Editor Area

The area on the right side of the Developer Studio environment is the editor area. This is the area where you perform all your editing when using Visual C++, where the code editor windows display when you edit C++ source code, and where the window painter displays when you design a dialog box. The editor area is even where the icon painter displays when you design the icons for use in your applications. The editor area is basically the entire Developer Studio area that is not otherwise occupied by panes, menus or toolbars.

1.3.4 Menu Bars

The first time you run Visual C++, three toolbars display just below the menu bar. Many other toolbars are available in Visual C++, and you can customize and create your own toolbars to accommodate how you best work. The three toolbars that are initially open are the following:

- The Standard toolbar contains most of the standard tools for opening and saving files, cutting, copying, pasting, and a variety of other commands that you are likely to find useful.
- The WizardBar toolbar enables you to perform a number of Class Wizard actions without opening the Class Wizard.
- The Build minibar provides you with the build and run commands that you are most likely to use as you develop and test your applications. The full Build toolbar also lets you switch between multiple build configurations (such as between the Debug and Release build configurations).

1.3.5 Rearranging the Developer Studio Environment

The Developer Studio provides two easy ways to rearrange your development environment. The first is by right-clicking your mouse over the toolbar area. This action opens the pop-up menu shown in Figure 1.2, allowing you to turn on and off various toolbars and panes.

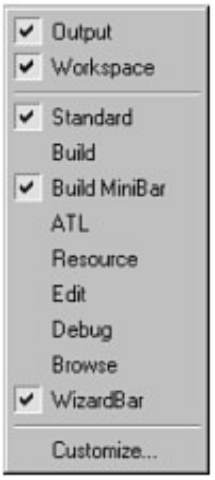


Figure 1.2: Toolbar on and off menu

Another way that you can easily rearrange your development environment is to grab the double bars at the left end of any of the toolbars or panes with the mouse. You can drag the toolbars away from where they are currently docked, making them floating toolbars, as in Figure 1.3. You can drag these toolbars (and panes) to any other edge of the Developer Studio to dock them in a new spot. Even when the toolbars are docked, you can use the double bars to drag the toolbar left and right to place the toolbar where you want it to be located.

Figure 1.3: Example of a floating minibar

> **Note:** On the workspace and Output panes, the double bars that you can use to drag the pane around the Developer Studio environment might appear on the top of the pane or on the left side, depending on how and where the pane is docked.

1.3.6 Starting Your First Project

For your first Visual C++ application, you are going to create a simple application that presents the user with two buttons as in Figure 1.4. The first button will present the user with a simple greeting message, shown in Figure 1.5, and the second button will close the application. In building this application, you will need to do the following things:

1. Create a new project workspace.
2. Use the Application Wizard to create the application framework.
3. Rearrange the dialog that is automatically created by the Application Wizard to resemble how you want the application to look.
4. Add the C++ code to show the greeting to the user.
5. Create a new icon for the application.

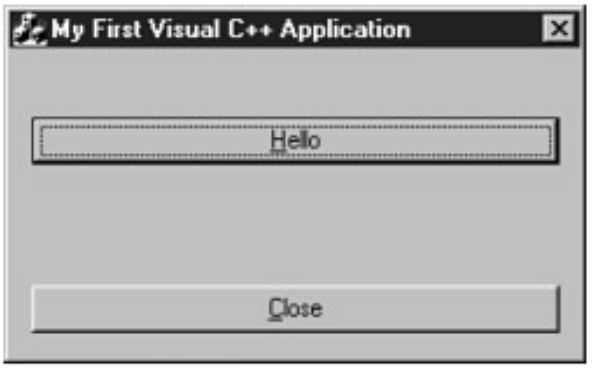


Figure 1.4: Your first Visual C++ application

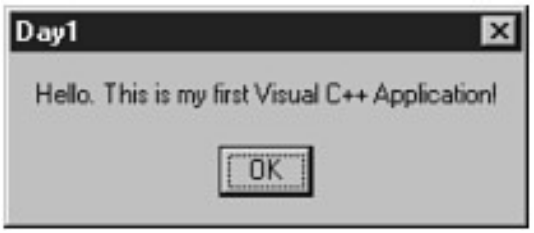


Figure 1.5: If the user clicks the first button, a simple greeting is shown

1.3.7 Creating the Project Workspace

Every application development project needs its own project workspace in Visual C++. The workspace includes the directories where the application source code is kept, as well as the directories where the various build configuration files are located. You can create a new project workspace by following these steps:

1. Select File | New. This opens the New Wizard shown in Figure 1.6.
2. On the Projects tab, select MFC AppWizard (exe).
3. Type a name for your project, such as Hello, in the Project Name field.
4. Click OK. This causes the New Wizard to do two things:Create a project directory (specified in the Location field) and then start the AppWizard.

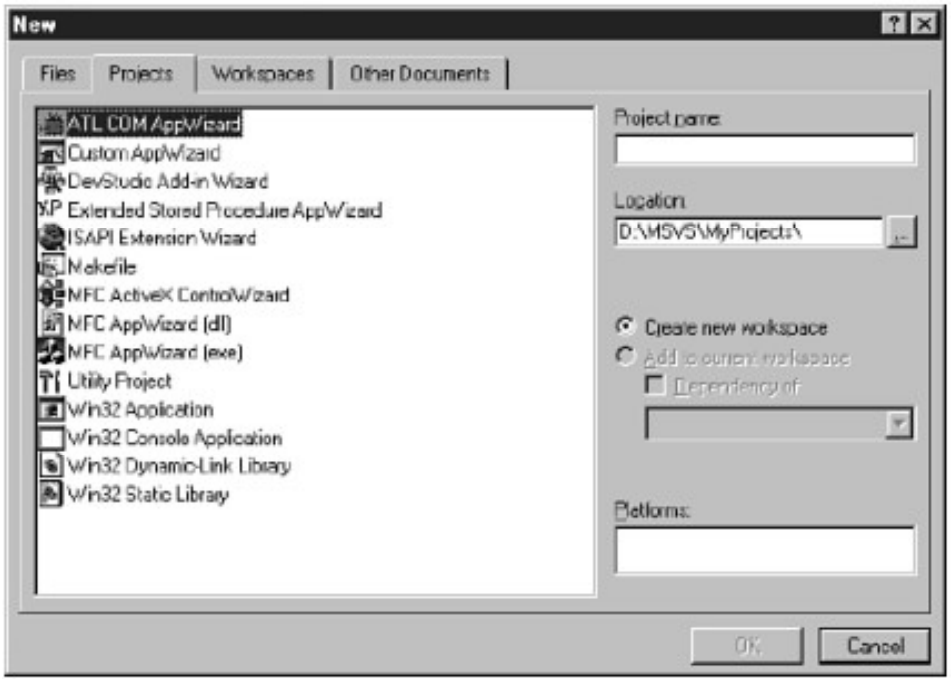


Figure 1.6: The New Wizard

1.3.8 Using the Application Wizard to Create the Application Shell

The AppWizard asks you a series of questions about what type of application you are building and what features and functionality you need. It uses this information to create a shell of an application that you can immediately compile and run. This shell provides you with the basic infrastructure that you need to build your application around. You will see how this works as you follow these steps:

1. In Step 1 of the AppWizard, specify that you want to create a Dialog-based application. Click Next at the bottom of the wizard.
2. In Step 2 of the AppWizard, the wizard asks you about a number of features that you can include in your application. You can uncheck the option for including support for ActiveX controls if you will not be using any ActiveX controls in your application. Because you won't be using any ActiveX controls in today's application, go ahead and uncheck this box.
3. In the field near the bottom of the wizard, delete the project name (Hello) and type in the title that you want to appear in the title bar of the main application window, such as My First Visual C++ Application. Click Next at the bottom of the wizard.

4. In Step 3 of the AppWizard, leave the defaults for including source file comments and using the MFC library as a DLL. Click Next at the bottom of the wizard to proceed to the final AppWizard step.
5. The final step of the AppWizard shows you the C++ classes that the AppWizard will create for your application. Click Finish to let AppWizard generate your application shell.
6. Before AppWizard creates your application shell, it presents you with a list of what it is going to put into the application shell, as shown in Figure 1.7, based on the options you selected when going through the AppWizard. Click OK and AppWizard generates your application.

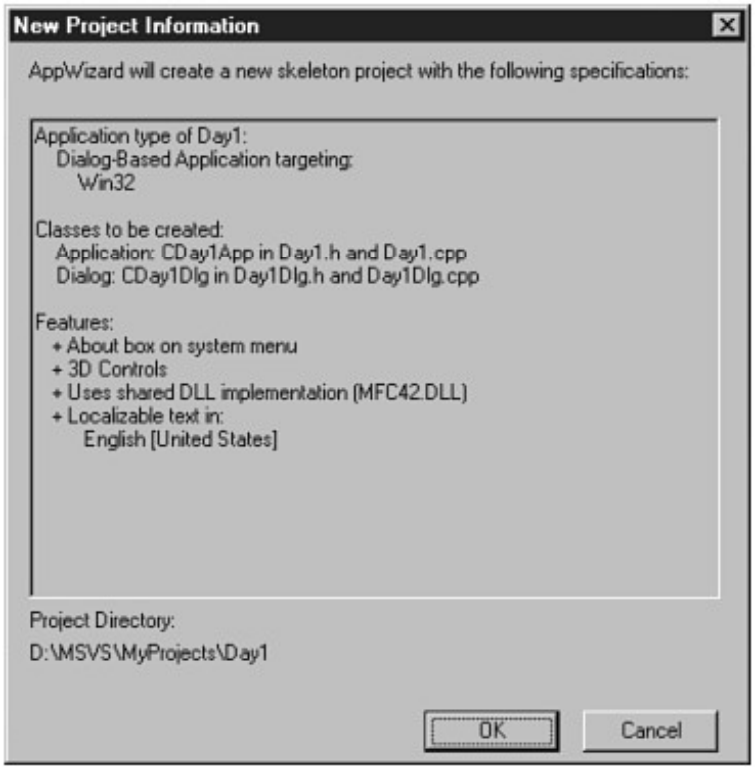


Figure 1.7: The New Project Information screen

7. After the AppWizard generates your application shell, you are returned to the Developer Studio environment. You will notice that the workspace pane now presents you with a tree view of the classes in your application shell, as in Figure 1.8. You might also be presented with the main dialog window in the editor area of the Developer Studio area.

8. Select Build | Build Hello.exe to compile your application.
9. As the Visual C++ compiler builds your application, you see progress and other compiler messages scroll by in the output pane. After your application is built, the output pane should display a message telling you that there were no errors or warnings, as in Figure 1.9.

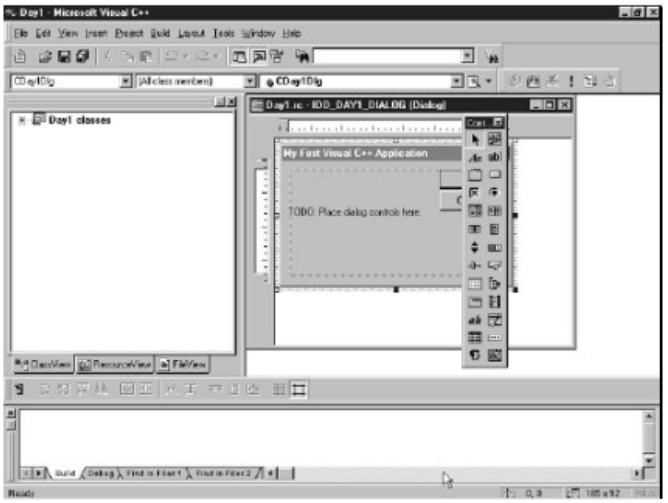


Figure 1.8: Your workspace with a tree view of the project's classes

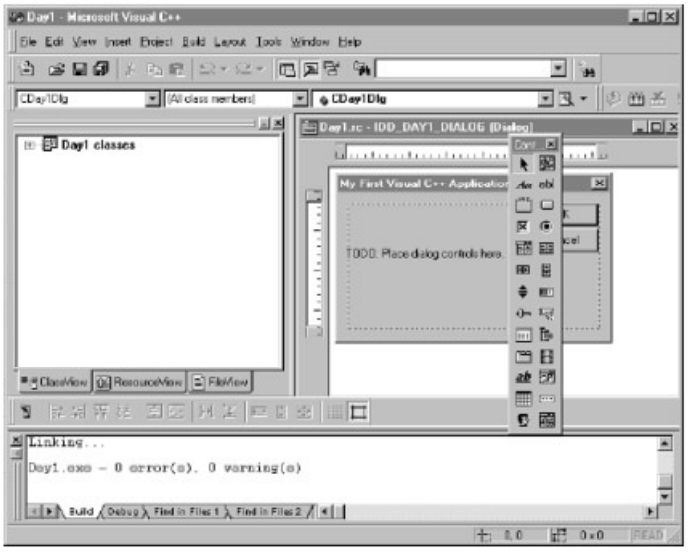


Figure 1.9: The output pane displays any compiler errors

10. Select Build | Execute Hello.exe to run your application.
11. Your application presents a dialog with a TODO message and OK and Cancel buttons, as shown in Figure 1.10. You can click either button to close the application.

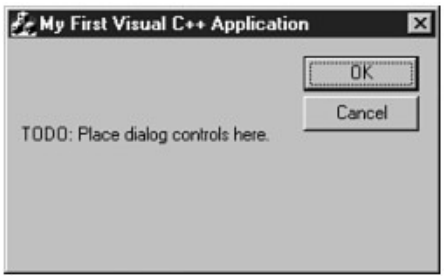


Figure 1.10: The unmodified application shell

1.3.9 Designing Your Application Window

Now that you have a running application shell, you need to turn your focus to the window layout of your application. Even though the main dialog window may already be available for painting in the editor area, you should still navigate to find the dialog window in the workspace so that you can easily find the window in subsequent development efforts. To redesign the layout of your application dialog, follow these steps:

1. Select the Resource View tab in the workspace pane, as in Figure 1.11.

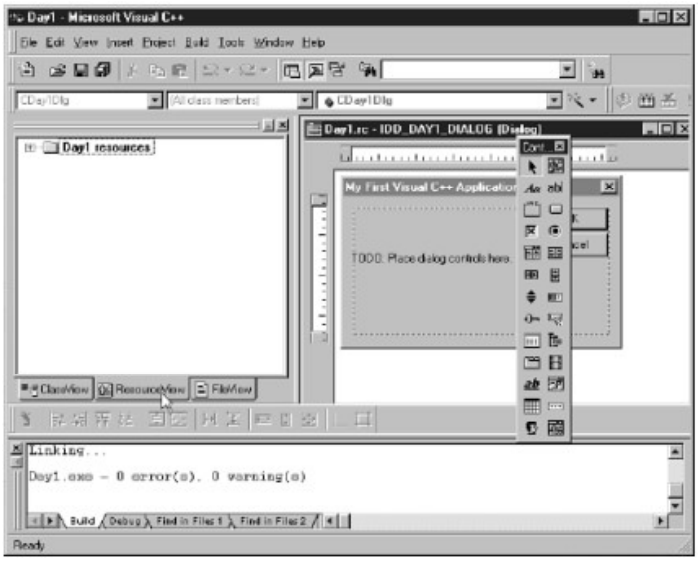


Figure 1.11: The resource view tab in the workspace pane

2. Expand the resources tree to display the available dialogs. At this point, you can double click the IDD_DAY1_DIALOG dialog to open the window in the Developer Studio editor area.
3. Select the text displayed in the dialog and delete it using the Delete key.
4. Select the Cancel button, drag it down to the bottom of the dialog and resize it so that it is the full width of the layout area of the window, as in Figure 1.12.

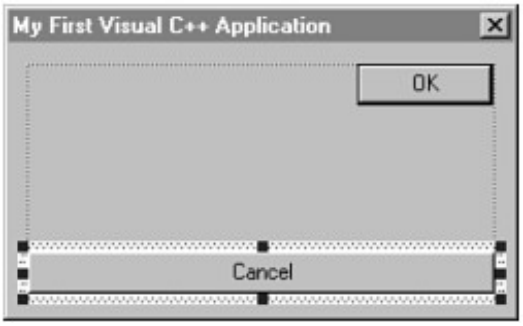


Figure 1.12: Positioning the Cancel button

5. Right-click the mouse over the Cancel button, opening the pop-up menu in Figure 1.13. Select Properties from the menu, and the properties dialog in Figure 1.14 opens.

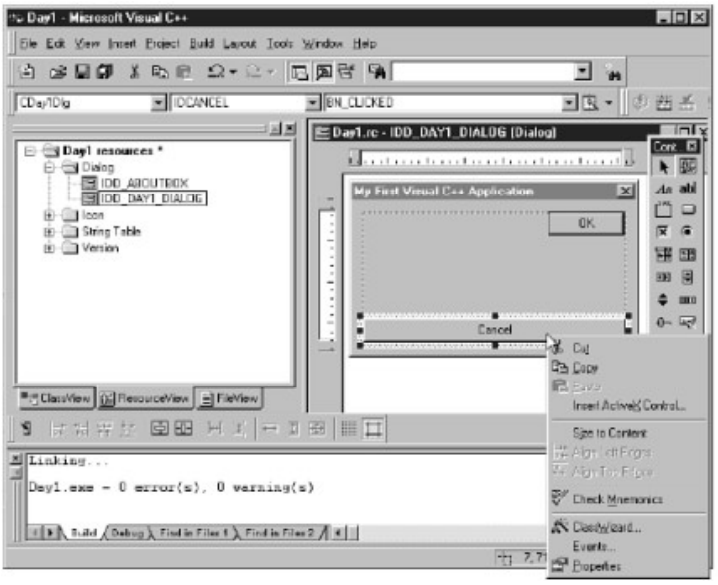


Figure 1.13: Right-clicking the mouse to open a pop-up menu

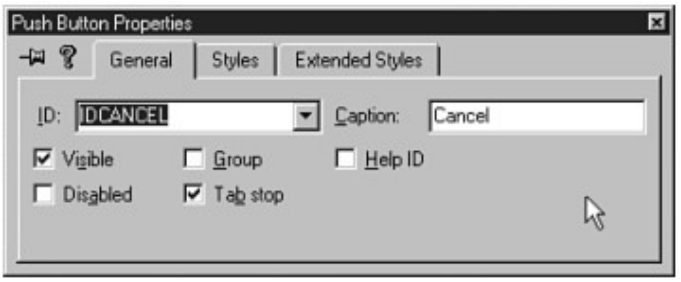


Figure 1.14: The Cancel button properties dialog

6. Change the value in the Caption field to &Close. Close the properties dialog by clicking the Close icon in the upper-right corner of the dialog.
7. Move and resize the OK button to around the middle of the window, as in Figure 1.15.

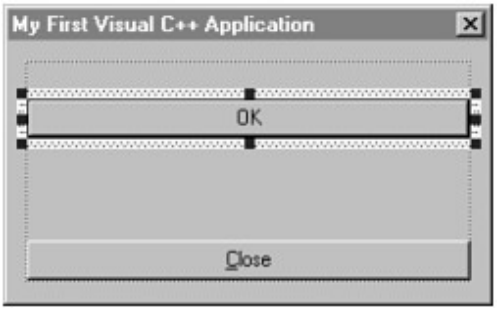


Figure 1.15: Positioning the OK button

8. On the OK button properties dialog, change the ID value to IDHELLO and the caption to &Hello.
9. Now when you compile and run your application, it will look like what you've just designed, as shown in Figure 1.16.

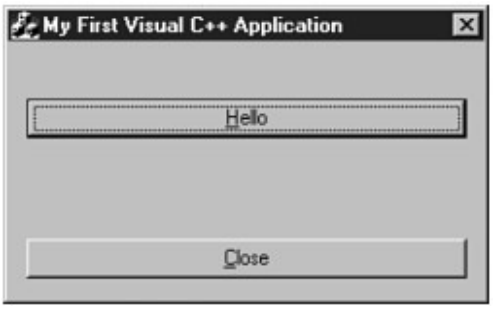


Figure 1.16: Running your redesigned application

Note: If you play with your application, you will notice that the Close button still closes the application. However, the Hello button no longer does anything because you changed the ID of the button. MFC applications contain a series of macros in the source code that determine which functions to call based on the ID and event message of each control in the application. Because you changed the ID of the Hello button, these macros no longer know which function to call when the button is clicked.

1.3.10 Adding Code to Your Application

You can attach code to your dialog through the Visual C++ Class Wizard. You can use the Class Wizard to build the table of Windows messages that the application might receive, including the functions they should be passed to for processing, that the MFC macros use for attaching functionality to window controls. You can attach the functionality for this first application by following these steps:

1. To attach some functionality to the Hello button, right-click over the button and select Class Wizard from the pop-up menu.
2. If you had the Hello button selected when you opened the Class Wizard, it is already selected in the list of available Object IDs, as in Figure 1.17.

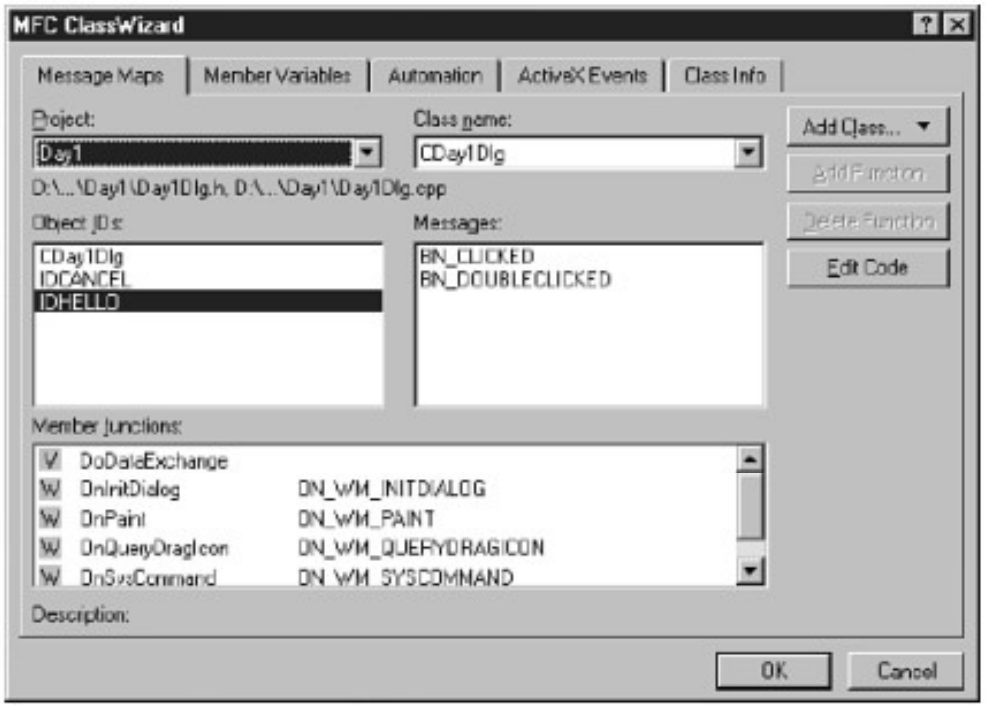


Figure 1.17: The Class Wizard

3. With IDHELLO selected in the Object ID list, select BN_CLICKED in the list of messages and click Add Function. This opens the Add Member Function dialog shown in Figure 1.18. This dialog contains a suggestion for the function name. Click OK to create the function and add it to the message map.

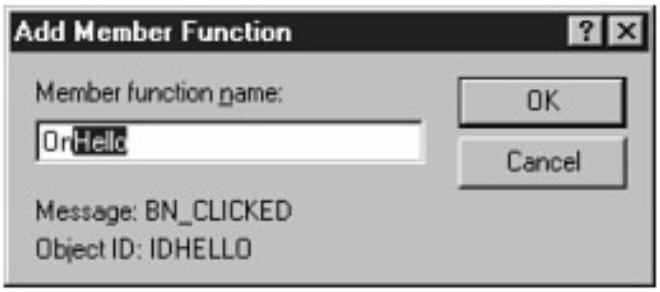


Figure 1.18: The Class Wizard Add Member Function dialog

4. After the function is added for the click message on the Hello button, select the OnHello function in the list of available functions, as in Figure 1.19. Click the Edit Code button so that your cursor is positioned in the source code for the function, right at the position where you should add your functionality.

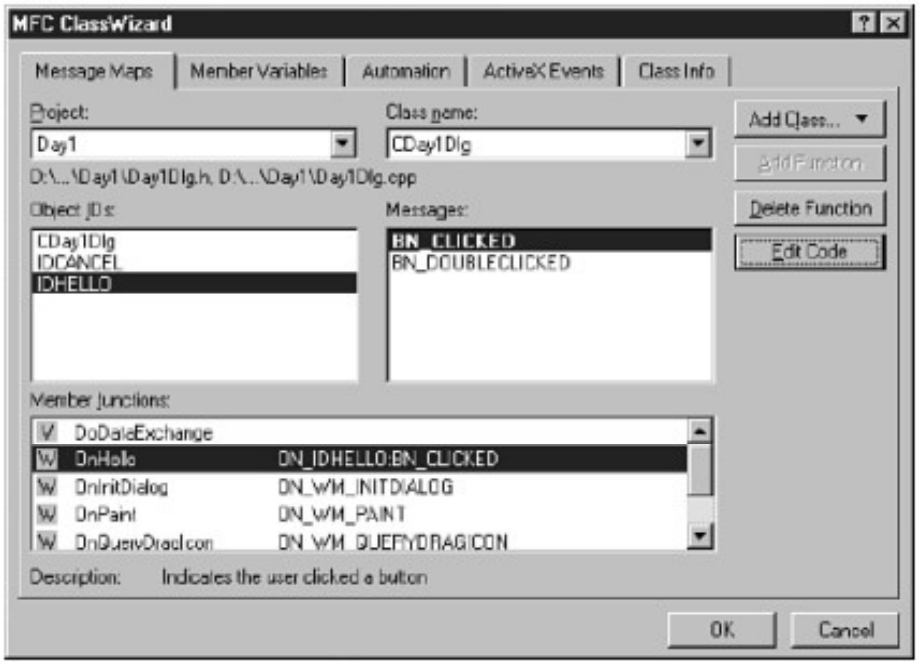


Figure 1.19: The list of available functions in the Class Wizard

5. Add the code in Listing 1.1 just below the TODO comment line, as shown in Figure 1.20.

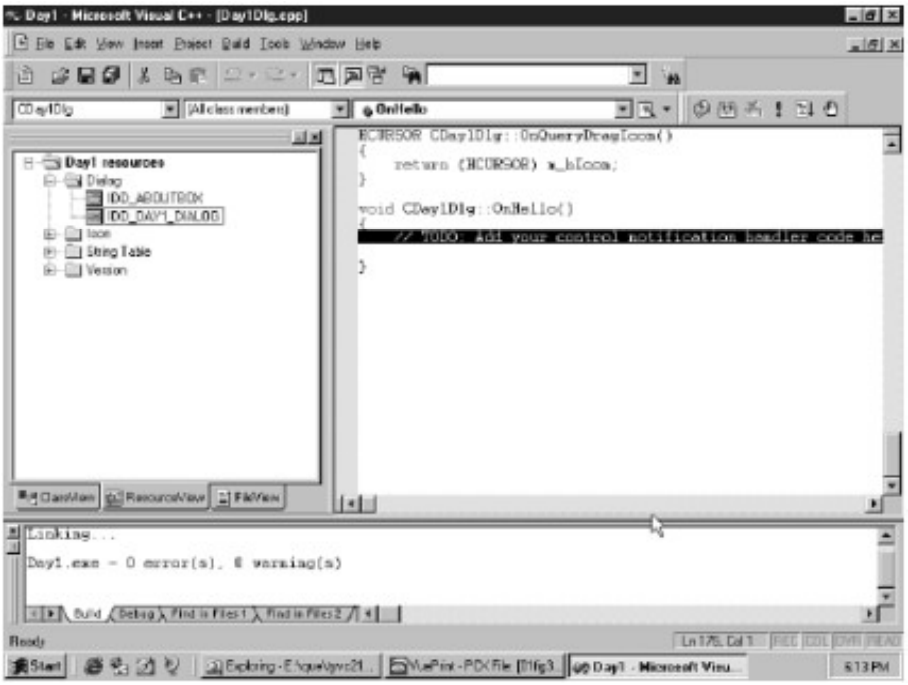


Figure 1.20: Source code view where you insert Listing 1.1

```
1: Void CHelloDlg::OnHello()
2: {
3:     // TODO: Add your control notification handler code here
4:
5:     ////////////////////////
6:     // MY CODE STARTS HERE
7:     ////////////////////////
8:
9:     // Say hello to the user
10:    MessageBox("Hello. This is my first Visual C++ Application!");
```

```
11:
12:     //////////////////////
13:    // MY CODE ENDS HERE
14:     //////////////////////
15: }
```

6. When you compile and run your application, the Hello button should display the message shown in Figure 1.21.

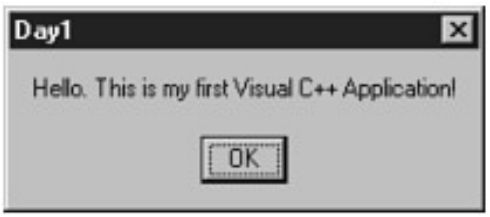


Figure 1.21: Now your application will say hello to you

REVIEW EXERCISE

1. How one can change the caption of a button?
2. What can to do with C++ AppWizard?
3. What all events can be associated with a button?
4. What exactly is Microsoft Visual C++ all about?
5. What are MFCs?
6. Visual C++ is object oriented. Comment.
7. What is included in Visual C++?
8. Describe the workspace in Visual C++.
9. Write a simple application that displays a message box on the screen.

CHAPTER 2

CLASS IN C++

2.1 FUNDAMENTALS

Object-oriented programming (OOPS) is a programming paradigm that uses 'objects' and their interactions to design applications and computer programs. Programming techniques may include features such as encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s. Many modern programming languages now support OOPS.

Following are the fundamental concepts in OOPS:

2.1.1 Class

Defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something. For example, the class Dog would consist of traits shared by all dogs, such as breed and fur color (characteristics), and the ability to bark and sit (behaviors). Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation). Collectively, the properties and methods defined by a class are called members.

2.1.2 Object

An object doesn't exist until an instance of the class has been created; the class is just a definition. When the object is physically created, space for that object is allocated in RAM. It is possible to have multiple objects created from one class. It can be considered as a pattern (exemplar) of a class. The class of Dog defines all possible dogs by listing the characteristics and behaviour they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur.

2.1.3 Instance

One can have an instance of a class or a particular object. The instance is the actual object created at runtime. In programmer jargon, the Lassie object is an instance of the Dog class. The set of values of the attributes of a particular object is called its state. The object consists of state and the behaviour that's defined in the object's class.

2.1.4 Method

In language, methods (sometimes referred to as 'functions') are verbs. Lassie, being a Dog, has the ability to bark. So bark() is one of Lassie's methods. She may have other methods as well, for example sit() or eat() or walk() or save_timmy(). Within the program, using a method usually affects only one particular object; all Dogs can bark, but you need only one particular dog to do the barking.

2.1.5 Message Passing

"The process by which an object sends data to another object or asks the other object to invoke a method." Also known to some programming languages as interfacing. For example, the object called Breeder may tell the Lassie object to sit by passing a 'sit' message which invokes Lassie's 'sit' method.

In the terminology of object-oriented programming languages, a message is the single means to pass control to an object. If the object 'responds' to the message, it has a method for that message. In pure object-oriented programming, message passing is performed exclusively through a dynamic dispatch strategy. Sending the same message to an object twice will usually result in the object applying the method twice. Two messages are considered to be the same message type, if the name and the arguments of the message are identical. Objects can send messages to other objects from within their method bodies. Message passing enables extreme late binding in systems.

Alan Kay has argued that message passing is a concept more important than objects in his view of object-oriented programming, however people often miss the point and place too much emphasis on objects themselves and not enough on the messages being

sent between them. The syntax varies between languages, for example, In Java code-level message passing corresponds to "method calling". Some dynamic languages use double-dispatch or multi-dispatch to find and pass messages.

2.1.6 Inheritance

Inheritance is the mechanism whereby specific classes are made from more general ones. The child or derived class inherits all the features of its parent or base class, and is free to add features of its own. In addition, this derived class may be used as the base class of an even more specialized class. Inheritance, or derivation, provides a clean mechanism whereby common classes can share their common features, rather than having to rewrite them. For example, consider a graph class which is represented by edges and vertices and some (abstract) method of traversal. Next, consider a tree class which is a special form of a graph. We can simply derive tree from graph and the tree class automatically inherits the concept of edges, vertices and traversal from the graph class. We can then restrict how edges and vertices are connected within the tree class so that it represents the true nature of a tree. Inheritance is supported in C++ by placing the name of the base class after the name of the derived class when the derived class is declared. It should be noted that a standard conversion occurs in C++ when a pointer or reference to a base class is assigned a pointer or reference to a derived class.

2.1.7 Multiple Inheritance

It is a type of inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other. For example, independent classes could define Dogs and Cats, and a Chimera object could be created from these two which inherits all the (multiple) behavior of cats and dogs. This is not always supported, as it can be hard both to implement and to use well.

2.1.8 Abstraction

Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. For example, Lassie the Dog may be treated as a Dog much of the time, a Collie when necessary to access Collie-specific attributes or behaviors, and as an Animal (perhaps the parent class of Dog) when counting Timmy's pets.

Abstraction is also achieved through Composition. For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, *i.e.*, send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

2.1.9 Encapsulation

Data encapsulation, sometimes referred to as data hiding, is the mechanism whereby the implementation details of a class are kept hidden from the user. The user can only perform a restricted set of operations on the hidden members of the class by executing special functions commonly called *methods*. The actions performed by the methods are determined by the designer of the class, who must be careful not to make the methods either overly flexible or too restrictive. This idea of hiding the details away from the user and providing a restricted, clearly defined interface is the underlying theme behind the concept of an *abstract data type*.

The advantage of using data encapsulation comes when the *implementation* of the class changes but the *interface* remains the same. For example, to create a stack class which can contain integers, the designer may choose to implement it with an array, which is hidden from the user of the class. The designer then writes the push() and pop() methods which puts integers into the array and removes them from the array respectively. These methods are made accessible to the user. Should an attempt be made by the user to access the array directly, a compile time error will result. Now, should the designer decide to change the stack's implementation to a linked list, the array can simply be replaced with a linked list and the push() and pop() methods rewritten so that they manipulate the linked list instead of the array. The code which the user has written to manipulate the stack is still valid because it was not given direct access to the array to begin with.

The concept of data encapsulation is supported in C++ through the use of the public, protected and private keywords which are placed in the declaration of the class. Anything in the class placed after the public keyword is accessible to all the users of the class; elements placed after the protected keyword are accessible only to the methods of the class or classes derived from that class; elements placed after the private keyword are accessible only to the methods of the class.

2.1.10 Polymorphism

Polymorphism allows the programmer to treat derived class members just like their parent class' members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations. If a Dog is commanded to speak(), this may elicit a bark(). However, if a Pig is commanded to speak(), this may elicit an oink(). They both inherit speak() from Animal, but their derived class methods override the methods of the parent class; this is Overriding Polymorphism. Overloading Polymorphism is the use of one method signature, or one operator such as '+', to perform several different functions depending on the implementation. The '+' operator, for example, may be used to perform integer addition, float addition, list concatenation, or string concatenation.

Any two subclasses of Number, such as Integer and Double, are expected to add together properly in an OOPS language. The language must therefore overload the addition operator, '+', to work this way. This helps improve code readability. How this is implemented varies from language to language, but most OOPS languages support at least some level of overloading polymorphism. Many OOPS languages also support Parametric Polymorphism, where code is written without mention of any specific type and thus can be used transparently with any number of new types. Pointers are an example of a simple polymorphic routine that can be used with many different types of objects.

2.1.11 Decoupling

Decoupling allows for the separation of object interactions from classes and inheritance into distinct layers of abstraction. A common use of decoupling is to polymorphically decouple the encapsulation, which is the practice of using reusable code to prevent discrete code modules from interacting with each other. Not all of the above concepts are to be found in all object-oriented programming languages, and so object-oriented programming that uses classes is called sometimes class-based programming. In particular, prototype-based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of object and instance.

2.1.12 Dynamic Binding of Function Calls

Dynamic binding is one of the main features of polymorphism. Quite often when using inheritance, one will discover that a series of classes share a common behaviour, but how that behaviour is implemented is different from class to class. Such a situation is a prime candidate for the use of dynamic or runtime binding which is also referred to as *polymorphism.*

2.2 CLASS DEFINITION

A class is an expanded concept of a data structure: Instead of holding only data, it can hold both data and functions. An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable. A class is a definition of an object. It's a type just like int. A class resembles a struct with just one difference: all struct members are public by default. All class members are private.

Remember: A class is a type, and an object of this class is just a variable. Before we can use an object, it must be created. The simplest definition of a class is

```
class name {
// members
}
```

Classes are generally declared using the keyword **class**, with the following format:

```
class class_name {
 access_specifier_1:
  member1;
 access_specifier_2:
  member2;
 …
 } object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members that can be either data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called access specifier. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

- Private members of a class are accessible only from within other members of the same class or from their friends.
- Protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

```
class CRectangle {
   int x, y;
public:
   void set_values (int,int);
   int area (void);
} rect;
```

Declares a class (i.e., a type) called *CRectangle* and an object (i.e., a variable) of this class called *rect.* This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition. Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

```
int a;
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

```
// classes example                    area: 12
#include <iostream>
using namespace std;

class CRectangle {
   int x, y;
 public:
   void set_values (int,int);
   int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
   x = a;
   y = b;
}
int main () {
   CRectangle rect;
   rect.set_values (3,4);
   cout << "area: " << rect.area();
   return 0;
}
```

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class CRectangle, we could have declared the object rectb in addition to the object rect:

```
// example: one class, two objects          rect area: 12
#include <iostream>                         rectb area: 30
using namespace std;

class CRectangle {
   int x, y;
 public:
   void set_values (int,int);
   int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
```

```
    x = a;
    y = b;
}
int main () {
    CRectangle rect, rectb;  rect.set_values (3,4);
    rectb.set_values (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

In this concrete case, the class (type of the objects) to which we are talking about is CRectangle, of which there are two instances or objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of object-oriented programming: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

Classes vs. Structures

Classes and structures are syntactically similar. In C++, the role of the structure was expanded, making it an alternative way to specify a class. In C, the structures include data members, in C++ they are expanded to have function members as well. This makes structures in C++ and classes to be virtually same. The only difference between a C++ struct and a class is that, by default all the struct members are public while by default class members are private.

2..3 CONSTRUCTORS AND DESTRUCTORS

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set_values()?

Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor:

```
// example: class constructor                rect area: 12
#include <iostream>                          rectb area: 30
using namespace std;
class CRectangle {
  int width, height;
public:
  CRectangle (int,int);
  int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}
int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition includes a return value; not even void.

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and destructors                    rect area: 12
#include <iostream>                                           rectb area: 30
using namespace std;

class CRectangle {
   int *width, *height;
public:
   CRectangle (int,int);
   ~CRectangle ();
   int area () {return (*width * *height);}
};
CRectangle::CRectangle (int a, int b) {
   width = new int;
   height = new int;
   *width = a;
   *height = b;
}
CRectangle::~CRectangle () {
   delete width;
   delete height;
}
int main () {
   CRectangle rect (3,4), rectb (5,6);
```

```
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

2.3.1 Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
// overloading class constructors
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};
CRectangle::CRectangle () {
    width = 5;
    height = 5;
}
CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}
int main () {
    CRectangle rect (3,4);
    CRectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

```
rect area: 12
rectb area: 25
```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

2.3.2 Default Constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
class CExample {
public:
  int a,b,c;
  void multiply (int n, int m) { a=n; b=m; c=a*b; };
};
```

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class CExample {
public:
  int a,b,c;
  CExample (int n, int m) { a=n; b=m; };
  void multiply () { c=a*b; };
};
```

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own. These are the copy constructor, the copy assignment operator, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv) {
  a=rv.a;  b=rv.b;  c=rv.c;
}
```

Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);
CExample ex2 (ex);   // copy constructor (data copied from ex)
```

2.3.3 Copy Constructors

Copy constructor is

- a constructor function with the same name as the class
- used to make deep copy of objects.

There are 3 important places where a copy constructor is called.

1. When an object is created from another object of the same type.
2. When an object is passed by value as a parameter to a function.
3. When an object is returned from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. This will ensure a shallow copy. If the class does not have pointer variables with dynamically allocated memory, then one need not worry about defining a copy constructor. It can be left to the compiler's discretion. But if the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

For example:

```
class A   //Without copy constructor
{
  private:
  int x;
  public:
```

```
    A() {A = 10;}
    ~A() {}
}
class B    //With copy constructor
{
    private:
    char *name;
    public:
    B()
    {
    name = new char[20];
    }
    ~B()
    {
    delete name[];
    }
  //Copy constructor
    B(const B &b)
    {
    name = new char[20];
    strcpy(name, b.name);
    }
};
```

Let us imagine if you don't have a copy constructor for the class B. At the first place, if an object is created from some existing object, we cannot be sure that the memory is allocated. Also, if the memory is deleted in destructor, the delete operator might be called twice for the same memory location.

This is a major risk. One happy thing is, if the class is not so complex this will come to the fore during development itself. But if the class is very complicated, then these kind of errors will be difficult to track.

When Copies of Objects are Made

A *copy constructor* is called whenever a new variable is created from an object. This happens in the following cases (but not in assignment).

- A variable is declared which is *initialized from another object*, e.g.,
- Person q("Mickey"); // constructor is used to build q.

- Person r(p); // copy constructor is used to build r.
- Person p = q; // copy constructor is used to initialize in declaration.
  p = q; // Assignment operator, no constructor or copy constructor.
- A value parameter is initialized from its corresponding argument.
  f(p); // copy constructor initializes formal value parameter.
- An object is returned by a function.

C++ calls a *copy constructor* to make a copy of an object in each of the above cases. If there is no copy constructor defined for the class, C++ uses the default copy constructor which copies each field, i.e., makes a *shallow copy*.

Don't Write a Copy Constructor if Shallow Copies are Ok

If the object has no pointers to dynamically allocated memory, a shallow copy is probably sufficient. Therefore the default copy constructor, default assignment operator, and default destructor are ok and you don't need to write your own.

If you need a copy constructor, you also need a destructor and operator=

If you need a copy constructor, it's because you need something like a deep copy, or some other management of resources. Thus, it is almost certain that you will need a destructor and override the assignment operator.

Copy constructor syntax

The copy constructor takes a reference to a const parameter. It is const to guarantee that the copy constructor doesn't change it and it is a reference because a value parameter would require making a copy, which would invoke the copy constructor, which would make a copy of its parameter, which would invoke the copy constructor, which...

Here is an example of a copy constructor for the Point class, which doesn't really need one because the default copy constructor's action of copying fields would work fine, but it shows how it works.

```
//=== file Point.h ========================================
class Point {
   public:
      . . .
      Point(const Point& p);   // copy constructor
      . . .
//=== file Point.cpp ======================================
. . .
```

```
Point::Point(const Point& p) {
   x = p.x;
   y = p.y;
}
   . . .
//=== file my_program.cpp ================================
. . .
Point p;          // calls default constructor
Point s = p;      // calls copy constructor.
p = s;            // assignment, not copy constructor.
```

Difference between copy constructor and assignment

A copy constructor is used to initialize a *newly declared* variable from an existing variable. This makes a deep copy like assignment, but it is somewhat simpler:

1. There is no need to test to see if it is being initialized from itself.
2. There is no need to clean up (*e.g.*, delete) an existing value (there is none).
3. A reference to itself is not returned.

2.4 POINTERS TO CLASSES

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example,

```
CRectangle * prect;
```

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream>
using namespace std;

class CRectangle {
   int width, height;
public:
   void set_values (int, int);
```

```
    int area (void) {return (width * height);}
};
void CRectangle::set_values (int a, int b) {
    width = a;
    height = b;
}
int main () {
    CRectangle a, *b, *c;
    CRectangle * d = new  CRectangle[2];
    b= new CRectangle;
    c= &a;
    a.set_values (1,2);
    b->set_values (3,4);
    d->set_values (5,6);
    d[1].set_values (7,8);
    cout << "a area: " << a.area() << endl;
    cout << "*b area: " << b->area() << endl;
    cout << "*c area: " << c->area() << endl;
    cout << "d[0] area: " << d[0].area() << endl;
    cout << "d[1] area: " << d[1].area() << endl;
    delete[] d;
    delete b;
    return 0;
}
```

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

expression	can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Be sure that you understand the logic under all of these expressions before proceeding with the next sections. If you have doubts, read again this section and/or consult the previous sections about pointers and data structures.

2.5 CLASSES DEFINED WITH STRUCT AND UNION

Classes can be defined not only with keyword class, but also with keywords struct and union. The concepts of class and data structure are so similar that both keywords (struct and class) can be used in C++ to declare classes (i.e., structs can also have function members in C++, not only data members). The only difference between both is that members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access. For all other purposes both keywords are equivalent. The concept of unions is different from that of classes declared with struct and class, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold function members. The default access in union classes is public.

2.6 THE KEYWORD THIS

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.

One of its uses can be to check if a parameter passed to a member function is the object itself. For example,

```
// this                                    yes, &a is b
#include <iostream>
using namespace std;

class CDummy {
  public:
    int isitme (CDummy& param);
};
int CDummy::isitme (CDummy& param)
{
  if (&param == this) return true;
  else return false;
}
int main () {
  CDummy a;
```

```
    CDummy* b = &a;
    if ( b->isitme(a) )
      cout << "yes, &a is b";
    return 0;
}
```

It is also frequently used in operator= member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an operator= function similar to this one:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

In fact, this function is very similar to the code that the compiler generates implicitly for this class if we do not include an operator= member function to copy objects of this class.

2.7 STATIC MEMBERS

A class can contain static members, either data or functions. Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another. For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```
// static members in classes                    7
#include <iostream>                             6
using namespace std;

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n—; };
};
```

```
int CDummy::n=0;
int main () {
  CDummy a;
  CDummy b[5];
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

```
int CDummy::n=0;
```

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
cout << a.n;
cout << CDummy::n;
```

These two calls included in the previous example are referring to the same variable: the static variable n within class CDummy shared by all objects of this class.

Once again, I remind you that in fact it is a global variable. The only difference is its name and possible access restrictions outside its class.

Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keyword this, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.

2.8 INLINE FUNCTIONS

What is Inline Function?

Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Reason for the need of Inline Function

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function saved program space and memory space are used because the function is stored only in one place and is only executed when it is called. This concept of function execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function. In this situation, the programmer may be wondering "why not write the short code repeatedly inside the program wherever needed instead of going for inline function?" Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program. The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

What happens when an inline function is written?

The inline function takes the format as a normal function but when it is compiled it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is compiled, the code present in function body is replaced in the place of function call.

General Format of inline Function

The general format of inline function is as follows:

```
inline datatype function_name(arguments)
```

The keyword inline specified in the above example, designates the function as inline function. For example, if a programmer wishes to have a function named demo with return value as integer and with no arguments as inline it is written as follows:

```
inline int demo( )
```

Example:

The concept of inline functions:

```
#include <iostream.h>
int demo(int);
void main( )
   {
int x;
```

```
cout << "\n Enter the Input Value: ";
cin>>x;
cout<<"\n The Output is: " << demo(x);
   }
inline int demo(int x1)
   {
return 5*x1;
   }
```

The output of the above program is:

```
Enter the Input Value: 10
The Output is: 50
```

The output would be the same even when the inline function is written solely as a function. The concept, however, is different. When the program is compiled, the code present in the inline function demo( ) is replaced in the place of function call in the calling program. The concept of inline function is used in this example because the function is a small line of code.

The above example, when compiled, would have the structure as follows:

```
#include <iostream.h>
int demo(int);
void main( )
   {
int x;
cout << "\n Enter the Input Value: ";
cin>>x;
//The demo(x) gets replaced with code return 5*x1;
cout<<"\n The Output is: " << demo(x);
   }
```

When the above program is written as normal function the compiled code would look like below:

```
#include <iostream.h>
int demo(int);
void main( )
{
```

```
int x;
cout << "\n Enter the Input Value:";
cin>>x;
//Call is made to the function demo
cout<<"\n The Output is:" << demo(x);
}
int demo(int x1){
return 5*x1;
}
```

2.9 ACCESS SPECIFIERS

There are three access specifiers as given by C++.

Private: If data are declared as private in a class then it is accessible by the member functions of the class where they are declared. The private member functions can be accessed only by the members of the class. By default, any member of the class is considered as private by the C++ compiler, if no specifier is declared for the member.

Public: The member functions with public access specifier can be accessed outside of the class. This kind of members is accessed by creating instance of the class.

Protected: Protected members are accessible by the class itself and it's sub-classes. The members with protected specifier act exactly like private as long as they are referenced within the class or from the instance of the class. This specifier specially used when you need to use inheritance facility of C++. The protected members become private of a child class in case of private inheritance, public in case of public inheritance, and stay protected in case of protected inheritance.

With the proper use of access specifier the data can be hidden from unauthorized access.

REVIEW EXERCISE

1. Discuss classes and object along with their significance in OOPs.
2. Discuss the role of constructors in C++ classes.
3. Discuss the types of Constructors in C++.
4. Design a EMP class with three functions: Get_Details, Calculate_Salary, Show_Details. Make use of default and parameterized constructors.
5. Design a class with an offline function that finds the average of N given numbers.
6. How constructors can be overloaded. Show via an example.

CHAPTER

3 OVERLOADING IN C++

C++ allows both functions and operators to be overloaded and hence it includes function and operator overloading.

3.1 FUNCTION OVERLOADING

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

```
//overloaded function
#include
void sum(int a,int b)
{
cout<<A+B<<ENDL;
}
void sum(int a,int b,int c)
{
cout<<A+B+C<<ENDL;
}
```

```
void sum(int a,int b,int c,int d)
{
cout<<A+B+C+D<<ENDL;
}
void sum(int a,int b,int c,int d,int e)
{
cout<<A+B+C+D+E<<ENDL;
}
void main()
{
cout<<"using overloaded function\n";
sum(10,20);//two arguments
sum(10,20,30);//three arguments
sum(10,20,30,40);//four arguments
sum(10,20,30,40,50);//five arguments
}
```

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
// overloaded function
#include <iostream>
using namespace std;
int operate (int a, int b)
{
  return (a*b);
}
float operate (float a, float b)
{
  return (a/b);
}
int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
```

```
    cout << operate (x,y);
    cout << "\n";
    cout << operate (n,m);
    cout << "\n";
    return 0;
}
```

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been overloaded.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Example 1: Overloading Functions that differ in terms of NUMBER OF PARAMETERS

```
//Example Program in C++
#include<iostream.h>
//FUNCTION PROTOTYPES
int func(int i);
int func(int i, int j);
void main(void)
{
cout<<func(10);//func (int i)is called\
cout<<func(10,10);//func(int i, int j) is called
}
int func(int i)
{
return i;
}
int func(int i, int j)
```

```
{
return i+j;
}
```

Example 2: Overloading Functions that differ in terms of TYPE OF PARAMETERS

```
//Example Program in C++
#include<iostream.h>
//FUNCTION PROTOTYPES
int func(int i);
double func(double i);
void main(void)
{
cout<<func(10);//func(int i)is called
cout<<func(10.201);//func(double i) is called
}
int func(int i)
{
return i;
}
double func(double i)
{
return i;
}
```

Example 3: Is the program below, valid?

```
//Example Program in C++
#include<iostream.h>
//FUNCTION PROTOTYPES
int func(int i);
double func(int i);
void main(void)
{
cout<<func(10);
cout<<func(10.201);
}
```

```
int func(int i)
{
return i;
}
double func(int i)
{
return i;
}
```

No, because you can't overload functions if they differ only in terms of the data type they return.

3.2 OPERATOR OVERLOADING

It allows existing C++ operators to be redefined so that they work on objects of user-defined classes. Overloaded operators are syntactic sugar for equivalent function calls. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs).

In computer programming, operator overloading (less commonly known as operator ad-hoc polymorphism) is a specific case of polymorphism in which some or all of operators like +, =, or == have different implementations depending on the types of their arguments. Sometimes the overloadings are defined by the language; sometimes the programmer can implement support for new types.

Operator overloading is useful because it allows the developer to program using notation closer to the target domain and allows user types to look like types built into the language. It can easily be emulated using function calls.

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example,

```
int a, b, c;
a = b + c;
```

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

```
struct {
   string product;
   float price;
} a, b, c;
a = b + c;
```

In fact, this will cause a compilation error, since we have not defined the behavior our class should have with addition operations. However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

Overloadable operators
+ - * / = < > += -= *= /= << >>
<<= >>= == != <= >= ++ — % & ^ ! |
~ &= ^= |= && || %= [] () , ->* -> new
delete new[] delete[]

To overload an operator in order to use it with classes we declare operator functions, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:

Type operator sign (parameters) { /*...*/ }

Here you have an example that overloads the addition operator (+). We are going to create a class to store bidimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bidimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1,1+2) = (4,3).

```
// vectors: overloading operators example
#include <iostream>
using namespace std;
class CVector {
   public:
      int x,y;
      CVector () {};
      CVector (int,int);
      CVector operator + (CVector);
};
CVector::CVector (int a, int b){
   x = a;
   y = b;
}
CVector CVector::operator+ (CVector param) {
   CVector temp;
```

```
    temp.x = x + param.x;
    temp.y = y + param.y;
    return (temp);
  }
  int main () {
    CVector a (3,1);
    CVector b (1,2);
    CVector c;
    c = a + b;
    cout << c.x << "," << c.y;
    return 0;
}
```

It may be a little confusing to see so many times the CVector identifier. But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

```
CVector (int, int);             // function name CVector (constructor)
CVector operator+ (CVector);    // function returns a CVector
```

The function operator + of class CVector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:

```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

```
CVector () { };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

```
CVector c;
```

included in main() would not have been valid.

Anyway, I have to warn you that an empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case, this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

```
CVector () { x=0; y=0; };
```

which in order to simplify and show only the point of the code I have not included in the example.

As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (=) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
CVector d (2,3);
CVector e;
e = d;          // copy assignment operator
```

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use operator + to subtract two classes or operator== to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function operator+ can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

Expression	Operator	Member function	Global function
@a	+ - * & ! ~ ++ —	A::operator@()	operator@(A)
a@	++ —	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@ (B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@ (B)	-
a(b, c...)	()	A::operator() (B, C...)	-
a->x	->	A::operator->()	-

Where a is an object of class A, b is an object of class B and c is an object of class C.

You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

EXAMPLES OF OPERATOR OVERLOADING

Example 1: overloading '+' Operator

```
#include <iostream.h>
 class myclass
  {
    int sub1, sub2;
  public:
    // default constructor
    myclass(){}
    // main constructor
    myclass(int x, int y){sub1=x;sub2=y;}
    // notice the declaration
    myclass operator +(myclass);
    void show(){cout<<sub1<<endl<<sub2;}
  };
 // returns data of type myclass
 myclass myclass::operator +(myclass ob)
 {
   myclass temp;
   // add the data of the object
   // that generated the call
   // with the data of the object
   // passed to it and store in temp
   temp.sub1=sub1 + ob.sub1;
   temp.sub2=sub2 + ob.sub2;
   return temp;
 }
```

```
void main()
{
  myclass ob1(10,90);
  myclass ob2(90,10);
  // this is valid
  ob1=ob1+ob2;
  ob1.show();
}
```

Example 2 : // Another example illustrates overloading the plus (+) operator.

```
#include <iostream>
using namespace std;
class complx
{
      double real,
            imag;
public:
      complx( double real = 0., double imag = 0.); // constructor
      complx operator+(const complx&) const;        // operator+()
};
// define constructor
complx::complx( double r, double i )
{
      real = r; imag = i;
}
// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const
{
      complx result;
      result.real = (this->real + c.real);
      result.imag = (this->imag + c.imag);
      return result;
}
```

```
int main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}
```

Example 3: Overloading Extraction Operator

Suppose you declared the following class:

```
class student
 {
 private:
   string name;
   string department;
 public:
   student(string n = "", string dep = 0)
   : name(n), department(dep) {}
   string get_name() const { return name; }
   string get_department () const { return department; }
   void set_name(const string& n) { name=n; }
   void set_department (const string& d) {department=d;}
 };
```

And you want to be able to use it in a cout statement as follows:

```
student st("Bill Jones", "Zoology"); // create instance
cout<<st; // display student's details
```

First, you need to overload the operator << of class ostream (note that cout is an instance of ostream). The canonical form of such an overloaded << is this:

```
ostream& operator << (ostream& os, const student& s);
```

The overloaded << returns a reference to an ostream object and takes two parameters by reference: an ostream object and a user-defined type. The user-defined type is passed as a const parameter because the output operation doesn't modify it. The body of the overloaded << inserts members of the user-defined object into the ostream object:

```
os<<s.get_name()<<'\t'<<st.get_department()<<endl;
```

Make sure that the members inserted are separated by a tab, newline or space so that they appear as if they were concatenated when displayed on the screen. Remember also to place the endl manipulator at the end of the insertion chain to force a buffer flush. Finally, the overloaded operator should return the ostream object after the members have been inserted to it. This will enable you to chain several objects in a single cout statement:

```
student s1, s2;
cout<<s1<<s2;  // chaining multiple objects
```

The insertion operations and the return statement can be accomplished in a single statement:

```
ostream& operator << (ostream& os, const student& s)
{
  return os<<s.get_name()<<'\t'<<s.get_department()<<endl;
}
```

Now you can use the overloaded <<in your code:

```
 int main()
{
  student st("Bill Jones", "Zoology");
  cout<<st;
}
```

As expected, this program displays:

```
Bill Jones        Zoology
```

Example 4: Overloading the ! Operator:

```
#include <iostream>
using namespace std;
struct X { };
void operator!(X) {
  cout << "void operator!(X)" << endl;
}
struct Y {
  void operator!() {
    cout << "void Y::operator!()" << endl;
```

```
  }
};
struct Z { };
int main() {
  X ox; Y oy; Z oz;
  !ox;
  !oy;
//  !oz;
}
```

The following is the output of the above example:

```
void operator!(X)
void Y::operator!()
```

The operator function call !ox is interpreted as operator!(x). The call !oy is interpreted as y.operator!(). (The compiler would not allow !oz because the **!** operator has not been defined for class Z.)

Example 5: Overloading Increment and Decrement

You overload the prefix increment operator ++ with either a non-member function operator that has one argument of class type or a reference to class type, or with a member function operator that has no arguments.

In the following example, the increment operator is overloaded in both ways:

```
class X {
public:
  // member prefix ++x
  void operator++() { }
};
class Y { };
// non-member prefix ++y
void operator++(Y&) { }
int main() {
  X x;
  Y y;
  // calls x.operator++()
  ++x;
  // explicit call, like ++x
```

```
  x.operator++();
  // calls operator++(y)
  ++y;
  // explicit call, like ++y
  operator++(y);
}
```

The postfix increment operator ++ can be overloaded for a class type by declaring a non-member function operator operator++() with two arguments, the first having class type and the second having type **int**. Alternatively, you can declare a member function operator operator++() with one argument having type **int**. The compiler uses the **int** argument to distinguish between the prefix and postfix increment operators. For implicit calls, the default value is zero.

For example:

```
class X {
public:
  // member postfix x++
  void operator++(int) { };
};
class Y { };
// non-member postfix y++
void operator++(Y&, int) { };
int main() {
  X x;
  Y y;
  // calls x.operator++(0)
  // default argument of zero is supplied by compiler
  x++;
  // explicit call to member postfix x++
  x.operator++(0);
  // calls operator++(y, 0)
  y++;
  // explicit call to non-member postfix y++
  operator++(y, 0);
}
```

The prefix and postfix decrement operators follow the same rules as their increment counterparts.

Example 6: Overloading Assignment Operator

You overload the assignment operator, operator=, with a nonstatic member function that has only one parameter. You cannot declare an overloaded assignment operator that is a non-member function. The following example shows how you can overload the assignment operator for a particular class:

```
struct X {
  int data;
  X& operator=(X& a) { return a; }
  X& operator=(int a) {
    data = a;
    return *this;
  }
};
int main() {
  X x1, x2;
  x1 = x2;       // call x1.operator=(x2)
  x1 = 5;        // call x1.operator=(5)
}
```

The assignment x1 = x2 calls the copy assignment operator X& X::operator=(X&). The assignment x1 = 5 calls the copy assignment operator X& X::operator=(int). The compiler implicitly declares a copy assignment operator for a class if you do not define one yourself. Consequently, the copy assignment operator (operator=) of a derived class hides the copy assignment operator of its base class.

However, you can declare any copy assignment operator as virtual. The following example demonstrates this:

```
#include <iostream>
using namespace std;
struct A {
  A& operator=(char) {
    cout << "A& A::operator=(char)" << endl;
    return *this;
  }
  virtual A& operator=(const A&) {
    cout << "A& A::operator=(const A&)" << endl;
    return *this;
```

```
  }
};
struct B : A {
    B& operator=(char) {
      cout << "B& B::operator=(char)" << endl;
      return *this;
    }
    virtual B& operator=(const A&) {
      cout << "B& B::operator=(const A&)" << endl;
      return *this;
    }
};
struct C : B { };
int main() {
  B b1;
  B b2;
  A* ap1 = &b1;
  A* ap2 = &b1;
  *ap1 = 'z';
  *ap2 = b2;
  C c1;
//  c1 = 'z';
}
```

The following is the output of the above example:

```
A& A::operator=(char)
B& B::operator=(const A&)
```

The assignment *ap1 = 'z' calls A& A::operator=(char). Because this operator has not been declared virtual, the compiler chooses the function based on the type of the pointer ap1. The assignment *ap2 = b2 calls B& B::operator=(const &A). Because this operator has been declared virtual, the compiler chooses the function based on the type of the object that the pointer ap1 points to. The compiler would not allow the assignment c1 = 'z' because the implicitly declared copy assignment operator declared in class C hides B& B::operator=(char).

REVIEW EXERCISE

1. How one can overload '++'operator to increment an object?
2. Overload a function called DRAW that can find area of a rectangle, triangle, circle, square and sphere.
3. Which of the operators cannot be overloaded?
4. What are the benefits of using overloading?
5. How the concept of polymorphism is associated with overloading?
6. Overload "=" operator to see if two objects are exactly same.
7. Overload ComputeSalary function that computes the salary of regular, visiting and part-time employees.

CHAPTER

4 INHERITANCE, POLYMORPHISM & VIRTUAL FUNCTIONS

4.1 WHAT IS INHERITANCE?

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

For example, a programmer can create a *base* class named fruit and define *derived* classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
        { /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier limits the most accessible level for the members inherited from the base class. The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

This concept of *Inheritance* leads to the concept of *polymorphism.* Inheritance is what separates abstract data type (ADT) programming from OO programming.

For example, we are going to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.

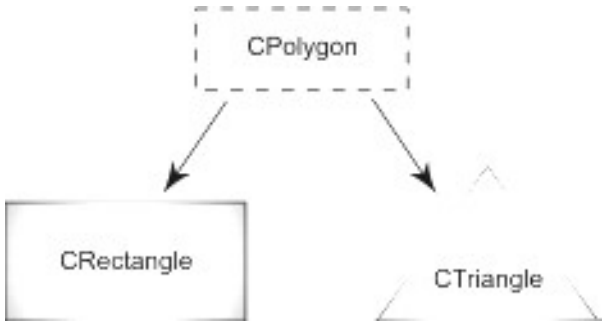


Figure 4.1

The class CPolygon would contain members that are common for both types of polygon. In our case, width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.

```
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
```

```
    };
class CRectangle: public CPolygon {
    public:
        int area ()
            { return (width * height); }
    };
class CTriangle: public CPolygon {
    public:
        int area ()
            { return (width * height / 2); }
    };
int main () {
    CRectangle rect;
    CTriangle trgl;
    rect.set_values (4,5);
    trgl.set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
```

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are: width, height and set_values().

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used protected access instead of private.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width          // protected access
CRectangle::width        // protected access

CPolygon::set_values()   // public access
CRectangle::set_values() // public access
```

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes.

```
class CRectangle: public CPolygon { ... }
```

This public keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case CPolygon) will have. Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

```
class daughter: protected mother;
```

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

4.2 WHAT IS INHERITED FROM THE BASE CLASS?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

For example:

```
// constructors and derived classes
#include <iostream>
using namespace std;

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};
class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};
class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};
int main () {
  daughter cynthia (0);
  son daniel(0);
  return 0;
}
```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son:

```
daughter (int a)          // nothing specified: call default
son (int a) : mother (a)  // constructor specified: call this
```

4.3 FEATURES OR ADVANTAGES OF INHERITANCE

Reusability: Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

Saves Time and Effort: The above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situations as needed.

Increases Program Structure which Results in Greater Reliability

For example, if the *base* class is *sample* and the derived class is sample it is specified as:

```
class sample: public sample
```

The above makes sample have access to both *public* and *protected* variables of base class *sample*.

Reminder about public, private and protected access specifiers:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
- Public members and variables are accessible from outside the class.
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

C++ inheritance is very similar to a parent-child relationship. When a class is inherited all the functions and data member are inherited, although not all of them will be accessible by the member functions of the derived class. But there are some exceptions to it too. Some of the exceptions to be noted in C++ inheritance are as follows.

- The constructor and destructor of a base class are not inherited.
- The assignment operator is not inherited.
- The friend functions and friend classes of the base class are also not inherited.

There are some points to be remembered about C++ inheritance. The **protected** and **public** variables or members of the base class are all accessible in the derived class. But a private member variable not accessible by a derived class. It is a well known fact that the private and protected members are not accessible outside the class. But a derived class is given access to protected members of the base class.

Let us see a piece of sample code for C++ inheritance. The sample code considers a class named vehicle with two properties to it, namely color and the number of wheels. A vehicle is a generic term and it can later be extended to any moving vehicles like car, bike, bus etc.

```
class vehicle //Sample base class for c++ inheritance
  {
  protected:
    char colorname[20];
    int number_of_wheels;
  public:
    vehicle();
    ~vehicle();
    void start();
    void stop();
  };
class Car: public vehicle //Sample derived class for C++ inheritance
  {
  protected:
    char type_of_fuel;
    public:
    Car();
  };
```

The derived class Car will have access to the protected members of the base class. It can also use the functions start, stop and run provided the functionalities remain the same. In case the derived class needs some different functionalities for the same functions start, stop and run, then the base class should implement the concept of virtual functions.

Inheritance Example:

```
class sample
{
public:
```

```
sample(void) { x=0; }
void f(int n1)
{
x= n1*5;
}
void output(void) { cout<<x; }
private:
int x;
};
class sample: public sample
{
public:
sample(void) { s1=0; }
void f1(int n1)
{
s1=n1*10;
}
void output(void)
{
sample::output();
cout << s1;
}
private:
int s1;
};
int main(void)
{
sample s;
s.f(10);
s.output();
s.f1(20);
s.output();
}
```

The output of the above program is

```
5                                                    0
200
```

In the above example, the derived class is sample and the base class is *sample*. The *derived* class defined above has access to all *public* and *private* variables. *Derived* classes cannot have access to base class *constructors* and *destructors*. The derived class would be able to add new member functions, or variables, or new constructors or new destructors. In the above example, the derived class sample has new member function f1( ) added in it. The line:

```
sample s;
```

creates a derived class object named as s. When this is created, space is allocated for the data members inherited from the base class *sample* and space is additionally allocated for the data members defined in the derived class *sample*.

The *base* class constructor *sample* is used to initialize the base class data members and the *derived* class *constructor* sample is used to initialize the data members defined in *derived* class.

The access specifier specified in the line:

```
class sample: public sample
```

Public indicates that the *public* data members which are inherited from the *base* class by the derived class sample remains *public* in the *derived* class.

4.4 TYPES OF INHERITANCE

C++ distinguishes two types of inheritance: *public* and *private*. As a default, classes are privately derived from each other. Consequently, we must explicitly tell the compiler to use public inheritance.

The type of inheritance influences the access rights to elements of the various superclasses. Using public inheritance, everything which is declared private in a superclass remains private in the subclass. Similarly, everything which is public remains public. When using private inheritance the things are quite different as is shown in table below.

Table 4.1: Access rights and inheritance

Type of Inheritance		
	Private	**Public**
Private	Private	Private
Protected	Private	Protected
Public	Private	Public

The leftmost column lists possible access rights for elements of classes. It also includes a third type protected. This type is used for elements which should be directly usable

in subclasses but which should not be accessible from the outside. Thus, one could say elements of this type are between private and public elements in that they can be used within the class hierarchy rooted by the corresponding class.

Inheritance in C++ can also be classified as Single, Multiple, multilevel, Hierarchical, multipath and Hybrid.

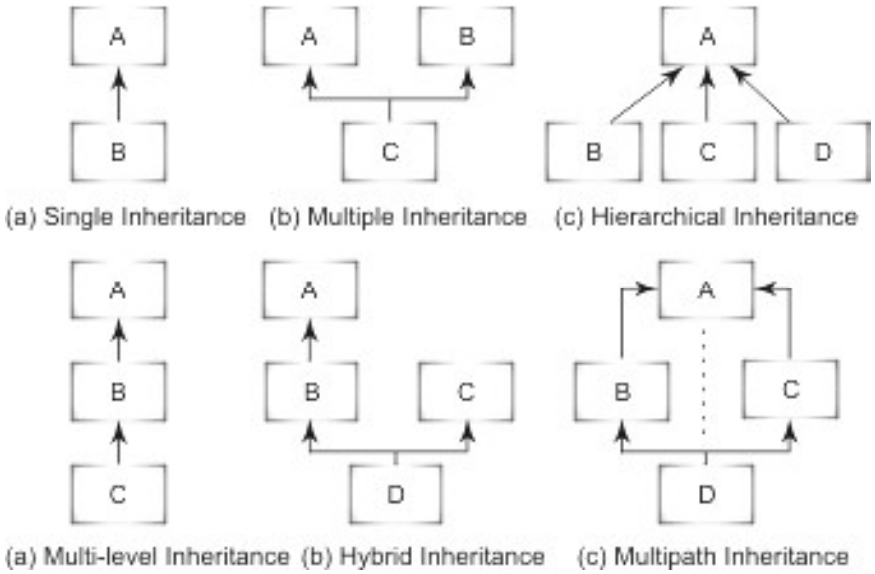


Figure 4.2: Different forms of Inheritance

Inheritance comes in two forms, depending on number of parents a subclass has

1. Single Inheritance (SI)

- Only one parent per derived class
- Form an inheritance tree
- SI requires a small amount of run-time overhead when used with dynamic binding
- e.g., Smalltalk, Simula, Object Pascal

2. Multiple Inheritance (MI)

- More than one parent per derived class
- Forms an inheritance Directed Acyclic Graph (DAG)
- Compared with SI, MI adds additional run-time overhead (also involving dynamic binding)
- e.g., C++, Eiffel, Flavors (a LISP dialect)

4.4.1 Single Inheritance

In "single inheritance," a common form of inheritance, classes have only one base class. Consider the relationship illustrated in the following figure.

Simple Single-Inheritance Graph

Note the progression from general to specific in the figure. Another common attribute found in the design of most class hierarchies is that the derived class has a "kind of" relationship with the base class. In the figure, a Book is a kind of a PrintedDocument, and a PaperbackBook is a kind of a book.

```
class abc // example of single inheritance
{
protected:
int x;
int y;
};
class def:abc
{
private:
int z;
public:
void display()
{
x=10;
y=20;
z=x+y;
cout<<z<<"
";
}
};
```

```
void main()
{
class def o;
o.display();
getch();
}
```

4.4.2 Multiple Inheritance

Multiple inheritance refers to a feature of some object-oriented programming languages in which a class can inherit behaviors and features from more than one superclass. This contrasts with single inheritance, where a class may inherit from at most one superclass.

Multiple inheritance allows a class to take on functionality from multiple other classes, such as allowing a class named StudentMusician to inherit from a class named Person, a class named Musician, and a class named Worker. This can be abbreviated StudentMusician : Person, Musician, Worker.

Ambiguities arise in multiple inheritance, as in the example above, if for instance the class Musician inherited from Person and Worker and the class Worker inherited from Person. This is referred to as the Diamond problem. There would then be the following rules:

Worker : Person

Musician : Person, Worker

StudentMusician : Person, Musician, Worker

If a compiler is looking at the class StudentMusician it needs to know whether it should join identical features together, or whether they should be separate features. For instance, it would make sense to join the "Age" features of Person together for StudentMusician. A person's age doesn't change if you consider them a Person, a Worker, or a Musician. It would, however, make sense to separate the feature "Name" in Person and Musician if they use a different stage name than their given name. The options of joining and separating are both valid in their own context and only the programmer knows which option is correct for the class they are designing.

An Example

```
class computer_screen {
public:
computer_screen(char *, long, int, int);
void show_screen(void);
   private:
```

```
char type[32];
long colors;
int x_resolution;
int y_resolution;
};
class mother_board {
 public:
mother_board(int, int, int);
void show_mother_board(void);
private:
int processor;
int speed;
int RAM;
};
class computer: public computer_screen, public mother_board
{
public:\
computer(char *, int, float,
char*, long, int,
int, int, int, int);
void show_computer(void);
private:
char name[64];
int hard_disk; //size of
float floppy;
};
```

Hiding Inherited Classes

	Public Derivation (e.g., public box)	Protected Derivation (e.g., protected box)	Private Derivation (e.g., private box)
Public Member Function	Remains public	Remains protected	Remains private
Protected Member Function	Becomes protected	Remains protected	Remains private
Private Member Function	Becomes private (to derived)	Becomes private (to derived)	Remains private

4.4.3 Multilevel Inheritance

Here the inheritance is extended beyond one level. For example, class A is inherited by say class B and further class C inherits class B. This is an example of multilevel inheritance.

```
/********* IMPLEMENTATION OF MULTILEVEL INHERITANCE *********/
        #include< iostream.h>
        #include< conio.h>
        class student // Base Class
        {
        protected:
        int rollno;
        char *name;
        public:
        void getdata(int b,char *n)
        {
        rollno = b;
        name = n;
        }
        void putdata
        (void)
        {
        cout< < " The Name Of Student \t:" < < name< < endl;
        cout< < " The Roll No. Is \t:" < < rollno< < endl;
        }
        };
        class test:public student // Derieved Class 1
        {
        protected:
        float m1,m2;
        public:
        void gettest(float b,float c)
        {
        m1 = b;
        m2 = c;
        }
```

```
void puttest(void)
{
cout< < " Marks In CP Is \t:" < < m1< < endl;
cout< < " Marks In Drawing Is \t:" < < m2< < endl;
}
};
class result:public test // Derieved Class 2
{
protected:
float total;
public:
void displayresult(void)
{
total = m1 + m2;
putdata();
puttest();
cout< < " Total Of The Two \t: "< < total< < endl;
}
};
void main()
{
clrscr();
int x;
float y,z;
char n[20];
cout< < "Enter Your Name:";
cin>>n;
cout< < "Enter The Roll Number:";
cin>>x;
result r1;
r1.getdata(x,n);
cout< < "ENTER COMPUTER PROGRAMMING MARKS:";
cin>>y;
cout< < "ENTER DRAWING MARKS:";
cin>>z;
```

```
r1.gettest(y,z);
cout< < endl< < endl< < "********** RESULT **********"< <
endl;
r1.displayresult();
cout< < "**********************************"< < endl;
getch();
}
/********** OUTPUT **********
Enter Your Name:Lionel
Enter The Roll Number:44
ENTER COMPUTER PROGRAMMING MARKS:95
ENTER DRAWING MARKS:90
********** RESULT **********
The Name Of Student : Lionel
The Roll No. Is : 44
Marks In CP Is : 95
Marks In Drawing Is : 90
Total Of The Two : 185
**********************************
*/
```

Another Example of Inheritance

```
#define male 'm'
#define female 'f'
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <string>
using namespace std;
class person
{
public:
    person(int,char,string);
    person();
   int id();
```

```
    char sex();
    string name();
    int changename(string);
    static int reset_count();
protected:
    int person_id;
    char person_sex;
    string person_name;
    static int count;
};
class student: public person
{
public:
    student (string,char);
    student (string,char,char);
    int number();
    char specialization();
protected:
    int student_number;
    char student_specialization;
};
class employee: public person
{
public:
    employee (int,char);
    employee (string,char,char);
    int number ();
    char working ();
protected:
    int employee_number;
    char employee_working;
};
int person::count;
int person::reset_count() {
    person::count=0;
```

```
    return (person::count);
}
person::person(int a, char b, string c)
{
    person_id=person::count;
    person::count++;
    person_sex=b;
    person_name=c;
}
person::person()
{
    person_id=person::count;
    person::count++;
    char this_persons_sex;
    int random=rand()%2;
    if (random==0) this_persons_sex='m';
    else this_persons_sex='f';
    person_sex=this_persons_sex;
    person_name="";
}
int person::id()
{
    return (person_id);
}
char person::sex()
{
    return (person_sex);
}
string person::name()
{
    return (person_name);
}
int person::changename(string a)
{
    person_name=a;
```

```
    return 0;
}
student::student(string a, char b )
{
    student_number=person_id;
    person_name=a;
    student_specialization=b;
}
student::student(string a, char b, char c )
{
    person_name=a;
    student_number=person_id;
    student_specialization=b;
    person_sex=c;
}
int student::number()
{
    return (student_number);
}
char student::specialization()
{
    return (student_specialization);
}
employee::employee(int a, char b )
{
    employee_number=a;
    employee_working=b;
}

employee::employee(string a, char b, char c )
{
    person_name=a;
    employee_number=person_id;
    person_sex=b;
    employee_working=c;
```

```
}
int employee::number()
{
    return (employee_number);
}
char employee::working()
{
    return (employee_working);
}
int main()
{
   person::reset_count();
    srand(time(NULL));
   employee ** emp;
   student ** stu;
   char tmpsex;
   string tmpname;
   string junk;
   char tmpspec;
   emp=new employee*[10];
   stu=new student*[10];
   cout<<"students:"<<endl;
    for (int i=0;i<10;i++)
    {
       cout<<"name:";
        getline(cin,tmpname);\
       cout<<"specialization: ";
       cin >> tmpspec;
       cout<<"sex:";
       cin >> tmpsex;
        getline(cin,junk); // clear input buffer from junk cin leaves there
        stu[i]=new student(tmpname,tmpspec,tmpsex);
    }
   cout<<"employees:"<<endl;
 for (int i=0;i<10;i++)
```

```
{
   cout<<"name:";
    getline(cin,tmpname);
    cout<<"sex:";
    cin>> tmpsex;
 cout << "Working?:";
    cin >> tmpspec;
    getline(cin,junk);
    emp[i]=new employee(tmpname,tmpsex,tmpspec);
 }
 cout << "Student Data:\nid\tname\tsex\tspecialization\n";
 cout<<"-------------------------------------------------------"<<endl;
 for (int i =0; i<10;++i)
 {
cout<<stu[i]->id()<<"\t"<<stu[i]->name()<<"\t"<<stu[i]->sex()<<"\t"<<stu[i]
>specialization()<<endl;
 }
cout<<"-------------------------------------------------------"<<endl;
 cout << "\nEmployee Data:\nid\tname\tsex\tworking\n";
cout<<"-------------------------------------------------------"<<endl;
 for (int i =0; i<10;++i)
 {
     cout<<emp[i]->id()<<"\t"<<emp[i]->name()<<"\t"<<emp[i]
>sex()<<"\t"<<emp[i]->working()<<endl;
 }
cout<<"-------------------------------------------------------"<<endl;
 cout<<"\n\nPress <ENTER> to Exit.";
 cin.get();
 return 0;
}
```

4.5 C++ POLYMORPHISM

4.5.1 Introduction

Polymorphism is the ability to use an operator or function in different ways. Polymorphism gives different meanings or functions to the operators or functions.

Poly, referring to many, signifies the many uses of these operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

Polymorphism is a powerful feature of the object oriented programming language C++. A single operator + behaves differently in different contexts such as integer, float or strings referring the concept of *polymorphism.* The above concept leads to operator *overloading*. The concept of overloading is also a branch of *polymorphism.* When the exiting operator or function operates on new data type it is *overloaded.* This feature of polymorphism leads to the concept of *virtual methods.*

Polymorphism refers to the ability to call different functions by using only one type of function call. Suppose a programmer wants to code vehicles of different shapes such as circles, squares, rectangles, etc. One way to define each of these classes is to have a member function for each that makes vehicles of each shape. Another convenient approach the programmer can take is to define a base class named Shape and then create an instance of that class. The programmer can have array that hold pointers to all different objects of the vehicle followed by a simple loop structure to make the vehicle, as per the shape desired, by inserting pointers into the defined array. This approach leads to different functions executed by the same function call. Polymorphism is used to give different meanings to the same concept. This is the basis for *Virtual function* implementation.

In polymorphism, a single function or an operator functioning in many ways depends upon the usage to function properly. In order for this to occur, the following conditions must apply:

- All different classes must be derived from a single base class. In the above example, the shapes of vehicles (circle, triangle, rectangle) are from the single base class called Shape.
- The member function must be declared virtual in the base class. In the above example, the member function for making the vehicle should be made as virtual to the base class.

4.5.2 Features and Advantages of the Concept of Polymorphism

Applications are easily extendable: Once an application is written using the concept of polymorphism, it can easily be extended, providing new objects that conform to the original interface. It is unnecessary to recompile original programs by adding new types. Only re-linking is necessary to exhibit the new changes along with the old application. This is the greatest achievement of C++ object-oriented programming. In programming language, there has always been a need for adding and customizing. By utilizing the concept of polymorphism, time and work effort is reduced in addition to making future maintenance easier.

- Helps in reusability of code.
- Provides easier maintenance of applications.
- Helps in achieving robustness in applications.

4.5.3 Types of Polymorphism

C++ provides three different types of polymorphism.

- Virtual functions
- Function name overloading
- Operator overloading

We have already covered the basics of Function and Operator overloading in previous chapter.

In addition to the above three types of polymorphism, there exist other kinds of polymorphism:

- run-time
- compile-time
- ad-hoc polymorphism
- parametric polymorphism

Other types of polymorphism defined:

Run-time: The *run-time* polymorphism is implemented with inheritance and virtual functions.

Compile-time: The *compile-time* polymorphism is implemented with templates.

Ad-hoc polymorphism: If the range of actual types that can be used is finite and the combinations must be individually specified prior to use, this is called *ad-hoc* polymorphism.

Parametric polymorphism: If all code is written without mention of any specific type and thus can be used transparently with any number of new types it is called *parametric* polymorphism.

4.6 VIRTUAL FUNCTION

4.6.1 What is a Virtual Function?

A virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. It is one that is declared as virtual in the base class using the virtual keyword. The virtual nature is inherited in the subsequent derived classes and the virtual keyword need not be re-stated there. The whole function body can be replaced with a new set of implementation in the derived class.

4.6.2 What is Binding?

Binding refers to the act of associating an object or a class with its member. If we can call a method fn() on an object o of a class c, we say that the object o is binded with the method fn(). This happens at compile time and is known as static or compile - time binding. The calls to the virtual member functions are resolved during run-time. This mechanism is known as dynamic binding. The most prominent reason why a virtual function will be used is to have a different functionality in the derived class. The difference between a non-virtual member function and a virtual member function is, the non-virtual member functions are resolved at compile time.

4.6.3 How does a Virtual Function Work?

Whenever a program has a virtual function declared, a v - table is constructed for the class. The v-table consists of addresses to the virtual functions for classes that contain one or more virtual functions. The object of the class containing the virtual function contains a virtual pointer that points to the base address of the virtual table in memory. Whenever there is a virtual function call, the v-table is used to resolve to the function address. An object of the class that contains one or more virtual functions contains a virtual pointer called the vptr at the very beginning of the object in the memory. Hence the size of the object in this case increases by the size of the pointer. This vptr contains the base address of the virtual table in memory. Note that virtual tables are class specific, i.e., there is only one virtual table for a class irrespective of the number of virtual functions it contains. This virtual table in turn contains the base addresses of one or more virtual functions of the class. At the time when a virtual function is called on an object, the vptr of that object provides the base address of the virtual table for that class in memory. This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call.

The following code shows how we can write a virtual function in C++ and then use the same to achieve dynamic or runtime polymorphism.

```
#include <iostream.h>
class base
{
public:
virtual void display()
{
  cout<<"\nBase";
}
};
class derived : public base
```

```
{
  public:
  void display()
  {
     cout<<"\nDerived";
  }
};
void main()
{
  base *ptr = new derived();
  ptr->display();
}
```

In the above example, the pointer is of type base but it points to the derived class object. The method display() is virtual in nature. Hence in order to resolve the virtual method call, the context of the pointer is considered, i.e., the display method of the derived class is called and not that of the base. If the method was non virtual in nature, the display() method of the base class would have been called.

4.6.4 Virtual Constructors and Destructors

A constructor cannot be virtual because at the time when the constructor is invoked the virtual table would not be available in the memory. Hence we cannot have a virtual constructor.

A virtual destructor is one that is declared as virtual in the base class and is used to ensure that destructors are called in the proper order. It is to be remembered that destructors are called in the reverse order of inheritance. If a base class pointer points to a derived class object and we some time later use the delete operator to delete the object, then the derived class destructor is not called. Refer to the code that follows:

```
#include <iostream.h>
class base
{
   public:
   ~base()
  {
  }
};
class derived : public base
```

```
{
  public:
  ~derived()
  {
}
};
void main()
{
  base *ptr = new derived();
  // some code
  delete ptr;
}
```

In this case the type of the pointer would be considered. Hence as the pointer is of type base, the base class destructor would be called but the derived class destructor would not be called at all. The result is memory leak. In order to avoid this, we have to make the destructor virtual in the base class. This is shown in the example below:

```
#include <iostream.h>
class base
{
public:
virtual ~base()
{
}
};
class derived : public base
{
  public:
  ~derived()
  {
  }
};
void main()
{
  base *ptr = new derived();
  // some code
  delete ptr;
}
```

Example: C++ Virtual Function

```
class Window // Base class for C++ virtual function example
    {
     public:
        virtual void Create() // virtual function for C++ virtual function example
        {
            cout <<"Base class Window"<<endl;
        }
};
class CommandButton : public Window
    {
     public:
        void Create()
        {
           cout<<"Derived class Command Button - Overridden C++ virtual
function"<<endl;
        }
    };
void main()
    {
       Window  *x, *y;
       x = new Window();
       x->Create();
       y = new CommandButton();
       y->Create();
    }
```

The output of the above program will be,

Base class Window

Derived class Command Button

If the function ***had not been declared virtual***, then the base class function would have been called all the times. Because, the function address would have been statically bound during compile time. But now, as the function is declared virtual it is a candidate for run-time linking and the derived class function is being invoked.

4.6.5 C++ Virtual function - Call Mechanism

Whenever a program has a C++ virtual function declared, a v-table is constructed for the class. The v-table consists of addresses to the virtual functions for classes and

pointers to the functions from each of the objects of the derived class. Whenever there is a function call made to the C++ virtual function, the v-table is used to resolve to the function address. This is how the Dynamic binding happens during a virtual function call.

Example of Virtual Function

```
#include <string.h>
#include <assert.h>
    #include <iostream.h>
    typedef double Coord;
    /*
    The type of X/Y points on the screen.
    */
    enum Color {Co_red, Co_green, Co_blue};
/*
    Colors.
    */
    // abstract base class for all shape types
    class Shape {
    protected:
        Coord xorig; // X origin
        Coord yorig; // Y origin
        Color co; // color
    /*
```

These are protected so that they can be accessed by derived classes. Private wouldn't allow this.

These data members are common to all shape types.

```
    */
public:
        Shape(Coord x, Coord y, Color c) :
            xorig(x), yorig(y), co(c) { } // constructor
    /*
    Constructor to initialize data members common to all shape types.
    */
        virtual ~Shape() { } // virtual destructor
    /*
```

```
Destructor for Shape.  It's a virtual function.
Destructors in derived classes are virtual also because this one is declared so.
*/
virtual void draw() = 0; // pure virtual draw() function
/*
Similarly for the draw() function.  It's a pure virtual and  is not called directly.
*/
};
// line with X,Y destination
class Line : public Shape {
/*
Line is derived from Shape, and picks up its data members.
*/
        Coord xdest; // X destination
        Coord ydest; // Y destination
/*
Additional data members needed only for Lines.
*/
public:
        Line(Coord x, Coord y, Color c, Coord xd, Coord yd) :
            xdest(xd), ydest(yd),
            Shape(x, y, c) {} // constructor with base initialization
/*
Construct a Line, calling the Shape constructor as well to initialize data members of
the base class.
*/
~Line() {cout << "~Line\n";} // virtual destructor
/*
Destructor.
*/
```

```
        void draw() // virtual draw function
        {
                cout << "Line" << "(";
                cout << xorig << "," << yorig << "," << int(co);
                cout << "," << xdest << "," << ydest;
                cout << ")\n";
        }
    /*
    Draw a line.
    */
    };
    // circle with radius
    class Circle : public Shape {
        Coord rad; // radius of circle
    /*
    Radius of circle.
    */
    public:
        Circle(Coord x, Coord y, Color c, Coord r) : rad(r),
            Shape(x, y, c) {} // constructor with base initialization
        ~Circle() {cout << "~Circle\n";} // virtual destructor
        void draw() // virtual draw function
        {
                cout << "Circle" << "(";
                cout << xorig << "," << yorig << "," << int(co);
            cout << "," << rad;
                cout << ")\n";
        }
    };
    // text with characters given
    class Text : public Shape {
     char* str; // copy of string
    public:
Text(Coord x, Coord y, Color c, const char* s) :
            Shape(x, y, c) // constructor with base initialization
```

```
        {
str = new char[strlen(s) + 1];
assert(str);
strcpy(str, s);
    /*
```

Copy out text string. Note that this would be done differently if we were taking advantage of some newer C++ features like exceptions and strings.

```
    */
     }
        ~Text() {delete [] str; cout << "~Text\n";} // virtual dtor
    /*
    Destructor; delete text string.
    */
        void draw() // virtual draw function
        {
            cout << "Text" << "(";
            cout << xorig << "," << yorig << "," << int(co);
            cout << "," << str;
            cout << ")\n";
        }
    };
    int main()
    {
        const int N = 5;
        int i;
        Shape* sptrs[N];
    /*
```

Pointer to vector of Shape* pointers. Pointers to classes derived from Shape can be assigned to Shape* pointers.

```
    */
       // initialize set of Shape object pointers
        sptrs[0] = new Line(0.1, 0.1, Co_blue, 0.4, 0.5);
        sptrs[1] = new Line(0.3, 0.2, Co_red, 0.9, 0.75);
        sptrs[2] = new Circle(0.5, 0.5, Co_green, 0.3);
```

```
        sptrs[3] = new Text(0.7, 0.4, Co_blue, "Howdy!");
        sptrs[4] = new Circle(0.3, 0.3, Co_red, 0.1);
/*
Create some shape objects.
*/
        // draw set of shape objects
        for (i = 0; i < N; i++)
                sptrs[i]->draw();
/*
Draw them using virtual functions to pick up the
right draw() function based on the actual object
type being pointed at.
*/
        // cleanup
        for (i = 0; i < N; i++)
                delete sptrs[i];
/*
Clean up the objects using virtual destructors.
*/
        return 0;
}
```

When we run this program, the output is:

```
Line(0.1, 0.1, 2, 0.4, 0.5)
Line(0.3, 0.2, 0, 0.9, 0.75)
Circle(0.5, 0.5, 1, 0.3)
Text(0.7, 0.4, 2, Howdy!)
Circle(0.3, 0.3, 0, 0.1)
~Line
~Line
~Circle
~Text
~Circle
```

with enum color values represented by small integers.

A few additional comments. Virtual functions typically are implemented by placing a pointer to a jump table in each object instance. This table pointer represents the "real" type of the object, even though the object is being manipulated through a base class pointer.

Because virtual functions usually need to have their function address taken, to store in a table, declaring them inline as the above example does is often a waste of time. They will be laid down as static copies per object file. There are some advanced techniques for optimizing virtual functions, but you can't count on these being available.

Note that we declared the Shape destructor virtual (there are no virtual constructors). If we had not done this, then when we iterated over the vector of Shape* pointers, deleting each object in turn, the destructors for the actual object types derived from Shape would not have been called, and in the case above this would result in a memory leak in the Text class.

Shape is an example of an abstract class, whose purpose is to serve as a base for derived classes that actually do the work. It is not possible to create an actual object instance of Shape, because it contains at least one pure virtual function.

4.6.6 Pure Virtual Function

What is Pure Virtual Function?

Pure Virtual Function is a Virtual function with no body.

Declaration of Pure Virtual Function:

Since pure virtual function has no body, the programmer must add the notation =0 for declaration of the pure virtual function in the base class.

General Syntax of Pure Virtual Function takes the form:

```
class classname //This denotes the base class of C++ virtual function
{
public:
virtual void virtualfunctioname() = 0 //This denotes the pure virtual function in C++
};
```

The other concept of pure virtual function remains the same as described in the previous section of virtual function. To understand the declaration and usage of Pure Virtual Function, refer to this example:

```
class Exforsys
{
public:
```

```
virtual void example()=0; //Denotes pure virtual Function Definition
};
class Exf1:public Exforsys
{
public:
void example()
{
cout<<"Welcome";
}
};
class Exf2:public Exforsys
{
public:
void example()
{
cout<<"To Training";
}
};
void main()
{
Exforsys* arra[2];
 Exf1 e1;
Exf2 e2;
arra[0]=&e1;
arra[1]=&e2;
arra[0]->example();
arra[1]->example();
}
```

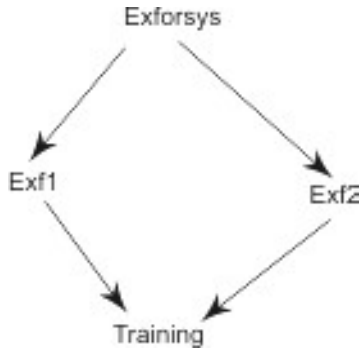
Since the above example has no body, the pure virtual function example() is declared with notation =0 in the base class Exforsys. The two derived class named Exf1 and Exf2 are derived from the base class Exforsys. The pure virtual function example() takes up new definition. In the main function, a list of pointers is defined to the base class.

Two objects named e1 and e2 are defined for derived classes Exf1 and Exf2. The address of the objects e1 and e2 are stored in the array pointers which are then used for accessing the pure virtual function example() belonging to both the derived class EXf1 and EXf2 and thus, the output is as in the above example.

The programmer must clearly understand the concept of pure virtual functions having no body in the base class and the notation =0 is independent of value assignment. The notation =0 simply indicates the Virtual function is a pure virtual function as it has no body.

Some programmers might want to remove this pure virtual function from the base class as it has no body but this would result in an error. Without the declaration of the pure virtual function in the base class, accessing statements of the pure virtual function such as, arra[0]->example() and arra[1]->example() would result in an error. The pointers should point to the base class Exforsys. Special care must be taken not to remove the statement of declaration of the pure virtual function in the base class.

Virtual Base Class



In the above example, there are two derived classes Exf1 and Exf2 from the base class Exforsys. As shown in the above diagram, the Training class is derived from both of the derived classes Exf1 and Exf2. In this scenario, if a user has a member function in the class Training where the user wants to access the data or member functions of the class Exforsys it would result in error if it is performed like this:

```
class Exforsys
{
protected:
int x;
};
 class Exf1:public Exforsys
{ };
 class Exf2:public Exforsys
{ };
 class Training:public Exf1,public Exf2
{
```

```
public:
int example()
{
return x;
}
};
```

The above program results in a compile time error as the member function example() of class Training tries to access member data x of class Exforsys. This results in an error because the derived classes Exf1 and Exf2 (derived from base class Exforsys) create copies of Exforsys called subobjects.

This means that each of the subobjects have Exforsys member data and member functions and each have one copy of member data x. When the member function of the class Training tries to access member data x, confusion arises as to which of the two copies it must access since it derived from both derived classes, resulting in a compile time error.

When this occurs, Virtual base class is used. Both of the derived classes Exf1 and Exf2 are created as virtual base classes, meaning they should share a common subobject in their base class.

For Example:

```
class Exforsys
{
protected:
int x;
;
class Exf1:virtual public Exforsys
{ };
 class Exf2:virtual public Exforsys
{ };
class Training:public Exf1,public Exf2
{
public:
int example()
{
return x;
}
};
```

In the above example, both Exf1 and Exf2 are created as Virtual base class

REVIEW EXERCISE

1. What is the significance of Inheritance in object oriented programming?
2. Describe the types of inheritance in C++.
3. Prepare a class EMP that has a function to accept the personal details of an employee. Design a derived class which has a function that accepts the salary details of an employee and a function that calculates the Net pay of the employee.
4. What is the concept of polymorphism and discuss its types and significance.
5. What are abstract classes?
6. How polymorphism can be implemented?
7. What are virtual functions?

CHAPTER

GETTING STARTED WITH VISUAL C++ 6

5

This chapter will teach you how to create a project in version six of Visual C++. This version of Microsoft's C++ IDE has probably helped millions of developers in their C++ Programming over the last 8 years. Microsoft Visual C++ has existed in many versions for over 13 years on the Win 32 platform. Version 6 is the last non .NET version and probably the most popular. It's been around since 1999 and has had six service packs.

In this chapter, you'll learn how to create a Command Line project, add some source code and then make it.

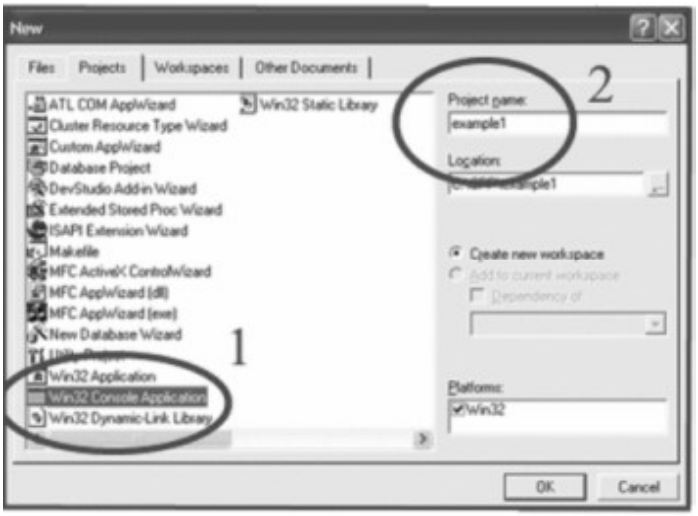


Figure 5.1

Before you start

Make sure your version has been updated with Service Pack 6. You can do this from the Microsoft website.

5.1 GETTING STARTED

We'll begin by creating a new Project. Visual C++ includes the AppWizard. This is a Wizard that does all the donkey-work of creating project files for you. You should get in the habit of using this as it saves a lot of time.

After starting the IDE, From the File menu click New and the New Dialog will popup. Select "Win 32 Console Application" (Red Circle 1 in the image), then enter a Project name (Red Circle 2 in the image) like Example1. Now select somewhere for the project files by clicking the location selector to the right of the Location: edit box and Press OK.

5.2 LEARN ABOUT PROJECTS AND WORKSPACES

Figure 5.2

Click the third radio button which says A "Hello, World" application, then click the Finish button. Press Ok on the next page and your workspace panel will now show Example 1 Files, with folders for Source Files, Header Files, and Resource files (There will be none) and a ReadMe.txt with a summary of the project files.

5.2.1 Projects in Visual C++ 6

Projects in Visual C++ 6 are organised in Workspaces; an application will normally have one workspace. Each workspace can hold one or more projects. In practice this means one project for each exe or dll.

If you look in the Win 32 folder where you created the project you'll see a number of files. There's the Source Files either c or cpp files (source code), at least one header file (stdafx.h) plus a .dsp file that holds the project details and a .dsw file that holds the workspace details.

Other files include the .plg which is created when you compile. It's a html file which holds the log of the compilation. Double click it and your default browser will open and display it.

The .ncb and .opt files holds information about the settings and log of Visual Studio- both are binary files so of no further interest to us.

5.3 COMPILING WITH VISUAL C++ 6

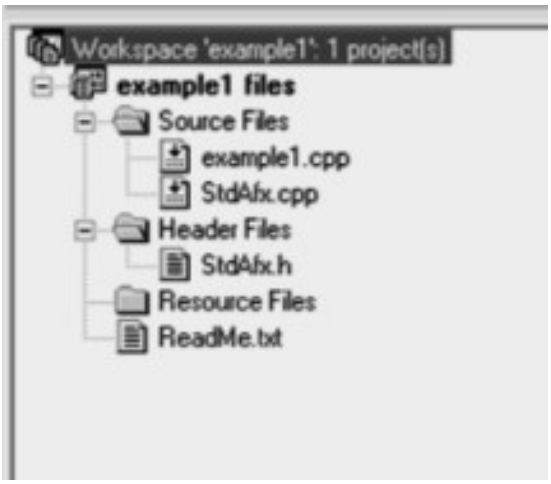


Figure 5.3

Compiling the Hello World Application

This is what the generated example1.cpp looks like.

```
// ex1.cpp : Defines the entry point for the Command Line application.
//
#include "stdafx.h"
```

```
int main(int argc, char* argv[])
{
printf("Hello World!\n") ;
return 0;
}
```

This is standard C++ file. In fact it's also a standard C as well though it defaults to cpp. You can mix cpp and c files but don't give them the same name as the compiler will expect to compile both example1.c and example1.cpp into example1.obj and it will object to having two files generate the same object file.

To remove files from the Project, just select each in the tree and press delete. To add a file right click on "Source Files" (for .cpp, or .c) or "Header Files" for .h and click "add files to Folder". This will open a window so you can browse to your file, select and add it.

Click on the Project name in the Workspace tree and press the **F7** key. That will Make the Hello World application. You can run it by pressing **F5** but you won't see much, as it's a console application and the window will open and close very quickly.

You need to get to the command line (Click the Start button, then click Run, type cmd and press enter) and navigate to the folder where the project files are located. After compiling, a debug folder is created there and this contains the debug executable which you can run.

5.4 DEBUG OR RELEASE PROJECTS?

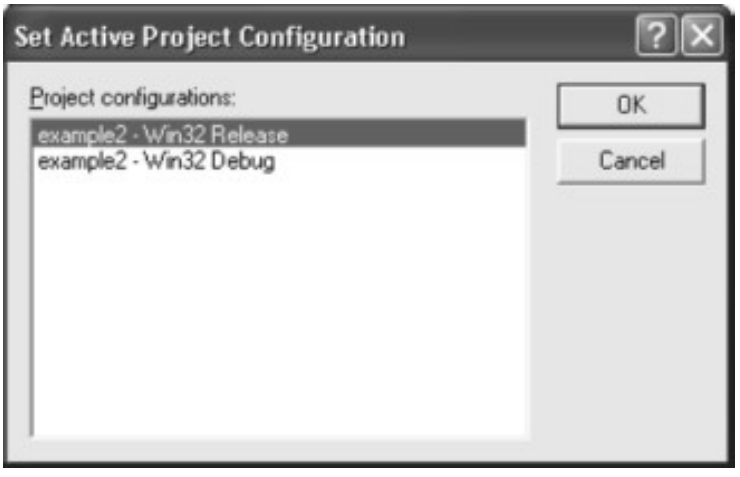


Figure 5.4

When developing a program, it's obviously important to be able to debug it. However for release you want to provide as small an executable as possible, unbloated by debug code. Here's how to do that.

Click "Build" on the main Menu, then "Set Active Configuration" on the drop-down menu. This opens a dialog that shows you all possible build configurations. Just switch to the "Release" configuration (that's the selected configuration in the picture), press Ok and then do another build. This creates a release folder containing the release executable. For a simple "Hello World" Application, the debug executable is 169 KB in size. The release exe is 40KB.

You aren't just limited to these two configurations either. Click "Build" on the main menu then "Configurations". This is where you add extra configurations. For example, a project for one customer may include additional functionality, perhaps implemented in an extra dll. This is where you create that configuration. You can then customize it in the Settings dialog. (Click "Project" on the Main menu then "Settings" on the drop-down).

5.5 CONFIGURING THE SETTINGS DIALOGS

Figure 5.5

Settings

This dialog is probably the most complex in Visual C++ 6. The defaults are good enough for many applications but there will come a time when you have to modify it. Here is few examples. For example **ALT-F7** is the keyboard shortcut to open the Settings dialog.

The settings tree control lets you create settings for different configurations. Choose a Configuration in the Combo. If you have common folders for resources then Choose "All Configurations" and click the "Resources Tab". Add one or more paths, separated by semi-colons to the Additional resource include directories.

The Project Options at the bottom of the first four settings tabs (C/C++, Link, Resources, Browse Info) show a summary of the options set by the controls on that tab. You can edit these directly or select the tab controls. For example Select the Link tab and scroll down the Project Options until you see **/out "Debug/example1.exe"** at the bottom line. Now select the p in example and delete it. You'll see the output file name edit box update to reflect this.

Most of the time you don't need to change the settings. Those that you are most likely to do will be specifying extra paths for include files (Select Preprocessor on the category combo on the C/C++ tab) and Resources as described above. For the rest, if you don't need to change them, don't!

5.6 HOW TO DEBUG YOUR VISUAL C++ APPLICATIONS

Figure 5.6

Visual C++ has a powerful debugger that's very easy to use. Let's step through our example program. To make it more interesting we'll add an int variable and watch it in a for loop.

Before the line 'std::cout', add the following two lines of code.

```
for (int i=0;i < 5; i++)
std::cout << "i =" << i << "\n\n";
```

Select the first line (for int i...) and press **F9**. This puts a breakpoint there- a red circle in the margin. Now press **F5** to start the debugging. You can exit the debugger at any time by pressing **Shift + F5**.

Without the breakpoint, the program would immediately run to completion and stop. Alternatively you can start a program by pressing either **F10** or **F11**.

You should see the three windows numbered 1-3, in the picture above. 1 Shows local variables, 2 shows the calling stack (for functions) and 3 shows variables you decide to watch. If you can't see these windows, Click "View" on the Menu then "Debug Windows" and then "Watch", "Calling Stack" or "variables" on the sub-menu.

When the program breaks at the **for (int i =0;...** line, the variables windows shows that i has a nonsense value, like -858993460. This has just picked up whatever was in RAM at the address of i. As soon as the loop starts executing, i takes the value 0. Press **F10** to step through execution line by line.

Stepping Through Program Execution

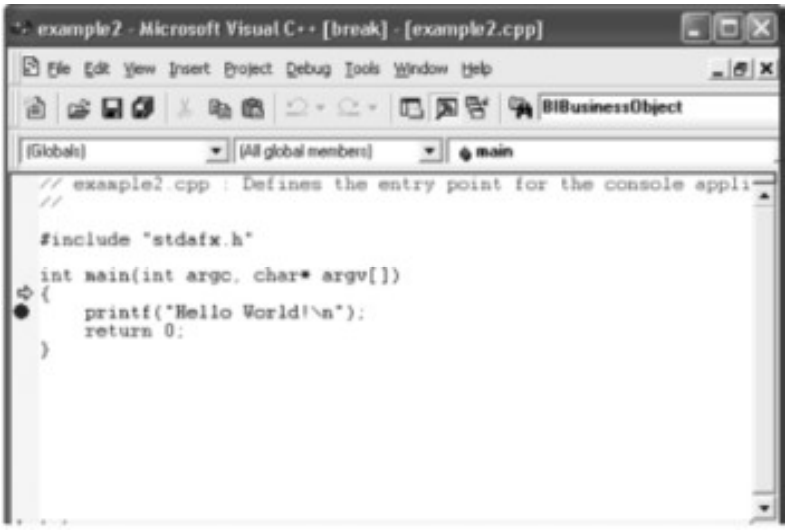


Figure 5.7

F10 and F11

Both of these step the program forward by one line. The current line is highlighted by the yellow arrow in the picture. The red spot is a breakpoint.

The difference between **F10** and **F11** is how functions are dealt with. **F10** will call the function and move on to the next line, whereas **F11** will step into it.

When the cursor is on either of the std::cout lines, pressing **F11** will enter **Ostream**, the library that implements cout. You should avoid system files until you are more proficient and comfortable with templates. Remember when you don't want to step into a function press **F10**.

If you do step into a function by mistake, don't panic. Just press **Shift+F11**, to take you to the end of the function.

The Debugger Windows

The "variables" window shows all the local variables. When stepping through the main function, you'll also see argc and argv which are command line parameters passed into an application.

The "Calling Stack" shows which function called which. When you are many levels deep this can be a life saver. Press **F11** on a cout statement to see this. As the debugger enters the function code, another level is added to the "Calling Stack" window.

The "watch" windows lets you watch variables, useful when those variables are no longer visible in the "variables" window. You can type or cut and paste the variable name into the window.

5.7 MANIPULATING DSP AND DSW FILES DIRECTLY

5.7.1 DSP and DSW Files

All Project settings are kept in a .dsp project file. This includes all configuration information and it's quite readable. Ignore the 3rd line # ** DO NOT EDIT **", but please do take care. Always close the workspace in Visual C++ and make a backup copy before editing the .dsp file.

You can edit a project file using notepad or any text editor and it can be faster to make changes than by using the Settings dialog. For instance you can see where the Debug and Release Directories are defined, and changing those is easy. Adding extra source or header files is not a hard task. Just add these three lines below to the .dsp to add newfile.cpp to your Source Files folder.

```
# Begin Source File
SOURCE=.\newfile.cpp
# End Source File
```

5.7.2 Copying DSP and DSW Files

It's easy to copy .dsp and .dsw files into a new folder. If you are keeping the same project name then there's no need to edit the .dsw file.

If the project has another name, you'll need to edit the .dsp and change all instances of the old name to the new. Also I've found that the Precompiled headers setting can cause problems. You need to load the project, and change the "precompiled headers" setting on the C/C++ tab (choose "Precompiled Headers" on the category Combo) to Create Precompiled header file the first time you build a project. After that change it to "use precompiled header file" (.pch).

REVIEW EXERCISE

1. What are the various types of projects provided in VC++.
2. How to create a new project in VC++.
3. Projects in Visual C++ 6 are organized in Workspaces. Comment
4. Differentiate between .EXE and .DLL project types.
5. How one can run a VC++ program from command line.
6. What is the procedure to debug a VC++ application.

CHAPTER

6 GENERATING A WINDOWS GUI PROGRAM

6.1 PROGRAMMING FOR THE WINDOWS GUI

Microsoft Visual C++ provides several different pathways for writing Windows GUI programs. First, you can write GUI programs in C or C++ by directly calling the functions provided by the underlying Win32 application program interface (API).

Using this approach, however, you must write many lines of routine code before you can begin to focus on the tasks specific to your application. Second, you can write Windows GUI programs in C++ using the Microsoft Foundation Classes. The MFC provides a large collection of prewritten classes, as well as supporting code, which can handle many standard Windows programming tasks, such as creating windows and processing messages. You can also use the MFC to quickly add sophisticated features to your programs, such as toolbars, split window views, and OLE support. And you can use it to create ActiveX controls, which are reusable software components that can be displayed in Web browsers and other container applications. The MFC can simplify your GUI programs and make your programming job considerably easier. Note that the MFC functions internally call Win32 API functions. The MFC is thus said to "wrap" the Win32 API, providing a higher-level, more portable programming interface. (In MFC programs, you're also free to directly call Win32 API functions, so you don't lose their capabilities by choosing to use the MFC.)

Third, you can write Windows GUI programs in C++ using both the MFC and the Microsoft Wizards. You can use AppWizard to generate the basic source files for a

variety of different types of GUI programs. You can then use the ClassWizard tool to generate much of the routine code required to derive classes, to define member functions for processing messages or customizing the behavior of the MFC, to manage dialog boxes, and to accomplish other tasks. The code generated by the Wizards makes full use of the MFC.

Note that the Wizards aren't limited to generating simple program shells, but rather can be used to produce programs containing extensive collections of advanced features, including toolbars, status windows, context-sensitive online help, OLE support, database access, and complete menus with partially or fully functional commands for opening and saving files, printing, print previewing, and performing other tasks. Once you've used the Wizards to generate the basic program source code, you can immediately begin adding code specific to the logic of your program. Using this third approach, you benefit not only from the prewritten code in the MFC, but also from the generated source code that *uses* the MFC and handles many routine programming tasks. The MFC and the Wizards free you from much of the effort required in creating your program's visual interface, and also help ensure that this interface conforms to Microsoft's guidelines.

6.2 CREATING AND BUILDING THE PROGRAM

In this section you'll create a program named WinGreet, which is an example of the simplest type of program that you can generate using AppWizard. You'll first generate the program source code, then make several modifications to the generated code, and finally build and run the program.

6.2.1 Generating the Source Code

To generate a program with AppWizard, you create a new project of the appropriate type, and then specify the desired program features in a series of dialog boxes that AppWizard displays. Begin by running the Microsoft Developer Studio, and then proceed as follows:

1. Choose the File -> New... menu command in Developer Studio or simply press Ctrl+N. The New dialog box will appear.
2. Open the Projects tab (if it's not already open) so that you can create a new project.
3. In the list of project types, select the "MFC AppWizard (exe)" item. Choosing this project type will cause AppWizard to prompt you for further information and then to generate the basic C++ code for a Windows GUI program that uses the MFC. (To create a dynamic link library with AppWizard, you would choose the "MFC AppWizard (dll)" project type. Creating dynamic link libraries isn't covered in this book.

4. Type the name **WinGreet** into the Project Name: text box. This will cause Visual C++ to assign the name WinGreet to the new project (as well as to the project workspace that contains this project).
5. In the Location: text box, specify the path of the folder to contain the project files (that is, the *project folder*). If you wish, you can simply accept the default folder that is initially contained in this box (the default folder is given the same name as the project workspace, WinGreet). Click the button with the ellipsis (...) if you want to search for a different location. If the specified project folder doesn't exist, the Developer Studio will create it (it will also create the -Res subfolder within the project folder to store several resource files, in addition to one or more output subfolders).
6. To complete the Projects tab of the **New** dialog box, make sure that the Win32 item is checked in the Platforms: area. Unless you've installed a cross-development edition of Visual C++, Win32 will be the only option in this area.
7. Click the OK button in the **New** dialog box. The first of the AppWizard dialog boxes, which is labeled "MFC AppWizard - Step 1," will now be displayed. In the following descriptions of the AppWizard options that need to be selected, the expression "(default)" follows the description of each option initially selected. For these options, you need only make sure that you don't change them.
8. In the Step 1 dialog box, select the **Single Document** application type, make sure the Document/View Architecture Support option (default) is checked, and select the English language (default).

   Choosing the Single Document application type causes AppWizard to generate a *single document interface* (SDI) application, which is designed to display only one document at a time. The Document/View Architecture Support option causes AppWizard to generate separate classes for storing and for viewing your program's data, as well as to provide code for reading and writing the data from disk. Finally, AppWizard will use the selected language for the program menu captions and for the standard messages that the program displays. Click the Next > button to display the Step 2 dialog box.
9. In the Step 2 dialog box, select the **None** item (default) to exclude database support from the program.

   Note that in any of the AppWizard dialog boxes (from Step 2 on) you can click the < Back button to return to a previous step to review and possibly revise your choices. Also, you can click the Finish button to skip the remaining dialog boxes and immediately generate the program source code using the default values for all choices in the remaining dialog boxes (*don't* click this button for the current exercise). And finally, you can click the button in the upper-right corner and then click a control in the dialog box to obtain information on the related option.

   Click the Next > button to reveal the Step 3 dialog box.

10. In the Step 3 dialog box, select the None item (default) to exclude compound document support from the program, make sure that the Automation option isn't checked to eliminate automation support, and remove the check from the ActiveX Controls option since you won't be adding any ActiveX controls to the program. Click the Next > button to display the Step 4 dialog box.
11. In the Step 4 dialog box, remove the check from each of the application features except "3D Controls" (default) and leave the value 4 (default) as the number of files you want to use in the "recent file list." You don't need to click the Advanced... button to select advanced options; rather, you'll accept the default values for these options.

    The File menu of the generated program will list the most recently opened documents; the number that you specify for the "recent file list" is the maximum number of documents that will be listed. Click the Next > button to display the Step 5 dialog box.
12. In the Step 5 dialog box, select the MFC Standard project-style option (default) to generate the traditional MFC user interface for your program (the Windows Explorer option implements the application as a workbook-like container). Select the "Yes, Please" option (default) to have AppWizard include comments within the source files it generates. The comments help clarify the code and clearly indicate the places where you need to insert your own code. And finally, choose the As *Statically Linked Library* option for the MFC library that's used. With the As A Statically Linked Library option, the MFC code is bound directly into your program's executable file. With the alternative option, As A Shared DLL, your program accesses MFC code contained in a separate dynamic link library (DLL), which can be shared by several applications (note that you'll have to select this option if you have the Standard Edition of Visual C++, which doesn't provide static MFC linking). The DLL option reduces the size of your program's executable file but requires you to distribute a separate DLL file together with your program file (as you must when you distribute a Visual Basic program). Click the Next > button to display the Step 6 dialog box.
13. The Step 6 dialog box displays information on each of the four main classes that AppWizard defines for your program.. Don't change any of this information because the remainder of the exercise assumes that you've accepted all the default values. This is the final AppWizard dialog box for collecting information; you should now click the Finish button to display the New Project Information dialog box.
14. The New Project Information dialog box, summarizes many of the program features that you chose in the previous dialog boxes. (If you want to change any feature, you can click the Cancel button and then go back to the appropriate dialog box to adjust the information.) Click the OK button in the New Project Information dialog box, and AppWizard will create the project folder that you specified (if necessary), generate the program source files, and open the newly created project, WinGreet.

6.2.2 Modifying the Source Code

The source files generated by AppWizard are sufficient for building a functional program. In other words, immediately after generating the source files with AppWizard, you could build and run the program (although it wouldn't do very much). Before building the program, however, you normally use the various Visual C++ development tools to add to the code features specific to your application.

To provide you with some practice in working with the source files, this section describes how to add code that displays the string "Greetings!" centered within the program window (if the generated code is unaltered, the program simply displays a blank window). To do this, proceed as follows:

1. Open the source file WinGreetDoc.h. The easiest way to open a source file belonging to the current project is to double-click the file name within the File View graph. WinGreetDoc.h is the header file for the program's *document class*, which is named CWinGreetDoc and is derived from the MFC class CDocument. The document class is responsible for reading, writing, and storing the program data. In this trivial example program, the document class simply stores the fixed message string ("Greetings!"), which constitutes the program data.
2. In the CWinGreetDoc class definition you'll add the protected data member m_Message, which stores a pointer to the message string, and you'll add the public member function GetMessage, which returns a pointer to this string. To do this, enter the lines marked in **bold** within the following code:

```
class CWinGreetDoc : public CDocument
{
protected:
char *m_Message;
public:
char *GetMessage ()
{
return m_Message;
}
protected: // create from serialization only
CWinGreetDoc();
DECLARE_DYNCREATE(CWinGreetDoc)
// remainder of CWinGreetDoc definition ...
```

The code excerpt above shows the beginning of the CWinGreetDoc class definition, and includes the code that was generated by AppWizard, as well as the lines of code that you manually add, which are marked in bold. In the instructions given in this part of the book, all lines of code that you manually add or modify are marked in bold. Although you add or modify only the bold lines, the book typically shows a larger block of code to help you find the correct position within the generated source file to make your additions or modifications.

3. Open the file WinGreetDoc.cpp, which is the implementation file for the program's document class, CWinGreetDoc. Within the CWinGreetDoc class constructor, add the statement that's marked in bold in the following code:

```
//////////////////////////////////////////////////////////////////
// CWinGreetDoc construction/destruction
CWinGreetDoc::CWinGreetDoc()
{
// TODO: add one-time construction code here
m_Message = "Greetings!";
}
```

As a result of adding this line, the data member m_Message will automatically be assigned the address of the string "Greetings!" when an instance of the CWinGreetDoc class is created.

4. Open the file WinGreetView.cpp, which is the implementation file for the program's *view* class; this class is named CWinGreetView and is derived from the MFC class CView. As you'll see later, the view class is responsible for processing input from the user and for managing the view window, which is used for displaying the program data.
5. In the file WinGreetView.cpp, add the statements marked in bold to the CWinGreetView member function OnDraw:

```
//////////////////////////////////////////////////////////////////
////////////
// CWinGreetView drawing
void CWinGreetView::OnDraw(CDC* pDC)
{
CWinGreetDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here
```

```
RECT ClientRect;
GetClientRect (&ampClientRect);
pDC->DrawText
(pDoc->GetMessage (), // obtain the string
-1,
&ampClientRect,
DT_CENTER | DT_VCENTER | DT_SINGLELINE);
}
```

The MFC calls the OnDraw member function of the program's view class whenever the program window needs drawing or redrawing (for example, when the window is first created, when its size is changed, or when it's uncovered after being hidden by another window). The code you added to OnDraw displays the string that's stored in the document class ("Greetings!"). OnDraw obtains a pointer to the program's document class by calling the CView member function GetDocument. It then uses this pointer to call the CWinGreetDoc member function GetMessage (which you added to the code in step 2) to obtain the message string. Although this is an elaborate method for getting a simple string, it's used here because it illustrates the typical way that the view class obtains program data from the document class, so that it can display this data. OnDraw is passed a pointer to a *device context object* that is an instance of the MFC class CDC. A device context object is associated with a specific device (in WinGreet it's associated with the view window), and it provides a set of member functions for displaying output on that device. OnDraw uses the CDC member function DrawText to display the message string. To center the string within the view window, it calls the CWnd member function GetClientRect to obtain the current dimensions of the view window, and then supplies these dimensions (in a RECT structure) to DrawText, together with a set of flags that cause DrawText to center the string horizontally and vertically within the specified dimensions (DT_CENTER and DT_VCENTER). In a full-featured application, you would of course make many more changes to the source code generated by AppWizard, typically using a variety of tools, including the resource editors and ClassWizard.

6.2.3 Building and Running the Program

To build the program, choose the Build -> Build WinGreet.exe menu command on the Build menu, or press F7, or click the Build button on the Build toolbar or Build MiniBar:

If the build process completes without error, you can run the program by choosing the

Build -> Execute

WinGreet.exe menu command, or by pressing Ctrl+F5, or by clicking the Execute Program button.

When you run the program, notice that AppWizard has created code for displaying a complete menu. The Exit command on the File menu and the About command on the Help menu are fully functional; that is, AppWizard has generated all the code needed to implement these commands. The commands on the Edit menu are nonfunctional; that is, AppWizard hasn't supplied any of the code for implementing these commands, and therefore they're disabled.

The commands on the File menu (other than Exit) are partially functional. That is, AppWizard has generated some of the code needed to implement the commands. If you select the Open... command, the program displays the standard Open dialog box. If you select a file in this dialog box and click OK, the name of the file is displayed in the window title bar (replacing the name "Untitled" that's displayed when the program first starts), but the contents of the file aren't actually read or displayed. If you then choose the **New** command, the program again displays the name "Untitled" in the title bar, but it doesn't actually initialize a new document

Finally, if you choose the Save As... command (or the Save command with an "Untitled" document), the AppWizard code will display the Save As dialog box. If you specify a file name and click OK, the program will create an empty file having the specified name, but won't write any data to this file.

If you "open" several files using the Open... command, you'll notice that the File menu displays a list of the most recently "opened" files (it will list up to four files). When you quit the program, the AppWizard code saves this list in the Windows Registry so that it can restore the list the next time you run the program.

6.3 THE PROGRAM CLASSES AND FILES

The WinGreet program is known as a *single document interface* (or SDI) application, meaning that it displays only one document at a time. When AppWizard generates an SDI application, it derives four main classes:

- The document class.
- The view class.
- The main frame window class.
- The application class.

The primary program tasks are divided among these four main classes, and AppWizard creates separate source files for each class. By default, it derives the names of both the classes and the class source files from the name of the project (though, as mentioned previously, you can specify alternative names when using AppWizard to generate the program). The WinGreet document class is named CWinGreetDoc and is derived from the MFC class CDocument. The CWinGreetDoc header file is named WinGreetDoc.h and the implementation file is named WinGreetDoc.cpp.

The document class is responsible for storing the program data as well as for reading and writing this data to disk files. The WinGreet document class stores only a single message string and doesn't perform disk I/O.

The WinGreet view class is named CWinGreetView and is derived from the MFC class CView. The CWinGreetView header file is named WinGreetView.h, and the implementation file is named WinGreetView.cpp. The view class is responsible for displaying the program data (on the screen, printer, or other device) and for processing input from the user. This class manages the *view window*, which is used for displaying program data on the screen. The WinGreet view class merely displays the message string within The WinGreet main frame window class is named CMainFrame and is derived from the MFC class CFrameWnd.

The CMainFrame header file is named MainFrm.h, and the implementation file is named MainFrm.cpp. The main frame window class manages the main program window, which is a *frame* window that contains a window frame, a title bar, a menu bar, and a system menu. The frame window also contains Minimize, Maximize, and Close boxes, and sometimes other user interface elements such as a toolbar or a status bar.

Note that the view window—managed by the view class—occupies the empty portion of the main frame window inside these interface elements (which is known as the *client area* of the main frame window). The view window has no visible elements except the text and graphics that the view class explicitly displays (such as the string "Greetings!" displayed by WinGreet). The view window is a *child* of the main frame window, which means—among other things—that it's always displayed on top of and within the boundaries of the client area of the main frame window.

Finally, the application class is named CWinGreetApp and is derived from the MFC class CWinApp. The CWinGreetApp header file is named WinGreet.h, and the implementation file is named WinGreet.cpp. The application class manages the program as a whole; that is, it performs general tasks that don't fall within the province of any of the other three classes, such as initializing the program and performing the final program cleanup. Every MFC Windows program must create exactly one instance of a class derived from CWinApp.

The four main classes communicate with each other and exchange data by calling each other's public member functions and by sending *messages*

AppWizard and the Developer Studio create several source and settings files in addition to the source files for the four main classes

The following listings provide the complete text of the header and implementation files for the four main program classes. These listings contain the code that was generated by AppWizard, plus the manual code additions.

Listing 1

```
// WinGreet.h : main header file for the WINGREET application
//
#if
!defined(AFX_WINGREET_H__E7D60DA4_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
#define
AFX_WINGREET_H__E7D60DA4_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif
#include "resource.h" // main symbols
////////////////////////////////////////////////////////////////////////////
// CWinGreetApp:
// See WinGreet.cpp for the implementation of this class
Title
_________
//
class CWinGreetApp : public CWinApp
{
public:
CWinGreetApp();
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CWinGreetApp)
public:
virtual BOOL InitInstance();
//}}AFX_VIRTUAL
// Implementation
//{{AFX_MSG(CWinGreetApp)
afx_msg void OnAppAbout();
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
```

```
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.
#endif
/                                                                              /
!defined(AFX_WINGREET_H__E7D60DA4_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
```

Listing 2

```
// WinGreet.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "WinGreet.h"
#include "MainFrm.h"
#include "WinGreetDoc.h"
#include "WinGreetView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CWinGreetApp
BEGIN_MESSAGE_MAP(CWinGreetApp, CWinApp)
//{{AFX_MSG_MAP(CWinGreetApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

```
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CWinGreetApp construction
CWinGreetApp::CWinGreetApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}
/////////////////////////////////////////////////////////////////////////////
// The one and only CWinGreetApp object
CWinGreetApp theApp;
/////////////////////////////////////////////////////////////////////////////
// CWinGreetApp initialization
BOOL CWinGreetApp::InitInstance()
{
// Standard initialization
// If you are not using these features and wish to reduce the size
// of your final executable, you should remove from the following
// the specific initialization routines you do not need.
#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
// Change the registry key under which our settings are stored.
// You should modify this string to be something appropriate
// such as the name of your company or organization.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings(); // Load standard INI file options (including MRU)
// Register the application's document templates. Document templates
// serve as the connection between documents, frame windows and views.
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CWinGreetDoc),
```

```
RUNTIME_CLASS(CMainFrame), // main SDI frame window
RUNTIME_CLASS(CWinGreetView));
AddDocTemplate(pDocTemplate);
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(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;
}
////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About
class CAboutDlg : public CDialog
{
public:
CAboutDlg();
// Dialog Data
//{{AFX_DATA(CAboutDlg)
enum { IDD = IDD_ABOUTBOX };
//}}AFX_DATA
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CAboutDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
//}}AFX_VIRTUAL
// Implementation
protected:
//{{AFX_MSG(CAboutDlg)
// No message handlers
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
```

```
};
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
//{{AFX_DATA_INIT(CAboutDlg)
//}}AFX_DATA_INIT
}
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)
//}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//{{AFX_MSG_MAP(CAboutDlg)
// No message handlers
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
// App command to run the dialog
void CWinGreetApp::OnAppAbout()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal();
}
/////////////////////////////////////////////////////////////////////////////
// CWinGreetApp commands
```

Listing 3

```
// WinGreetDoc.h : interface of the CWinGreetDoc class
//
/////////////////////////////////////////////////////////////////////////////
#if
!defined(AFX_WINGREETDOC_H__E7D60DAA_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
#define
AFX_WINGREETDOC_H__E7D60DAA_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_
#if _MSC_VER > 1000
```

```
#pragma once
#endif // _MSC_VER > 1000
Title
_____
class CWinGreetDoc : public CDocument
{
protected:
char *m_Message;
public:
char *GetMessage ()
{
return m_Message;
}
protected: // create from serialization only
CWinGreetDoc();
DECLARE_DYNCREATE(CWinGreetDoc)
// Attributes
public:
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CWinGreetDoc)
public:
virtual BOOL OnNewDocument();
virtual void Serialize(CArchive& ar);
//}}AFX_VIRTUAL
// Implementation
public:
virtual ~CWinGreetDoc();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
```

```
// Generated message map functions
protected:
//{{AFX_MSG(CWinGreetDoc)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.
#endif
//
!defined(AFX_WINGREETDOC_H__E7D60DAA_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
```

Listing 4

```
// WinGreetDoc.cpp : implementation of the CWinGreetDoc class
//
#include "stdafx.h"
#include "WinGreet.h"
#include "WinGreetDoc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CWinGreetDoc
IMPLEMENT_DYNCREATE(CWinGreetDoc, CDocument)
BEGIN_MESSAGE_MAP(CWinGreetDoc, CDocument)
//{{AFX_MSG_MAP(CWinGreetDoc)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
////////////////////////////////////////////////////////////////////////////
// CWinGreetDoc construction/destruction
CWinGreetDoc::CWinGreetDoc()
{
// TODO: add one-time construction code here
m_Message = "Greetings!";
}
CWinGreetDoc::~CWinGreetDoc()
{
}
BOOL CWinGreetDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
return TRUE;
}
////////////////////////////////////////////////////////////////////////////
// CWinGreetDoc serialization
void CWinGreetDoc::Serialize(CArchive& ar)
{
if (ar.IsStoring())
{
// TODO: add storing code here
}
else
{
// TODO: add loading code here
}
}
////////////////////////////////////////////////////////////////////////////
// CWinGreetDoc diagnostics
#ifdef _DEBUG
```

```
void CWinGreetDoc::AssertValid() const
{
CDocument::AssertValid();
}
void CWinGreetDoc::Dump(CDumpContext& dc) const
{
CDocument::Dump(dc);
}
#endif //_DEBUG
/////////////////////////////////////////////////////////////////////////////
// CWinGreetDoc commands
```

Listing 5

```
// MainFrm.h : interface of the CMainFrame class
#if
!defined(AFX_MAINFRM_H__E7D60DBB_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
#define
AFX_MAINFRM_H__E7D60DBB_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
CMainFrame();
DECLARE_DYNCREATE(CMainFrame)
// Attributes
public:
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CMainFrame)
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
//}}AFX_VIRTUAL
```

```
// Implementation
public:
virtual ~CMainFrame();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
// Generated message map functions
protected:
//{{AFX_MSG(CMainFrame)
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.
#endif
//!defined(AFX_MAINFRM_H__E7D60DBB_9891_11D1_
                                                80FC_00C0F6A83B7F__INCLUDED_)
```

Listing 6

```
// MainFrm.cpp : implementation of the CMainFrame class
//
#include “stdafx.h”
#include “WinGreet.h”
#include “MainFrm.h”
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
```

```
// CMainFrame
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction
CMainFrame::CMainFrame()
{
// TODO: add member initialization code here
}
CMainFrame::~CMainFrame()
{
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
if( !CFrameWnd::PreCreateWindow(cs) )
return FALSE;
// TODO: Modify the Window class or styles here by modifying
// the CREATESTRUCT cs
return TRUE;
}
/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
CFrameWnd::Dump(dc);
```

```
}
#endif //_DEBUG
/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
```

Listing 7

```
// WinGreetView.h : interface of the CWinGreetView class
//
/////////////////////////////////////////////////////////////////////////////
#if
!defined(AFX_WINGREETVIEW_H__E7D60DAC_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
#define
AFX_WINGREETVIEW_H__E7D60DAC_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class CWinGreetView : public CView
{
protected: // create from serialization only
CWinGreetView();
DECLARE_DYNCREATE(CWinGreetView)
// Attributes
public:
CWinGreetDoc* GetDocument();
// Operations
public:
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CWinGreetView)
public:
virtual void OnDraw(CDC* pDC); // overridden to draw this view
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
//}}AFX_VIRTUAL
// Implementation
```

```
public:
Title
_____
virtual ~CWinGreetView();
#ifdef _DEBUG
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
#endif
protected:
// Generated message map functions
protected:
//{{AFX_MSG(CWinGreetView)\
// NOTE - the ClassWizard will add and remove member functions here.
// DO NOT EDIT what you see in these blocks of generated code !
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
#ifndef _DEBUG // debug version in WinGreetView.cpp
inline CWinGreetDoc* CWinGreetView::GetDocument()
{ return (CWinGreetDoc*)m_pDocument; }
#endif
////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.
#endif
//!defined(AFX_WINGREETVIEW_H__E7D60DAC_9891_11D1_80FC_00C0F6A83B7F__INCLUDED_)
```

Listing 8

```
// WinGreetView.cpp : implementation of the CWinGreetView class
//
#include “stdafx.h”
#include “WinGreet.h”
#include “WinGreetDoc.h”
#include “WinGreetView.h”
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CWinGreetView
IMPLEMENT_DYNCREATE(CWinGreetView, CView)
BEGIN_MESSAGE_MAP(CWinGreetView, CView)
//{{AFX_MSG_MAP(CWinGreetView)
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CWinGreetView construction/destruction
CWinGreetView::CWinGreetView()
{
// TODO: add construction code here
}
CWinGreetView::~CWinGreetView()
{
}
BOOL CWinGreetView::PreCreateWindow(CREATESTRUCT& cs)
{
// TODO: Modify the Window class or styles here by modifying
// the CREATESTRUCT cs
return CView::PreCreateWindow(cs);
}
/////////////////////////////////////////////////////////////////////////////
// CWinGreetView drawing
void CWinGreetView::OnDraw(CDC* pDC)
{
CWinGreetDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
```

```
// TODO: add draw code for native data here
RECT ClientRect;
GetClientRect (&ampClientRect);
pDC->DrawText
(pDoc->GetMessage (), // obtain the string
-1,
&ampClientRect,
DT_CENTER | DT_VCENTER | DT_SINGLELINE);
}
/////////////////////////////////////////////////////////////////////////////
// CWinGreetView diagnostics
#ifdef _DEBUG
void CWinGreetView::AssertValid() const
{
CView::AssertValid();
}
void CWinGreetView::Dump(CDumpContext& dc) const
{
CView::Dump(dc);
}
CWinGreetDoc* CWinGreetView::GetDocument() // non-debug version is inline
{
ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CWinGreetDoc)));
return (CWinGreetDoc*)m_pDocument;
}
#endif //_DEBUG
/////////////////////////////////////////////////////////////////////////////
// CWinGreetView message handlers
```

6.4 HOW THE PROGRAM WORKS

If you're accustomed to procedural programming for MS-DOS or Unix, or even if you're familiar with conventional Windows GUI programming, you might be wondering how the WinGreet program works—where it first receives control, what it does next, where it exits, and so on. This section briefly describes the overall flow of control of the program, and then discusses the tasks performed by the application initialization function, InitInstance.

The following is a list of some of the significant events that occur, when you run the WinGreet program. These five events were selected from the many program actions that take place, because they best help you understand how the WinGreet program works and they illustrate the purpose of the different parts of the source code:

1. The CWinApp class constructor is called.

2. The program entry function, WinMain, receives control.

3. WinMain calls the program's InitInstance function.

4. WinMain enters a loop for processing messages.

5. WinMain exits and the program terminates.

1. The *CWinApp* Constructor Is Called: As mentioned previously, an MFC application must define exactly one instance of its application class. The file WinGreet.cpp defines an instance of the WinGreet application class, CWinGreetApp, in the following global definition:

```
////////////////////////////////////////////////////////////////
// The one and only CWinGreetApp object
CWinGreetApp theApp;
```

Because the CWinGreetApp object is defined globally, the class constructor is called *before* the program entry function, WinMain, receives control. The CWinGreetApp constructor generated by AppWizard (also in WinGreet.cpp) does nothing:

```
////////////////////////////////////////////////////////////////////////
CWinGreetApp construction
CWinGreetApp::CWinGreetApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}
```

Such a do-nothing constructor causes the compiler to invoke the default constructor of the base class, which is CWinApp. The CWinApp constructor (supplied by the MFC) performs the following two important tasks:

- It makes sure that the program declares only *one* application object (that is, only one object belonging to CWinApp or to a class derived from it).
- It saves the address of the program's CWinGreetApp object in a global pointer declared by the MFC. It saves this address so that the MFC code can later call the WinGreetApp member functions. Calling these member functions will be described under step 3.

2. WinMain Receives Control: After all global objects have been created, the program entry function, WinMain, receives control. This function is defined within the MFC code; it's linked to the WinGreet program when the executable file is built. The WinMain function performs many tasks. The following steps describe the tasks that are the most important for understanding how the WinGreet program works.

3. WinMain Calls InitInstance: Shortly after it receives control, WinMain calls the InitInstance member function of the CWinGreetApp class. It calls this function by using the object address that the CWinApp constructor saved in step 1. InitInstance serves to initialize the application.

4. WinMain Processes Messages: After completing its initialization tasks, WinMain enters a loop that calls the Windows system to obtain and dispatch all *messages* sent to objects within the WinGreet program (this loop is actually contained in a CWinApp member function named Run that's called from WinMain). Control remains within this loop during the remaining time that the application runs. Under Windows 95 (and later) and Windows NT, however, preemptive multitasking allows other programs to run at the same time.

5. WinMain Exits and the Program Terminates: When the user of the WinGreet program chooses the Exit command on the File menu or the Close command on the system menu, or clicks the Close box, the MFC code destroys the program window and calls the Win32 API function: PostQuitMessage, which causes the message loop to exit. The WinMain function subsequently returns, causing the application to terminate.

The InitInstance Function

InitInstance is a member function of the application class, CWinGreetApp, and it's defined in the source file WinGreet.cpp. The MFC calls this function from WinMain, and its job is to initialize the application.

At the time InitInstance is called, a more traditional Windows GUI application would simply create a main program window because of the view-document programming model used by the MFC. However, the AppWizard code does something a bit more complex. It creates a *document template*, which stores information about the program's document class, its main frame window class, and its view class. The document template also stores the identifier of the program resources used in displaying and managing a document (the menu, icon, and so on). When the program first begins running and it creates a new document, it uses the document template to create an object of the document class for storing the document, an object of the view class for creating a view window to display the document, and an object of the main frame window class to provide a main program window for framing the view window. A document template is a C++ object; for an SDI application such as WinGreet, it's an instance of the CSingleDocTemplate MFC class. The following code in InitInstance creates the document template and stores it within the application object:

```
// Register the application's document templates. Document
// templates serve as the connection between documents, frame
// windows, and views.
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
IDR_MAINFRAME,
RUNTIME_CLASS(CWinGreetDoc),
RUNTIME_CLASS(CMainFrame), // main SDI frame window
RUNTIME_CLASS(CWinGreetView));
AddDocTemplate(pDocTemplate);
```

This code works as follows:

- It defines a pointer to a document template object, pDocTemplate.
- It uses the new operator to dynamically create a document template object (that is, an instance of the CSingleDocTemplate class), assigning the object's address to pDocTemplate.
- It passes four parameters to the CSingleDocTemplate constructor. The first parameter is the identifier of the program resources used in displaying and managing a document (namely, the accelerator table, the icon, the menu, and a descriptive string).
- The next three parameters supply information on the document class, the main frame window class, and the view class. Information on each class is obtained by calling the MFC macro RUNTIME_CLASS (which supplies a pointer to a CRuntimeClass object). This information allows the program to dynamically create an object of each class when a new document is first created.
- The template object pointer is passed to the CWinApp member function AddDocTemplate, which stores the document template within the application object so that the template will be available.

After creating the document template, InitInstance extracts the command line—if any—that was passed to the program when it was run, by calling the CWinApp member function ParseCommandLine:

```
// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
```

It then calls the CWinApp member function ProcessShellCommand to process the command line:

```
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
return FALSE;
```

If the command line contains a file name, ProcessShellCommand attempts to open the file. The WinGreet program, however, doesn't fully implement the code for opening a file.

Normally, however, when you run WinGreet (for example, through the Developer Studio), the command line is empty. In this case ProcessShellCommand calls the CWinApp member function OnFileNew to create a new, empty document. When OnFileNew is called, the program uses the document template to create a CWinGreetDoc object, a CMainFrame object, a CWinGreetView object, and the associated main frame window and view window. The resources used for the main frame window (the menu, icon, and so on) are those identified by the resource identifier stored in the document template. Because these objects and windows are created internally by OnFileNew, you don't see within the WinGreet code explicit definitions of the objects, nor do you see function calls for creating windows. Finally, InitInstance calls the ShowWindow and UpdateWindow member functions of the main frame window object to make the main frame window visible on the screen and to cause the window contents to be displayed. It calls these functions by using the pointer to the main frame window object that's stored in the CWinGreetApp object's m_pMainWnd data member (which it inherits from CWinThread):

```
// The one and only window has been initialized, so show and
// update it.
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
```

Other Code in InitInstance

InitInstance calls CWinApp::Enable3dControlsStatic (or Enable3dControls if you choose the shared MFC DLL, as described previously in the chapter) to cause Windows to display controls (such as check boxes) that have a

```
// Standard initialization
// If you are not using these features and wish to reduce the
// size of your final executable, you should remove from the
// following the specific initialization routines you do not
// need.
#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a
// shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC
// statically
#endif
```

InitInstance also calls the CWinApp member function SetRegistryKey, which causes the program settings to be stored in the Windows Registry (rather than in an .ini file) and specifies the name of the key under which these settings are stored:

```
// Change the Registry key under which our settings are stored.
// You should modify this string to be something appropriate
// such as the name of your company or organization.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

To customize the name of the key under which the program settings are stored (for example, to set it to your company name), simply replace the string passed to SetRegistryKey. (Note that the macro _T converts the string to Unicode format, which SetRegistryKey requires. This format stores each character as a 16-bit value, and can be used to encode the characters in any language.)

The primary setting stored in the Registry is the list of most recently opened documents that's displayed on the program's File menu (which is also known as the MRU, or Most Recently Used, file list). InitInstance loads this document list, as well as any other program settings stored in the Registry, by calling the CWinApp::LoadStdProfileSettings function:

```
LoadStdProfileSettings(); // Load standard INI file options
// (including MRU)
```

If you need to perform any other application initialization tasks, the InitInstance function is the place to add the code.

REVIEW EXERCISE

1. What is application program interface (API)?
2. What is the use of MFC classes?
3. Write short notes on:
   (*a*) Active-X Control
   (*b*) OLE
   (*c*) Reusable Software Components.
   (*d*) dynamic link library (DLL)
   (*e*) Single document interface
   (*f*) Multiple document interface
4. Give the use of AppWizard in VC++?
5. When AppWizard generates an SDI application, it derives four main classes: namely
   - The document class
   - The view class
   - The main frame window class
   - The application class

   Describe each of them
6. How a VC++ program works?

CHAPTER 7

WINDOWS, DIALOG BOXES AND CONTROLS

A window in Windows can be defined as a rectangular area on the screen. However, this definition, in all its simplicity, hides the volumes of functionality behind the abstract idea of a window as the primary unit through which a user and a Windows application interact.

A window is not only an area through which an application can present its output; it is also a target of events, a target of messages within the Windows environment. Although the window concept in Windows predates the use of object oriented languages on the PC by several years, the terminology is more than appropriate here: the *properties* of a window determine its appearance, while its *methods* determine how it responds to user input.

A window is identified by a *window handle*. This handle (usually a variable of type HWND) uniquely identifies each window in the system. The list includes the "obvious" application windows and dialog boxes as well as the less obvious ones such as the desktop, certain icons, or buttons. User-interface events are packaged into Windows messages with the appropriate window handle attached and then sent, or queued, to the application (or thread, to be more precise) that owns that window.

Needless to say, Windows offers a lot of functionality covering the creation and management of windows.

7.1 THE WINDOW HIERARCHY

Windows maintains its windows in a hierarchical organization. Each window has a parent and zero or more siblings. At the root of all windows is the *desktop window*,

created by Windows at startup time. The parent window for *top-level* windows is the desktop window; the parent window for *child windows* is either a top-level window or another child window higher up in the hierarchy. Figure 7.1 demonstrates this hierarchy by dissecting a typical Windows screen.

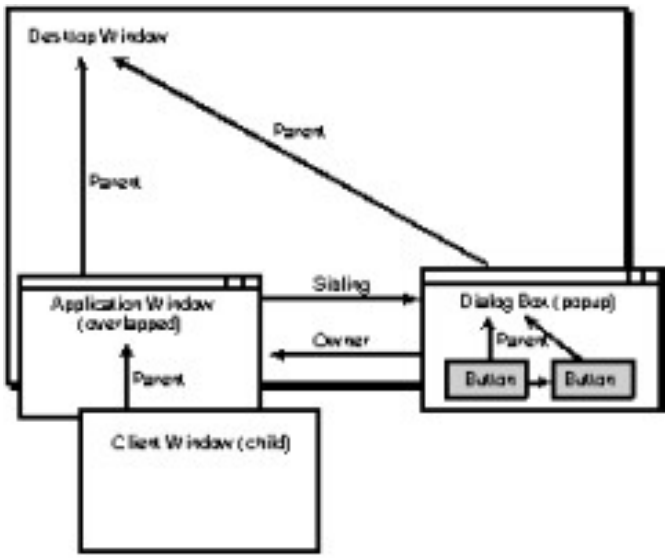


Figure 7.1: The window hierarchy

Actually, the situation under Windows NT is somewhat more complex. Unlike its simpler cousins, Windows NT has the capability to maintain multiple desktops simultaneously. In fact, Windows NT normally maintains three desktops: one for the Winlogon screen, one for user applications, and one for the screen saver.

The visual window hierarchy normally reflects the logical hierarchy. That is, windows at the same hierarchy level are normally displayed in the *Z-order*, which is essentially the order in which siblings appear. However, this order can be changed for top-level windows. Top-level windows with the extended window style WM_EX_TOPMOST appear on top of any non-topmost top-level windows.

Another relationship exists between top-level windows. A top-level window may have an *owner*, which is another top-level window. An owned window always appears on top of its owner and disappears if its owner is minimized. A typical case of a top-level window owned by another occurs when an application displays a dialog box. The dialog box is not a child window (it is not confined to the client area of the application's main window), but it remains owned by the application window.

Several functions enable applications to traverse the window hierarchy and find a specific window. Here's a review of a few of the more frequently used functions:

GetDesktop Window**.** Through the GetDesktopWindow function, an application can retrieve the handle of the current desktop window.

EnumWindows. The EnumWindows function enumerates all top-level windows. A user-defined callback function, the address of which is supplied in the call to EnumWindows, is called once for every top-level window. EnumWindows does not enumerate top-level windows that are created after the function has been called, even if it has not yet completed the enumeration when the new window is created.

EnumChildWindows. The EnumChildWindows function enumerates all child windows of a given window, identified by a handle that is supplied in the call to EnumChildWindows. The enumeration is accomplished by a user-defined callback function, the address of which is also supplied in the call to EnumChildWindows. This function also enumerates descendant windows; that is, child windows that are themselves children (or descendants) of child windows of the window specified in the call to EnumChildWindows.

Child windows that are destroyed before they are enumerated, or child windows that are created after the enumeration process started, will not be enumerated.

EnumThreadWindows. The EnumThreadWindows function enumerates all windows owned by a specific thread by calling a user-supplied callback function once for every such window. The handle to the thread and the address of the callback function are supplied by the application in the call to EnumThreadWindows. The enumeration includes top-level windows, child windows, and descendants of child windows.

Windows that are created after the enumeration process began are not enumerated by EnumThreadWindows.

FindWindow. The FindWindow function can be used to find a top-level window by its window class name or window title.

GetParent. The GetParent function identifies the parent window of the specified window.

GetWindow. The GetWindow function offers the most flexible way for manipulating the window hierarchy. Depending on its second parameter, uCmd, this function can be used to retrieve the handle to a window's parent, owner, sibling, or child windows.

7.2 WINDOW MANAGEMENT

Typically, an application creates a window in two steps. First, the *window class* is registered; next, the window itself is created through the CreateWindow function. The window class determines the overall behavior of the new window type, including most notably the address of the new *window procedure*. Through CreateWindow the application controls minor aspects of the new window, such as its size, position, and appearance.

7.2.1 The *RegisterClass* Function and the *WNDCLASS* Structure

A new window class is registered when an application calls the following function:

```
ATOM RegisterClass(CONST WNDCLASS *lpwc);
```

The single parameter of this function, lpwc, points to a structure of type WNDCLASS describing the new window type. The return value is a *Windows atom*, a 16-bit value identifying a unique character string in a table maintained by Windows.

The WNDCLASS structure is defined as follows:

```
typedef struct _WNDCLASS {
    UINT     style;
    WNDPROC lpfnWndProc;
    int      cbClsExtra;
    int      cbWndExtra;
    HANDLE  hInstance;
    HICON    hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS;
```

The meaning of some of these parameters is fairly straightforward. For example, hIcon is a handle to the icon used to represent minimized windows of this class; hCursor is a handle to the standard mouse cursor that is used when the mouse enters the window rectangle; hbrBackground is a handle to the GDI brush that is used to draw the window's background. The string pointed to by lpszMenuName identifies the menu resource (by name or, through the MAKEINTRESOURCE macro, by an integer identifier) that is used as the standard menu for this class; lpszClassName is the name of the window class.

The parameters cbClsExtra and cbWndExtra can be used to allocate extra memory for the window class or for individual windows. Applications can use this extra memory to store application-specific information pertaining to the window class or individual windows.

The parameter lpfnWndProc specifies the address of the window procedure function. This function is responsible for handling any messages the window receives. It can either handle those messages itself, or invoke the default window procedure,

DefWindowProc. The messages can be anything: window sizing and moving, mouse events, keyboard events, commands, repaint requests, timer and other hardware-related events, and so on.

A typical window procedure contains a large switch statement block. Inside, case blocks exist for every message the application is interested in. Messages that the application does not handle are passed to DefWindowProc through the default block. The skeleton of such a window procedure is shown in Listing 9.1.

Listing 7.1 Window Procedure Skeleton

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
{
   switch(uMsg)
   {
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
      // Other case blocks come here
      default:
         return DefWindowProc(hwnd, uMsg, wParam, lParam);
   }
   return 0;
}
```

Certain global characteristics of the window class are controlled through the class style parameter, style. This parameter may be set to a combination of values (using the bitwise OR operator, |). For example, CS_BYTEALIGNCLIENT specifies that the window's client area is always to be positioned on a byte boundary in the screen display's bitmap to enhance graphics performance (a very useful thing to remember when writing performance-intensive applications intended to run on lower-end graphics hardware). The value CS_DBLCLKS specifies that Windows should generate double-click mouse messages when the user double-clicks the mouse within the window. The pair of values CS_HREDRAW and CS_VREDRAW specify that the window be redrawn in its entirety every time its horizontal or vertical size changes. Or the value CS_SAVEBITS specifies that Windows should allocate what UNIX and X programmers often refer to as *backing store*; a copy of the window bitmap in memory, so that it can

automatically redraw the window when parts of it become unobscured. (This should be used with caution; the large amounts of memory required for this may cause a significant performance hit.)

Note: In 16-bit Windows, it was possible to register an application global class through the style CS_GLOBALCLASS. An application global class was accessible from all other applications and DLLs. This is not true in Win32. In order for an application global class to work as intended, it must be registered from a DLL that is loaded by every application. Such a DLL can be defined through the Registry.

7.2.2 Creating a Window through *CreateWindow*

Registering a new window class is the first step in window creation. Next, applications must actually create a window through the CreateWindow function:

```
HWND CreateWindow(
    LPCTSTR  lpClassName,
    LPCTSTR  lpWindowName,
    DWORD  dwStyle,
    int  x,
    int  y,
    int  nWidth,
    int  nHeight,
    HWND  hWndParent,
    HMENU  hMenu,
    HANDLE  hInstance,
    LPVOID  lpParam
);
```

The first parameter, lpClassName, defines the name of the class that this window inherits its behavior form. The class must either be registered through RegisterClass or be one of the *predefined control classes*. The predefined classes include the BUTTON, COMBOBOX, EDIT, \, SCROLLBAR, and STATIC classes. There are also some window classes that are mostly used internally by Windows and are referenced only through integer identifiers; these include classes for menus, the desktop window, and icon titles, to name but a few.

The dwStyle parameter specifies the window's style. This parameter should not be confused with the class style, passed to RegisterClass through the WNDCLASS structure when the new window class is registered. While the class style determines some of the

permanent properties of windows belonging to that class, the window style passed to CreateWindow is used to initialize the more transient properties of the window. For example, dwStyle can be used to determine the window's initial appearance (minimized, maximized, visible or hidden). As is the case with the class style, the window style is also typically a combination of values (combined with the bitwise OR operator). In addition to the generic style values that are common to all types of windows, some values are specific to the predefined window classes; for example, the BS_PUSHBUTTON style can be used for windows of the BUTTON class that are to send WM_COMMAND messages to their parents when clicked.

Some dwStyle values are important enough to deserve a closer look.

The WS_POPUP and WS_OVERLAPPED styles specify top-level windows. The basic difference is that a WS_OVERLAPPED window always has a caption, while a WS_POPUP window does not need to have one. Overlapped windows are typically used as the main window of applications, while popup windows are used for dialog boxes.

When a top-level window is created, the calling application sets its owner window through the hwndParent parameter. The parent window of a top-level window is the desktop window.

Child windows are created with the WS_CHILD style. The major difference between a child window and a top-level window is that a child window is confined to the client area of its parent.

Windows defines some combinations of styles that are most useful when creating "typical" windows. The WS_OVERLAPPEDWINDOW style setting combines the WS_OVERLAPPED style with the WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles to create a typical top-level application window. The WS_POPUPWINDOW style setting combines WS_POPUP with the WS_BORDER and WS_SYSMENU styles to create a typical dialog box.

7.2.3 Extended Styles and the *CreateWindowEx* Function

The CreateWindowEx function, while otherwise identical to the CreateWindow function, enables you to specify a combination of *extended window styles*. Extended window styles provide finer control over certain aspects of a window's appearance or the way it functions.

For example, through the WS_EX_TOPMOST style applications can make a window a topmost window; that is, a top-level window that is not obscured by other top-level windows. A window created with the WS_EX_TRANSPARENT style does not obscure other windows and only receives a WM_PAINT message when all windows under it have been updated.

Other extended window styles are specific to Windows 95 and versions of Windows NT later than 3.51; for example, Windows NT 3.51 with the beta version of the Windows 95 style shell installed. For example, the WS_EX_TOOLWINDOW style can be used to create a tool window. A tool window is a window with a smaller than usual title bar and other properties that make it useful as a floating toolbar window.

Yet another set of Windows 95 specific extended styles specifies the window's behavior with respect to the selected shell language. For example, the WS_EX_RIGHT, WS_EX_RTLREADING, and WS_EX_LEFTSCROLLBAR extended styles can be used in conjunction with a right-to-left shell language selection such as Hebrew or Arabic.

7.3 PAINTING WINDOW CONTENTS

Painting in a window is performed through the normal set of GDI drawing functions. Applications usually obtain a handle to the display device context through a function such as GetDC, and then call GDI functions such as LineTo, Rectangle, or TextOut.

But even more typically, window painting occurs in response to a specific message, WM_PAINT.

7.3.1 The *WM_PAINT* Message

The WM_PAINT message is sent to a window when parts of it require redrawing by the application and no other message is pending in the message queue of the thread that owns the window. Applications typically respond to this with a set of drawing instructions enclosed between calls to the BeginPaint and EndPaint functions.

The BeginPaint function retrieves a set of parameters that are stored in a PAINTSTRUCT structure:

```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

BeginPaint also takes care of erasing the background, if necessary, by sending the application a WM_ERASEBKGND message.

Note: The BeginPaint function should only be called in response to a WM_PAINT message. Each call to BeginPaint must be accompanied by a subsequent call to the EndPaint function.

Applications can use the hDC member of the structure to draw into the client area of the window. The rcPaint member represents the smallest rectangle that encloses all areas of the window that require updating. By limiting their activities to this rectangular region, applications can speed up the painting process.

7.3.2 Repainting a Window by Invalidating its Contents

The functions InvalidateRect and InvalidateRgn can be used to invalidate all or parts of a window. Windows sends a WM_PAINT message to a window if its *update region*, that is, the union of all update regions specified in prior calls to InvalidateRect and InvalidateRgn, is not empty and the thread that owns the window has no more messages in its message queue.

This behavior suggests a very efficient mechanism for applications that need to update parts of their window. Instead of updating the window immediately, they can schedule the update by invalidating the appropriate region. When they process WM_PAINT messages, they can examine the update region (the rcPaint member of the PAINTSTRUCT structure) and update only those elements in the window that fall into this region. Alternatively (or in addition to this), applications can maintain private variables in which they store *hints;* that is, information that assists the window updating procedure in determining the most efficient way of updating the window.

The use of such hints to assist in efficiently updating a window is present throughout the Microsoft Foundation Classes.

7.4 WINDOW MANAGEMENT MESSAGES

A typical window responds to many other messages in addition to WM_PAINT messages. Some of the more frequently processed messages are reviewed in this section.

WM_CREATE. The first message that the window procedure of a newly created window receives is the WM_CREATE message. This message is sent before the window is made visible and before the CreateWindow or CreateWindowEx function returns.

In response to this message, applications can perform initialization functions that are necessary before the window is made visible.

WM_DESTROY. The WM_DESTROY message is sent to the window procedure of a window that has already been removed from the screen and is about to be destroyed.

WM_CLOSE. The WM_CLOSE message is sent to a window indicating that the window should be closed. The default implementation in DefWindowProc calls DestroyWindow when this message is received. Applications can, for example, display

a confirmation dialog and call DestroyWindow only if the user confirms closing the window.

WM_QUIT. The WM_QUIT message is usually the last message an application's main window receives. Receiving this message causes GetMessage to return zero, which terminates the message loop of most applications.

This message indicates a request to terminate the application. It is generated in response to a call to PostQuitMessage.

WM_QUERYENDSESSION. The WM_QUERYENDSESSION notifies the application that the Windows session is about to be ended. An application may return FALSE in response to this message to prevent the shutdown of Windows. After processing the WM_QUERYENDSESSION message, Windows sends all applications a WM_ENDSESSION message with the results of the WM_QUERYENDSESSION processing.

WM_ENDSESSION. The WM_ENDSESSION message is sent to applications after the WM_QUERYENDSESSION message has been processed. It indicates whether Windows is about to shut down or whether the shutdown has been aborted.

If an imminent shutdown is indicated, the Windows session may end at any time after the WM_ENDSESSION message has been processed by all applications. It is important, therefore, that applications perform all tasks pertaining to safe termination.

WM_ACTIVATE. The WM_ACTIVATE message indicates when a top-level window is about to be activated or deactivated. The message is first sent to the window that is about to be deactivated, then to the window that is about to be activated.

WM_SHOWWINDOW. The WM_SHOWWINDOW message indicates when a window is about to be hidden or shown. A window can be hidden as a result of a call to the ShowWindow function, or as a result of another window being maximized.

WM_ENABLE. The WM_ENABLE message is sent to a window when it is enabled or disabled. A window can be enabled or disabled through a call to the EnableWindow function. A window that is disabled cannot receive mouse or keyboard input.

WM_MOVE. The WM_MOVE message indicates that the window's position has been changed.

WM_SIZE. The WM_SIZE message indicates that the window's size has been changed.

WM_SETFOCUS. The WM_SETFOCUS message indicates that the window has gained keyboard focus. An application may, for example, display the caret in response to this message.

WM_KILLFOCUS. The WM_KILLFOCUS message indicates that the window is about to lose keyboard focus. If the application displays a caret, the caret should be destroyed in response to this message.

WM_GETTEXT. The WM_GETTEXT message is sent to a window requesting that the window text be copied to a buffer. For most windows, the window text is the window title. For controls like buttons, edit controls, static controls, or combo boxes, the window text is the text displayed in the control. This message is usually handled by the DefWindowProc function.

WM_SETTEXT. The WM_SETTEXT message requests that the window text be set to the contents of a buffer. The DefWindowProc function sets the window text and displays it in response to this message.

Several messages concern the nonclient area of a window; that is, its title bar, border, menu, and other areas that are typically not updated by the application program. An application can intercept these messages to create a window frame with a customized appearance or behavior.

WM_NCPAINT. The WM_NCPAINT message indicates that the nonclient area of a window (the window frame) needs to be repainted. The DefWindowProc function handles this message by repainting the window frame.

WM_NCCREATE. Before the WM_CREATE message is sent to a window, it also receives a WM_NCCREATE message. Applications may intercept this message to perform initializations specific to the nonclient area of the window.

WM_NCDESTROY. The WM_NCDESTROY message indicates that a window's nonclient area is about to be destroyed. This message is sent to a window after the WM_DESTROY message.

WM_NCACTIVATE. The WM_NCACTIVATE message is sent to a window to indicate that its nonclient area has been activated or deactivated. The DefWindowProc function changes the color of the window title bar to indicate an active or inactive state in response to this message.

7.5 WINDOW CLASSES

Every window is associated with a window class. A window class is either a class provided by Windows, or a user-defined window class registered through the RegisterClass function.

7.5.1 The Window Procedure

The purpose of a window class is to define the characteristics and behavior of a set of related windows. Perhaps the most notable, but by far not the only property of a window class, is the window procedure.

The window procedure is called every time a message is sent to the window through the SendMessage function, and every time a posted message is dispatched through the DispatchMessage function. The role of the window procedure is to process messages

sent or posted to that window. In doing so, it can rely on the default window procedure (DefWindowProc, or in the case of dialog boxes, DefDlgProc) for the processing of unwanted messages.

It is through the window procedure that the behavior of a window is implemented. By responding to various messages, the window procedure determines how the window reacts to mouse and cursor events and how its appearance changes in reaction to those events. For example, in the case of a button, the window procedure may respond to WM_LBUTTONDOWN messages by repainting the window indicating that the button is pressed. Or in the case of an edit control, the window procedure may respond to a WM_SETFOCUS message by displaying the caret.

Windows supplies two default window procedures: DefWindowProc and DefDlgProc. The DefWindowProc function implements the default behavior for typical top-level windows. It processes nonclient area messages and manages the window frame. It also implements some other aspects of top-level window behavior, such as responding to keyboard events; for example, responding to the Alt key by highlighting the first item in the window's menu bar.

The DefDlgProc function is for the use of dialog boxes. In addition to the default top-level window behavior, it also manages the focus within a dialog box. It implements the behavior of dialogs whereby the focus jumps from one dialog control to the next when the user presses the Tab key.

In addition to the default window procedures, Windows also supplies a set of window classes. These implement the behavior of dialog box controls, such as buttons, edit fields, list and combo boxes, and static text fields. The name for these classes is *system global class*, which is a leftover from the days of 16-bit Windows. In Win32 these classes are no longer global. That is, a change that affects a system global class will only affect windows of that class within the same application and have no effect on windows in another application because Win32 applications run in separate address spaces, and thus they are shielded from one another.

Whether it is a Windows-supplied class, or a class defined by the application, an application can use an existing window class from which to derive a new class and implement new or modified behavior. The mechanisms for accomplishing this are called subclassing and superclassing.

Warning: An application should not attempt to subclass or superclass a window that belongs to another process.

7.5.2 Subclassing

Subclassing means substituting the window procedure for a window class with another. This is accomplished by calling the SetWindowLong or SetClassLong function.

Calling SetWindowLong with the GWL_WNDPROC index value substitutes the window procedure for a specific window. In contrast, calling SetClassLong with the GCL_WNDPROC index value substitutes the window procedure for all windows of that class that are created after the call to SetClassLong.

Consider the simple example shown in Listing 7.2. (You can compile this code from the command line by typing cl subclass.c user32.lib.) This example displays the "Hello, World!" message. In a somewhat unorthodox fashion, it uses the BUTTON system class for this purpose. However, it subclasses the BUTTON class by providing a replacement window procedure. This replacement procedure implements special behavior when a WM_LBUTTONUP message is received; it destroys the window, effectively ending the application. To ensure proper termination, the WM_DESTROY message also receives special handling: a WM_QUIT message is posted through a call to PostQuitMessage.

Listing 7.2 Subclassing the BUTTON Class

```
#include <windows.h>
WNDPROC OldWndProc;
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
{
   switch(uMsg)
   {
      case WM_LBUTTONUP:
         DestroyWindow(hwnd);
         break;
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
      default:
         return CallWindowProc(OldWndProc,
                               hwnd, uMsg, wParam, lParam);
   }
   return 0;
}
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE d2,
                                    LPSTR d3, int d4)
{
   MSG msg;
   HWND hwnd;
   hwnd = CreateWindow("BUTTON", "Hello, World!",
                    WS_VISIBLE | BS_CENTER, 100, 100, 100, 80,
                    NULL, NULL, hInstance, NULL);
   OldWndProc =
      (WNDPROC)SetWindowLong(hwnd, GWL_WNDPROC, (LONG)WndProc);
   while (GetMessage(&msg, NULL, 0, 0))
      DispatchMessage(&msg);
   return msg.wParam;
}
```

I would like to call your attention to the mechanism used in the new window procedure, WndProc, to reference the old window procedure for the default processing of messages. The old procedure is called through the Win32 function CallWindowProc. In 16-bit Windows, it was possible to call the address obtained by the call to SetWindowLong directly; this was always the address of the old window procedure. In Win32, this is not necessarily so; the value may instead be a handle to the window procedure.

In this example, I performed the subclassing through SetWindowLong, meaning that it only affected the single button window for which SetWindowLong was called. If I had called SetClassLong instead, I would have altered the behavior of all buttons created subsequently. Consider the example program in Listing 7.3 (to compile this program from the command line, type cl subclass.c user32.lib).

Listing 7.3 Subclassing the BUTTON Class

```
#include <windows.h>
WNDPROC OldWndProc;
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                    WPARAM wParam, LPARAM lParam)
{
   switch(uMsg)
```

```
    {
        case WM_LBUTTONDOWN:
            MessageBeep(0xFFFFFFFF);
        default:
            return CallWindowProc(OldWndProc,
                                  hwnd, uMsg, wParam, lParam);
    }
    return 0;
}
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE d2, LPSTR d3, int d4)
{
    HWND hwnd;
    hwnd = CreateWindow("BUTTON", " ",
                        0, 0, 0, 0, 0,
                        NULL, NULL, hInstance, NULL);
    OldWndProc =
        (WNDPROC)SetClassLong(hwnd, GCL_WNDPROC, (LONG)WndProc);
    DestroyWindow(hwnd);
    MessageBox(NULL, "Hello, World!", " ", MB_OK);
}
```

This example creates a button control but never makes it visible; the sole purpose of this control's existence is so that through its handle, the class behavior can be modified. Immediately after the call to SetClassLong, the button control is actually destroyed.

But the effects of SetClassLong linger on! The subsequently displayed message box contains an OK button; and the behavior of this button (namely that when it is clicked by the left mouse button, the PC speaker emits a short beep) reflects the new window procedure. Similarly, if the program displayed other dialogs or message boxes, indeed anything that had button controls in it, all the newly created buttons would exhibit the modified behavior.

7.5.3 Global Subclassing

In 16-bit Windows, a subclassing mechanism similar to that presented in the previous section was often used to change the *system-wide* behavior of certain types of windows

such as dialog controls. (This is how the 3-D control library CTL3D.DLL was implemented.) Subclassing the window class affected all newly created windows of that class, regardless of the application that created them. Unfortunately, in Win32 this is no longer the case; only windows of the same application are affected by such a change.

So how can developers influence the global behavior of certain types of windows? The answer is, you have to use a DLL and ensure that it is loaded into every application's address space.

Under Windows NT, this can be accomplished easily by creating a setting in the registry. The following registry value needs to be modified:

\HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\APPINIT_DLLS

DLLs that are listed under this registry key are loaded into the address space of every newly created process. If you wish to add several DLLs, separate the pathnames by spaces.

Listing 9.4 shows a DLL that subclasses the BUTTON class just like the example shown in Listing 9.3. If you add the full pathname of this DLL to the above-mentioned registry key, every time a button control is clicked, a short beep will be heard.

Listing 7.4 Subclassing in a DLL

```
#include <windows.h>

WNDPROC OldWndProc;

LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
{
   switch(uMsg)
   {
      case WM_LBUTTONDOWN:
         MessageBeep(0xFFFFFFFF);
      default:
         return CallWindowProc(OldWndProc,
                               hwnd, uMsg, wParam, lParam);
   }
   return 0;
```

```
}
BOOL WINAPI DllMain (HANDLE hModule, DWORD dwReason,
                    LPVOID lpReserved)
{
   HWND hwnd;
   switch(dwReason)
   {
      case DLL_PROCESS_ATTACH:
         hwnd = CreateWindow("BUTTON", "",
                             0, 0, 0, 0, 0,
                             NULL, NULL, hModule, NULL);
         OldWndProc = (WNDPROC)SetClassLong(hwnd, GCL_WNDPROC,
                                            (LONG)WndProc);
         DestroyWindow(hwnd);
   }
   return TRUE;
}
```

To compile this DLL from the command line, use cl /LD beepbtn.c user32.lib. The /LD command line flag instructs the compiler to create a DLL instead of an executable file.

Warning: Be careful to only add a fully tested DLL to the Registry. A faulty DLL may render your system unstable or may prevent it from starting altogether. If that happens, a quick-and-dirty remedy is to boot into MS-DOS and rename the DLL file to prevent it from being loaded. Obviously, if your DLL file sits on an NTFS partition, this may not be so easy to do.

Adding your DLL's pathname to the APPINIT_DLLS Registry key is perhaps the simplest, but certainly not the only technique to inject your DLL's code into another application's address space. Another drawback of this technique includes the fact that a DLL specified this way is loaded into the address space of every application—or, to be more precise, every GUI application that links with USER32.DLL. Even the slightest bug in your DLL may seriously affect the stability of the entire system.

Fortunately, there are other techniques available that enable you to inject your DLL into the address space of another process.

The first such technique requires the use of a Windows hook function. By using the SetWindowsHookEx function, it is possible to install a hook function into the another application's address space. Through this mechanism, you can add a new window function to a window class owned by another application.

The second technique relies on the CreateRemoteThread function and its ability to create a thread that runs in the context of another process.

7.5.4 Superclassing

Superclassing means creating a new class based on the behavior of an existing class. An application that wishes to superclass an existing class can use the GetClassInfo function to obtain a WNDCLASS structure describing that class. After this structure has been suitably modified, it can be used in a call to the RegisterClass function that registers the new class for use.

The example shown in Listing 9.5 demonstrates the technique of superclassing. In this example, a new window class, BEEPBUTTON, is created, its behavior based on the default BUTTON class. This new class is then used to display a simple message. To compile this program from the command line, type cl supercls.c user32.lib.

Listing 7.5 Superclassing the BUTTON Class

```
#include <windows.h>
WNDPROC OldWndProc;
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
{
   switch(uMsg)
   {
      case WM_LBUTTONDOWN:
         MessageBeep(0xFFFFFFFF);
      default:
         return CallWindowProc(OldWndProc,
                               hwnd, uMsg, wParam, lParam);
   }
   return 0;
}
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE d2,
                                LPSTR d3, int d4)
{
   MSG msg;
   HWND hwnd;
   WNDCLASS wndClass;
   GetClassInfo(hInstance, "BUTTON", &wndClass);
   wndClass.hInstance = hInstance;
   wndClass.lpszClassName = "BEEPBUTTON";
   OldWndProc = wndClass.lpfnWndProc;
   wndClass.lpfnWndProc = WndProc;
   RegisterClass(&wndClass);
   hwnd = CreateWindow("BEEPBUTTON", "Hello, World!",
                    WS_VISIBLE | BS_CENTER, 100, 100, 100, 80,
                    NULL, NULL, hInstance, NULL);
   while (GetMessage(&msg, NULL, 0, 0))
   {
      if (msg.message == WM_LBUTTONUP)
      {
         DestroyWindow(hwnd);
         PostQuitMessage(0);
      }
      DispatchMessage(&msg);
   }
   return msg.wParam;
}
```

We have looked at the difference between the two techniques, subclassing and superclassing, in terms of their implementation. But what is the difference between them in terms of their utility? In other words, when would you use subclassing, and when would you use superclassing?

The difference is simple. Subclassing modifies the behavior of an existing class; superclassing creates a new class based on the behavior of an existing class. In other words, if you use subclassing, you implicitly alter the behavior of every feature in your application that relies on the class that you subclass. In contrast, superclassing only affects windows that are based explicitly on the new class; windows based on the original class are not be affected.

7.6 DIALOG BOXES

In addition to its main application window with its title and menu bar and application-defined contents, an application most commonly uses dialogs to exchange information with the user. Typically, the application's main window exists throughout the life of the application, while its dialogs are more transient in nature, popping up only for the duration of a brief exchange of data; but this is not the key distinguishing characteristics of a main window and a dialog. Indeed, there are applications that use a dialog box as their main window; in other applications, a dialog may remain visible for most of the application's lifetime.

A dialog box usually contains a set of dialog controls, themselves child windows, through which the user and the application exchange data. There are several Win32 functions that assist in constructing, displaying, and managing the contents of a dialog box. Applications developers usually need not be concerned about painting a dialog's controls or handling user-interface events; instead, they can focus on the actual exchange of data between the dialog's controls and the application.

Dialogs represent a versatile capability in Windows. To facilitate their efficient use, Windows provides two types of dialog boxes: modeless and modal.

7.6.1 Modal Dialogs

When an application displays a modal dialog box, the window that owns the dialog box is disabled, effectively suspending the application. The user must complete interaction with the modal dialog before the application can continue.

A modal dialog is usually created and activated through the DialogBox function. This function creates the dialog window from a dialog template resource and displays the dialog as a modal dialog. The application that calls the DialogBox function supplies the address of a callback function; DialogBox does not return until the dialog box is dismissed through a call to EndDialog made from this callback function (possibly in response to a user-interface event, such as a click on the OK button).

Although it is possible to create a modal dialog with no owner, it is not usually recommended. If such a dialog box is used, several issues must be taken into account. As the application's main window is not disabled, steps must be taken to ensure that messages sent or posted to it continue to be processed. Windows does not destroy or hide an ownerless dialog when other windows of the application are destroyed.

7.6.2 Modeless Dialogs

In contrast to modal dialogs, presenting a modeless dialog does not suspend execution of the application by disabling the owner window of the dialog box. However, modeless dialogs remain on top of their owner window even when the owner window gains focus. Modeless dialogs represent an effective way of continuously displaying relevant information to the user.

A modeless dialog is typically created through the CreateDialog function. As there is no equivalent of the DialogBox function for modeless dialogs, applications are responsible for retrieving and dispatching messages for the modeless dialog. Most applications do this in their main message loop; however, to ensure that the dialog responds to keyboard events as expected and enables the user to move between controls using keyboard shortcuts, the application must call the IsDialogMessage function.

A modeless dialog does not return a value to its owner. However, the modeless dialog and its owner can communicate using SendMessage calls.

The dialog box procedure for a modeless dialog must not call the EndDialog function. The dialog is normally destroyed by a call to DestroyWindow. This function can be called in response to a user-interface event from the dialog box procedure.

Applications are responsible for destroying all modeless dialog boxes before terminating.

7.6.3 Message Boxes

Message boxes are special dialogs that display a user-defined message, a title, and a combination of predefined buttons and icons. Their intended use is to display brief informational messages to the user and present the user with a limited set of choices. For example, message boxes can be used to notify the user of an error condition and request instructions whether to retry or cancel the operation.

A message box is created and displayed through the MessageBox function. The application that calls this function specifies the text string that is to be displayed and a set of flags indicating the type and appearance of the message box.

In addition to the default *application modal* behavior of a message box, application can specify two other modes of behavior: *task modal* and *system modal.* Use a task modal message box if you wish to disable interaction with all top-level windows of the application, not just the owner window of the message box. A system modal message box should be used in extreme cases, warning the user of a potential disaster that requires immediate attention. System modal message boxes disable interaction with all other applications until the user deals with the message box.

Note: System modal message boxes should be used very carefully. Few things are more annoying than a misbehaving application that displays a system modal message box repeatedly in a loop (perhaps due to a programming error), effectively rendering the entire system useless.

7.6.4 Dialog Templates

Although it is possible to create a dialog in memory, most applications rely on a *dialog template resource* to determine the type and appearance of controls within a dialog.

Dialog templates are typically created as part of the application's resource file. They can be created manually as a set of instructions in the resource file, or they can be created through a visual resource file editor, such as the resource editor of the Developer Studio.

The dialog template defines the style, position, and size of the dialog and lists all controls within it. The style, position, and appearance of controls are also defined as part of the dialog template. The various dialog box functions draw the entire dialog based on the dialog template, except for controls that are marked as owner-drawn.

7.6.5 The Dialog Box Procedure

Dialog box procedure is just another name for the window procedure of a dialog box. There is no fundamental difference between a dialog box procedure and a window procedure, except perhaps the fact that a dialog procedure relies on DefDlgProc, rather than DefWindowProc, for default processing of messages.

A typical dialog box procedure responds to WM_INITDIALOG and WM_COMMAND messages but little else. In response to WM_INITDIALOG, the dialog box procedure initializes the controls in the dialog. Windows does not send a WM_CREATE message to a dialog box procedure; instead, the WM_INITDIALOG message is sent, but only after all the controls within the dialog have been created, just before the dialog is displayed. This enables the dialog box procedure to properly initialize controls before they are seen by the user.

Most controls send WM_COMMAND messages to their owner window (that is, the dialog box itself). To carry out the function represented by a control, the dialog box procedure responds to WM_COMMAND messages by identifying the control and performing the appropriate action.

7.7 COMMON DIALOGS

Win32 implements a series of commonly used dialogs, freeing the programmer from the need to implement these for every application. These *common dialogs* are well known to every Windows user. They include dialogs for opening and saving files, selecting a color or a font, printing and setting up the printer, selecting a page size, and searching and replacing text.

Common dialogs can be used in two ways. Applications can utilize the common dialog "as is" by calling one of the common dialog functions that are part of the

Win32 API. Alternatively, applications can customize common dialogs by implementing a special hook function and supplying a custom dialog template.

Windows 95 has introduced several changes to the common dialogs that were known to Windows 3.1 and Windows NT programmers. However, most of these changes are cosmetic, and do not affect typical usage of the dialogs. Where the differences are significant.

Note: The appearance of all common dialog boxes has changed substantially in Windows 95. Applications that supply their own dialog templates must take this fact into account in order to present a visual appearance that is consistent with the rest of the operating system.

When a common dialog function encounters an error, the CommDlgExtendedError function can often be used to obtain additional information about the cause and nature of the problem.

7.7.1 The Open and Save As Dialogs

The File Open and File Save As dialogs are perhaps the most often seen common dialogs. The purpose of these dialogs is to enable the user to browse the file system and select a file to be opened for reading or writing.

The File Open dialog is displayed when the application calls the GetOpenFileName function. The function's single parameter is a pointer to an OPENFILENAME structure. Members of this structure provide initialization values for the dialog box, and, optionally, the address of a hook function and the name of a custom dialog template, which are used for customizing the dialog. When the dialog is dismissed, applications can obtain the user's selection from this structure. A typical File Open dialog is shown in Figure 7.2.

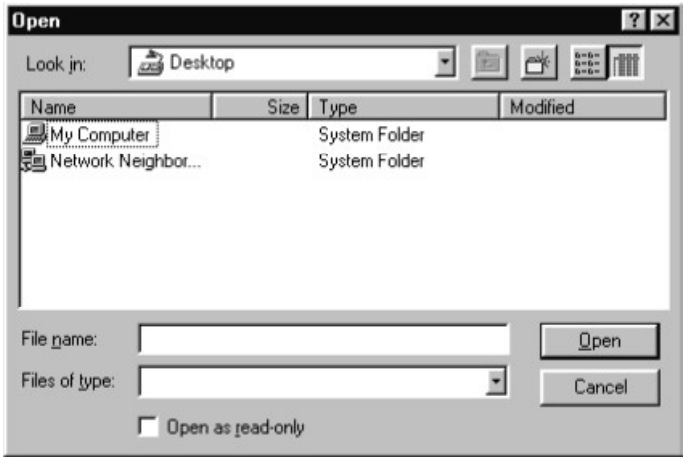


Figure 7.2: The File Open dialog (Explorer-style)

The File Save As dialog is displayed in response to a call to GetSaveFileName. This function also takes a pointer to an OPENFILENAME structure as its single parameter. An example for the File Save As dialog is shown in Figure 7.3.

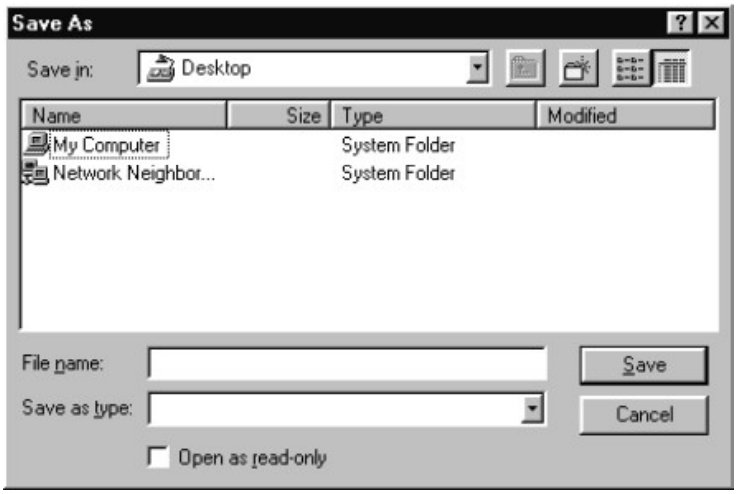


Figure 7.3: The File Save As dialog (Explorer-style)

For those familiar with the Windows 3.1 look of the common file dialogs, the difference between that and the new Windows 95 look is striking. Applications that wish to use the new look (and take advantage of the new Explorer-related functionality) must specify the style OFN_EXPLORER in the Flags member of the OPENFILENAME structure.

The Windows 95 versions of the common file dialogs have another new feature. When a file dialog is customized, it is no longer necessary to reproduce the entire dialog template before adding your modifications. Instead, it is possible to create a dialog template containing *only* the controls you wish to add to the dialog and an optional special field, labeled with the ID stc32, indicating where the standard components of the dialog should be placed.

7.7.2 The Choose Color Dialog

The Choose Color dialog box is used when the user is requested to select a color. The dialog can be used to select a color from the system palette, or to specify a custom color.

The Choose Color dialog, shown in Figure 7.4, is presented in response to a call to the Choose Color function. Applications can control the initialization values of this dialog through the pointer to a CHOOSECOLOR structure, passed to the ChooseColor function as its single parameter. Through this structure, applications can also customize

the dialog's behavior by supplying a hook function and the name of a custom dialog template. When the dialog is dismissed, the new color selection is available as the rgbResult member of the CHOOSE COLOR structure.

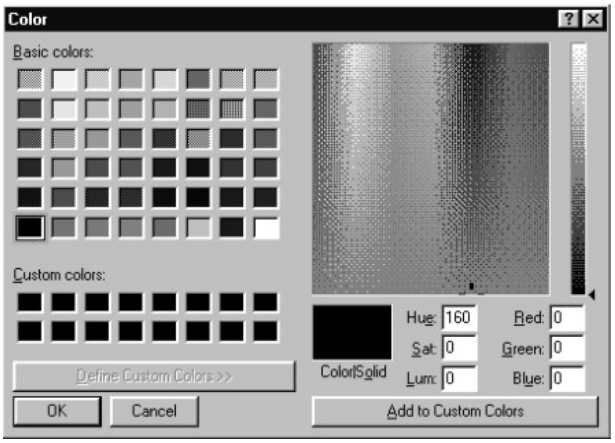


Figure 7.4: The Choose Color dialog

7.7.3 The Font Selection Dialog

Another of the more frequently seen common dialogs is the font selection dialog. Through this dialog, the user can select a typeface, a font style, font size, special effects, text color, and, in the case of Windows 95, a script. The font selection dialog is shown in Figure 7.5.

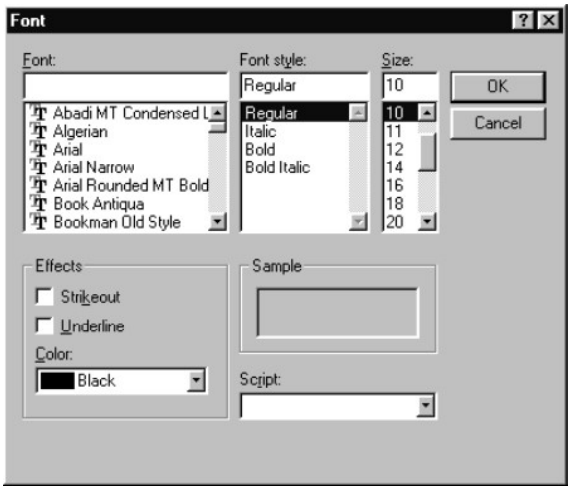


Figure 7.5: The Font Selection dialog

The font selection dialog is initialized through the CHOOSEFONT structure. This structure can also be used to specify a custom hook function and the name of a custom dialog template. The lpLogFont member of this structure points to a LOGFONT structure that can be used to initialize the dialog and receives information about the newly selected font when the dialog is dismissed. This structure can be used in a call to the GDI function CreateFontIndirect to actually create the font for use.

7.7.4 Dialogs for Printing and Print Setup

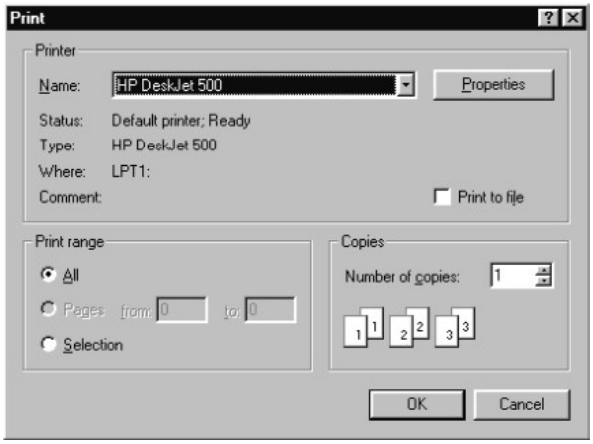


Figure 7.6: The Print dialog

To use the Print dialog, applications must first prepare the contents of a PRINTDLG structure, then call the PrintDlg function with a pointer to this structure as the function's only parameter.

The Page Setup dialog is displayed when applications call the PageSetupDlg function. The function's only parameter is a pointer to a PAGESETUPDLG structure. Through this structure, applications can control the fields of the dialog and possibly specify customization. When the dialog is dismissed, the user's selections are available in this structure.

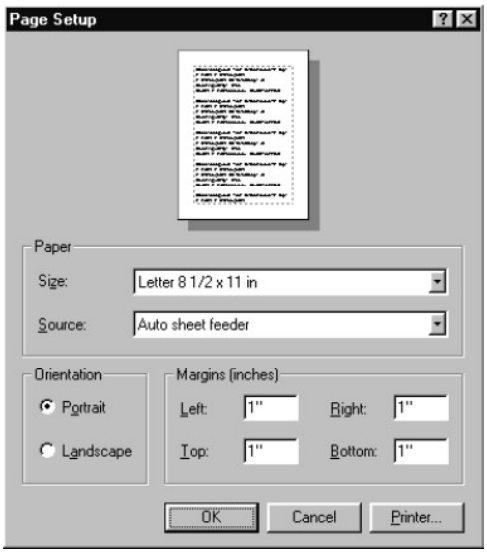


Figure 7.7: The Page Setup dialog

7.7.5 Text Find and Replace Dialogs

The Find and Find and Replace dialogs present an interface where the user can enter a text search string and, optionally, a replacement string. These dialogs differ fundamentally from the other common dialogs in that they are modeless dialogs; the other common dialogs all operate as modal dialogs. Therefore, the application that creates them is responsible for providing the message loop and dispatching dialog messages through the IsDialogMessage function.

The Find dialog, shown in Figure 7.8, is displayed in response to a call to the Find Text function. The function returns a dialog handle that can be used in the application's message loop in a call to IsDialogMessage. The dialog is initialized through a FINDREPLACE structure, which also receives any values the user may enter in the dialog.

The dialog communicates with its owner window through a series of messages. Before calling FindText, applications should register the message string "FINDMSGSTRING" through a call to the RegisterWindowMessage function. The Find dialog will send this message to the application whenever the user enters a new search value.

Figure 7.8: The Find Text dialog

The Replace dialog (Figure 7.9) is a close cousin to the Find dialog and is initialized through an identical FINDREPLACE structure. This dialog is displayed in response to a call to the ReplaceText function.

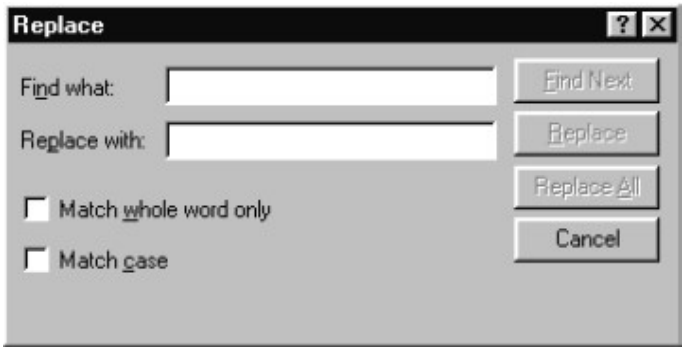


Figure 7.9: The Replace dialog

When the application receives a message from a Find or Replace dialog, it can check the Flags member of the FINDREPLACE structure to determine what action was requested by the user.

Note: The Find and Replace dialogs are not destroyed when the FindText or ReplaceText functions return. For this reason, an application would normally allocate the FINDREPLACE structure in global memory. If memory allocated for the FINDREPLACE structure is deallocated before the Find or Replace dialogs are destroyed, the application will fail.

7.7.6 Common Dialogs Example

The example program shown in Listing 7.6 creates and displays each of the common dialogs in sequence. This example has little practical value; it simply demonstrates, with a minimum amount of code, how these dialogs can be created and displayed. This sample can be compiled from the command line with cl commdlgs.c comdlg32.lib user32.lib.

Listing 7.6 Common Dialogs

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
                         WPARAM wParam, LPARAM lParam)
{
   switch(uMsg)
   {
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
      default:
         return DefWindowProc(hwnd, uMsg, wParam, lParam);
   }
   return 0;
}
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                   LPSTR d3, int nCmdShow)
{
```

```
MSG msg;
HWND hwnd;
WNDCLASS wndClass;
 OPENFILENAME ofn;
CHOOSECOLOR cc;
 CHOOSEFONT cf;
 PRINTDLG pd;
 PAGESETUPDLG psd;
 FINDREPLACE fr;
COLORREF crCustColors[16];
 LOGFONT lf;
char szFindWhat[80];
char szReplaceWith[80];
HWND hdlgFt, hdlgFr;
if (hPrevInstance == NULL)
{
    memset(&wndClass, 0, sizeof(wndClass));
    wndClass.style = CS_HREDRAW | CS_VREDRAW;
    wndClass.lpfnWndProc = WndProc;
   wndClass.hInstance = hInstance;
    wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndClass.hbrBackground = (HBRUSH)(COLOR_APPWORKSPACE + 1);
  wndClass.lpszClassName = "COMMDLGS";
    if (!RegisterClass(&wndClass)) return FALSE;
}
hwnd = CreateWindow("COMMDLGS", "Common Dialogs Demonstration",
                    WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
                     NULL, NULL, hInstance, NULL);
```

```
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
memset(&ofn, 0, sizeof(ofn));
ofn.lStructSize = sizeof(OPENFILENAME);
GetOpenFileName(&ofn);
memset(&ofn, 0, sizeof(ofn));
ofn.lStructSize = sizeof(OPENFILENAME);
GetSaveFileName(&ofn);
memset(&cc, 0, sizeof(cc));
memset(crCustColors, 0, sizeof(crCustColors));
cc.lStructSize = sizeof(cc);
cc.lpCustColors = crCustColors;
ChooseColor(&cc);
memset(&cf, 0, sizeof(cf));
memset(&lf, 0, sizeof(lf));
cf.lStructSize = sizeof(cf);
cf.lpLogFont = &lf;
cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
ChooseFont(&cf);
memset(&pd, 0, sizeof(pd));
pd.lStructSize = sizeof(pd);
PrintDlg(&pd);
memset(&psd, 0, sizeof(psd));
psd.lStructSize = sizeof(psd);
PageSetupDlg(&psd);
memset(&fr, 0, sizeof(fr));
memset(szFindWhat, 0, sizeof(szFindWhat));
memset(szReplaceWith, 0, sizeof(szReplaceWith));
fr.lStructSize = sizeof(fr);
```

```
    fr.hwndOwner = hwnd;
    fr.lpstrFindWhat = szFindWhat;
    fr.lpstrReplaceWith = szReplaceWith;
    fr.wFindWhatLen = sizeof(szFindWhat);
    fr.wReplaceWithLen = sizeof(szReplaceWith);
    hdlgFt = FindText(&fr);
    hdlgFr = ReplaceText(&fr);
    while (GetMessage(&msg, NULL, 0, 0))
        if(!IsDialogMessage(hdlgFt, &msg))
            if(!IsDialogMessage(hdlgFr, &msg))
                DispatchMessage(&msg);
    return msg.wParam;
}
```

7.7.7 OLE Common Dialogs

As part of the OLE 2 implementation, the system provides common dialogs for the following set of functions: Insert Object, Paste Special, Change Source, Edit Links, Update Links, Object Properties, Convert, and Change Icon. Most applications do not invoke these dialogs directly, but use the Microsoft Foundation Classes (and, in particular, the wrapper classes for these dialogs) to implement OLE functionality.

7.8 CONTROLS

A control is a special window that typically enables the user to perform a simple function and sends messages to this effect to its owner window. For example, a pushbutton control has one simple function, namely that the user can click on it; when that happens, the pushbutton sends a WM_COMMAND message to the window (typically a dialog) that owns it.

Windows offers several built-in control classes for the most commonly used controls. A dialog with a sample collection of these controls is shown in Figure 7.10.

Windows 95 introduced a set of new control classes, collectively referred to as Windows 95 Common Controls. This name is slightly misleading as the new control classes are now also available in Windows NT 3.51 and Win32s 1.3.

Applications can also create their own controls. These can be derived from the standard control classes, or they can be built independently.

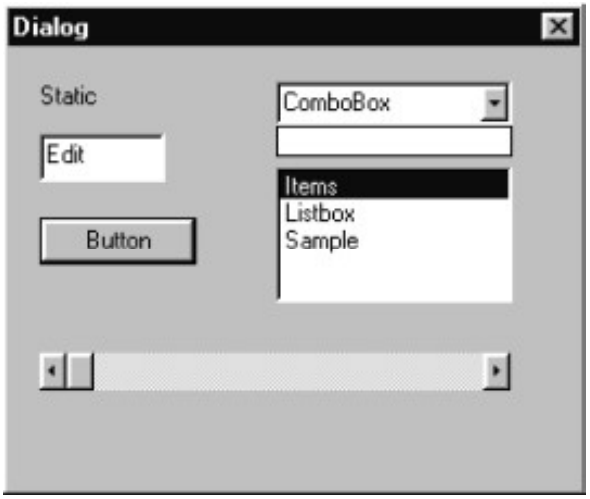


Figure 7.10: A collection of standard controls

The control class and the control style (which defines variations of behavior within a button class) are usually both defined in an application's resource file. Alternatively, applications that create controls programmatically select the button class and specify the button style as parameters to the CreateWindow function.

7.8.1 Static Controls

Static controls are perhaps the simplest of all control types. The sole purpose of their existence is to display a piece of text, such as a label for another control. Static controls do not respond to user-interface events and do not send messages to their owner window.

7.8.2 Buttons

Buttons, as their name implies, are controls that respond to simple mouse clicks. There are several button types. A *pushbutton* is a button that posts a WM_COMMAND message to its owner window when it is clicked. A *check box* indicates one of two states, selected and not selected. A variant of the check box, the *three-state check box*, adds a third, disabled state to the other two. A *radio button* is a control that is typically used in groups, indicating a set of mutually exclusive choices.

There are variants to these control styles that define secondary aspects of their behavior.

7.8.3 Edit Controls

An edit control is a rectangular area where the user can enter unformatted text. The text can be a few characters—such as the name of a file—or an entire text file; for

example, the client area of the Windows Notepad application is one large edit control. Applications typically communicate with the edit control through a series of messages that are used to set or retrieve text from the control.

7.8.4 List Boxes

A list box contains a collection of values arranged in rows. Users can use the mouse cursor to select the desired value from the list. If the list box contains more values than can be displayed at ones, a vertical scrollbar is also displayed as part of the list box.

7.8.5 Combo Boxes

A combo box combines the functionality of a list box and an edit control. Users can enter a value in the edit control part of the combo box. Alternatively, they can click the down arrow next to the edit control to display the list box part, where a value can be selected.

7.8.6 Scrollbars

A scrollbar control consists of a rectangular area with two arrows at the end and a sliding pointer. A scrollbar can be vertical or horizontal. Scrollbars are typically used to indicate the position of the visible portion within a larger area. Applications also used scrollbars to implement the functionality of a slide control; however, as one of the new Windows 95 common controls is a slider control, using scrollbars for this purpose is no longer necessary.

7.8.7 Tab Controls

Tab controls help in implementing multipage dialogs, also known as tabbed dialogs or property sheets. A tab control provides a user-interface where the user can select the dialog page (property page) by clicking on a little tab. The tab gives the visual appearance of several sheets organized on top of each other and clicking on the tab gives the visual impression of bringing the selected sheet to front.

7.8.8 Tree Controls

Tree controls present a list of items in a hierarchical organization. Tree controls are ideal for displaying hierarchical lists, such as a list of directories on disk. Tree controls provide an efficient mechanism for displaying a large number of items by providing the ability to expand and collapse higher-level items.

7.8.9 List Controls

List controls expand the functionality of a list box by providing a means to display a list of items in one of several formats. In a typical list control, items have an icon and some text; the control can display these items in a variety of formats as icons, or as list items arranged in rows.

7.8.10 Slider Control

A slider control provides the functionality similar to the sliding volume control on many stereo systems. The user can position the sliding tab with the mouse to set a specific position in the slider control. Slider controls are ideal in multimedia applications as volume or picture controls, or controls through which the user can set the position during playback of a multimedia data source.

7.8.11 Progress Bars

Progress bars are used to indicate the progress of a lengthy process. Progress bars do not accept user input; they are used for informational purposes only.

7.8.12 Spin Buttons

Spin buttons are used to increment or decrement the value of an associated control, usually an edit control.

7.8.13 Rich-text Edit Control

The rich-text edit control expands the functionality of the Windows 3.1 edit control by enabling the editing of Microsoft RTF (Rich Text Format) files. Rich-text controls encapsulate the capability of a reasonably sophisticated word processor.

7.8.14 Hot Key Control

A hot key control accepts a keystroke combination from the user, which the application can use to set up a hot key through the WM_SETHOTKEY message.

Other Windows common controls include the animation control, header control, status bar, toolbar control, and tooltip control.

Figure 7.11 presents a collection of Windows 95 common controls in a dialog.

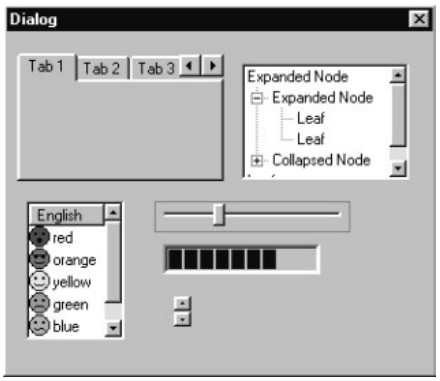


Figure 7.11: Some Windows 95 common controls

Summary

A window is a rectangular area on the screen through which applications and the user communicate. Applications draw into the window to display information for the user. Applications receive messages on user-interface events through a handle to the window.

Windows are arranged hierarchically. At top is the desktop window. Top-level windows are those whose parent is the desktop window—or those that have no parent window. Child windows are those whose parent is a top-level window or another child window. Windows sharing the same parent are siblings; the order in which sibling windows are displayed is called the Z-order. A special category of windows contains top-level windows that have the topmost attribute; these windows always precede non-topmost windows in the Z-order, even when a non-topmost window is the active window.

A top-level window may have an owner window that is different from its parent window.

Typical windows that users normally interact with include overlapped windows (normal application windows); popup windows (dialog boxes); and controls.

Window messages are handled in the window procedure. A window procedure and other window attributes are associated with the window class from which windows are derived. In addition to the capability of defining their own window classes, applications can also superclass and subclass existing window classes. Subclassing means modifying the behavior of an existing window class; superclassing means creating a new window class based on the behavior of an existing class.

Part of the Win32 API is a set of functions that assist in creating, displaying, and managing dialogs. Windows distinguishes between modal dialogs and modeless dialogs. A modal dialog disables its owner window while it is displayed and does not return control to the application until the user dismisses the dialog. In contrast, modeless dialogs are displayed without disabling their owner window. Applications must provide message loop functionality and dispatch dialog messages through the IsDialogMessage function for modeless dialogs.

Windows also provides a set of common dialogs for common tasks. These include dialogs for opening and saving a file, printer and page setup, color and font selection, and text find and replace functions. In addition, a set of common dialogs is available to implement OLE-related functionality.

Controls include buttons, static text, edit boxes, list boxes, combo boxes, and scrollbars. Applications can also implement new control types. In addition, Windows 95 defines a set of new common controls: list views, tree views, tab controls, hot key controls, sliders, progress bars, spin buttons, and rich-text edit controls.

Controls are usually defined through dialog box templates in the application's resource file. Controls communicate with the application by sending messages (such as WM_COMMAND messages) to their owner window, that is, the dialog box.

REVIEW EXERCISE

1. Define window in VC++ environment?
2. A window is identified by a *window handle*. Comment.
3. What are the steps in® creating windows?
4. How Painting in a window is performed?
5. What is the purpose of a window class?
6. An application should not attempt to subclass or superclass a window that belongs to another process. Comment.
7. What is Subclassing?

CHAPTER

8 DIALOGS AND PROPERTY SHEETS

Applications use dialogs in many situations. The MFC Library supports dialogs through the CDialog class and derived classes.

A CDialog object corresponds to a dialog window, the content of which is based on a dialog template resource. The dialog template resource can be edited using any dialog editor; typically, however, you would use the dialog editor that is part of the Developer Studio for this purpose.

Perhaps the most important feature of CDialog is its support for Dialog Data Exchange, a mechanism that facilitates the easy and efficient exchange of data between controls in a dialog and member variables in the dialog class.

The CDialog class is derived from CWnd; thus, you can use many CWnd member functions to enhance your dialog. Furthermore, your dialog classes can have message maps; indeed, except for the most simple dialogs, it is often necessary to add message map entries to handle specific control messages.

Newer applications often support tabbed dialogs, or *property sheets*. A property sheet is really several dialogs merged into one; the user uses tab controls to pick any one of the *property pages* that comprise a property sheet.

8.1 CONSTRUCTING DIALOGS

The basic steps in constructing a dialog and making it part of your application include creating the dialog template resource, creating a CDialog-derived class that corresponds to this resource, and constructing an object of this class at the appropriate location in your application.

For our experiments with dialogs, we use a simple AppWizard-created SDI application named DLG. Other than selecting the Single document application type, this application should be created with AppWizard's defaults.

The next section shows you how to create a simple dialog that has an editable text field and make it part of the DLG application by connecting it to a new menu item, View Dialog. The dialog, as displayed by DLG, is shown in Figure 8.1.

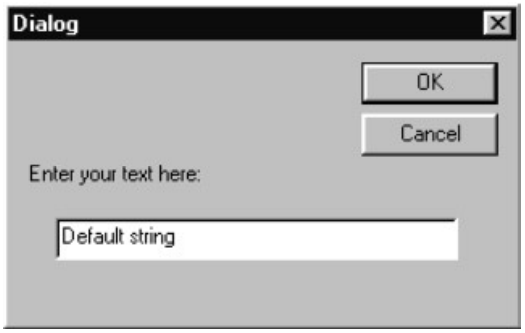


Figure 8.1: A simple dialog

8.1.1 Adding a Dialog Template

The first step in constructing a dialog is to create the dialog template resource. This resource can be built using the integrated dialog editor that is part of the Developer Studio. Figure 8.2 shows the dialog under construction. The OK and Cancel buttons are supplied by the dialog editor when a blank dialog is created; to that, we should add a static control and an edit control. The edit control will have the identifier IDC_EDIT; the dialog itself will be identified as IDD_DIALOG.

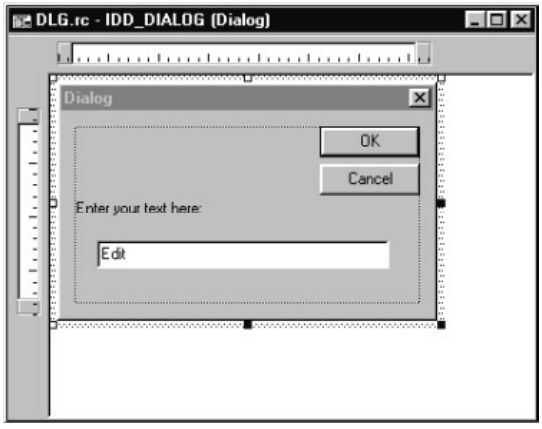


Figure 8.2: Constructing a simple dialog

While the dialog template is open for editing in the dialog editor, you can directly invoke the ClassWizard to construct the dialog class corresponding to this template.

8.1.2 Constructing the Dialog Class

Although it is possible to create a dialog class by hand, in many cases it is easier to rely on the ClassWizard for this purpose. To create a dialog class corresponding to the dialog shown in Figure 8.2, use the right mouse button anywhere in the dialog editor window to bring up a popup menu; from this popup menu, select the ClassWizard command.

The ClassWizard, after detecting that it has been invoked for a newly constructed resource, presents the Adding a Class dialog that is shown in Figure 8.3. Select the Create a new class radio button and click OK.

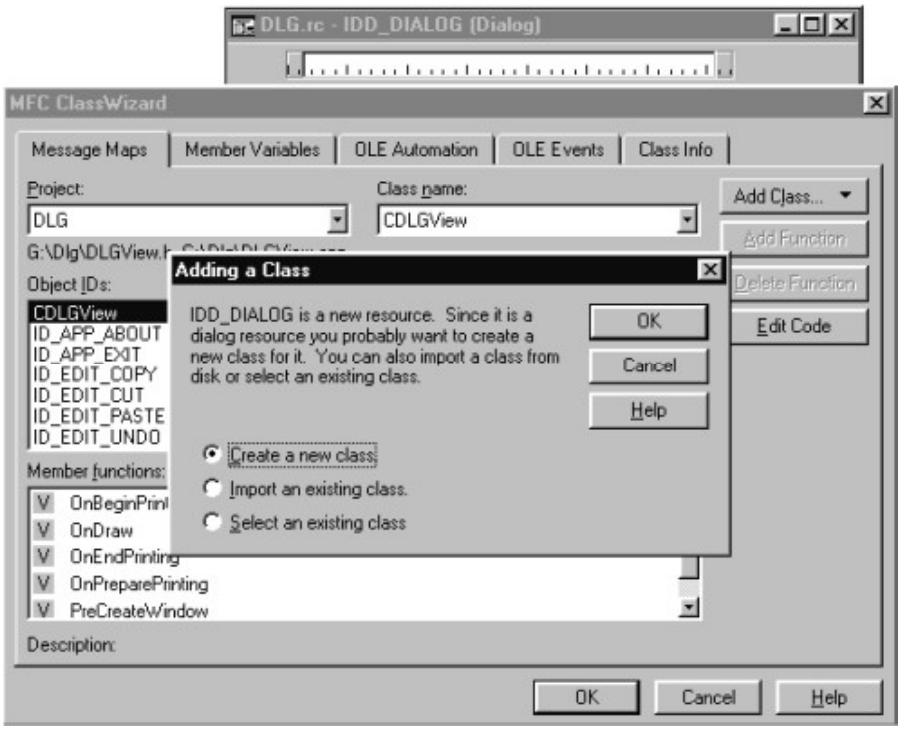


Figure 8.3: The adding a class dialog

At this time, the ClassWizard displays the Create New Class dialog. Here, you can enter the dialog's name and set other options, such as the filename, the resource identifier, or OLE automation settings. You can also add this class to the Component Gallery for later reuse in other applications.

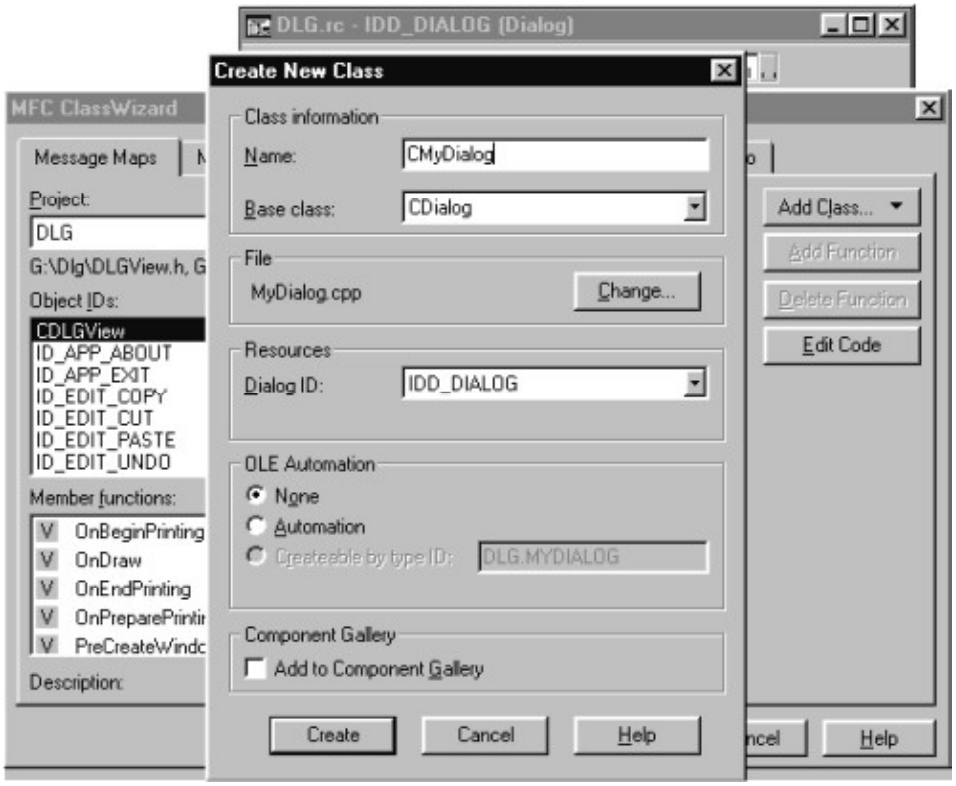


Figure 8.4: The create new class dialog

Add a suitable name for the new class, for example, CMyDialog. It may also be a good idea to uncheck the Add to Component Gallery check box; after all, it is not necessary to clutter the component gallery with code that is used for demonstration purposes only.

Should you change the filenames that the ClassWizard suggests for your new class's header and implementation files? Should you use a separate header and implementation file for every new dialog you create? This is an interesting question. At the surface, the answer would appear to be a yes; then again, even the AppWizard itself violates this "rule" when it places both the declaration and implementation of your application's about dialog into the application object's implementation file. Thus, I believe that in the end, it is best left to the judgment of the programmer. I often grouped dialog classes together if they were small, simple, and related. Leaving them in separate files tended to clutter the application workspace. However, this is less of a concern with Visual C++ where you no longer have to use File View to access your source code; also, using separate files makes it easier to use the Component Gallery.

For now, leave the filenames at the ClassWizard-generated defaults: MyDialog.h and MyDialog.cpp. Clicking on the Create button actually creates the new class and leaves the ClassWizard main dialog open.

The next step is to add a member variable that corresponds to the edit field in the dialog template.

8.1.3 Adding Member Variables

To add a new member variable, select the Member Variables tab in ClassWizard (Figure 8.5).

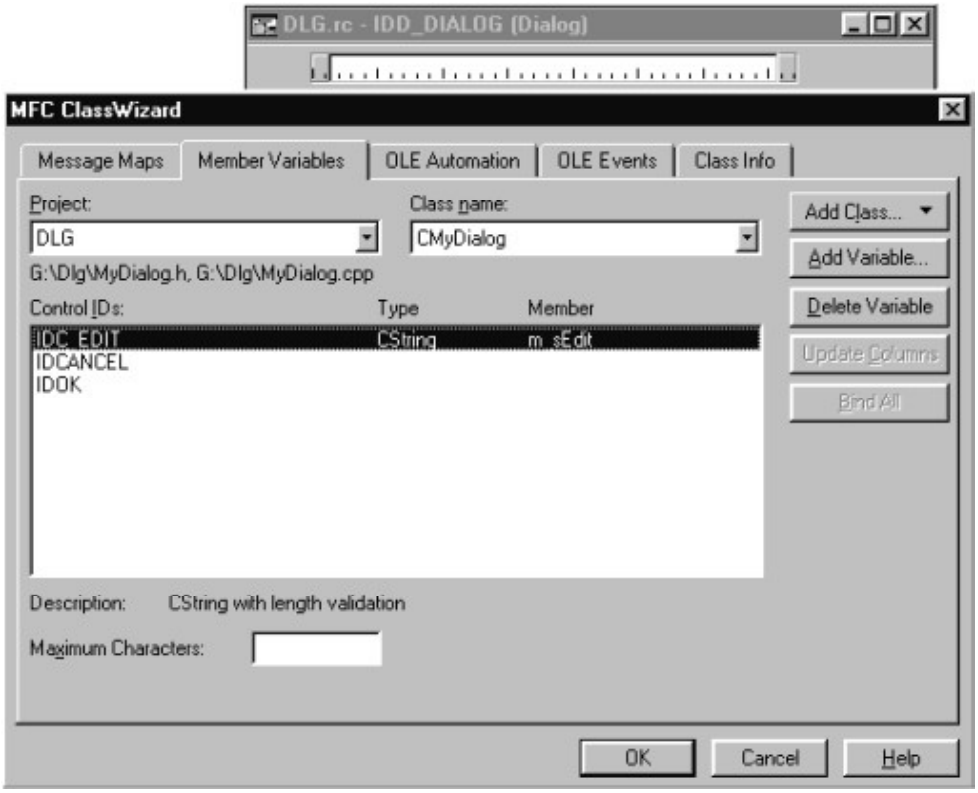


Figure 8.5. Member variables in ClassWizard.

The member variable for the IDC_EDIT control can be added by double-clicking this identifier in the Control IDs column. This invokes yet another dialog, shown in Figure 8.6. Type in the new variable's name (m_sEdit) and click on the OK button. Once the member variable has been added, you can dismiss the ClassWizard altogether by clicking on the OK button in the ClassWizard dialog.

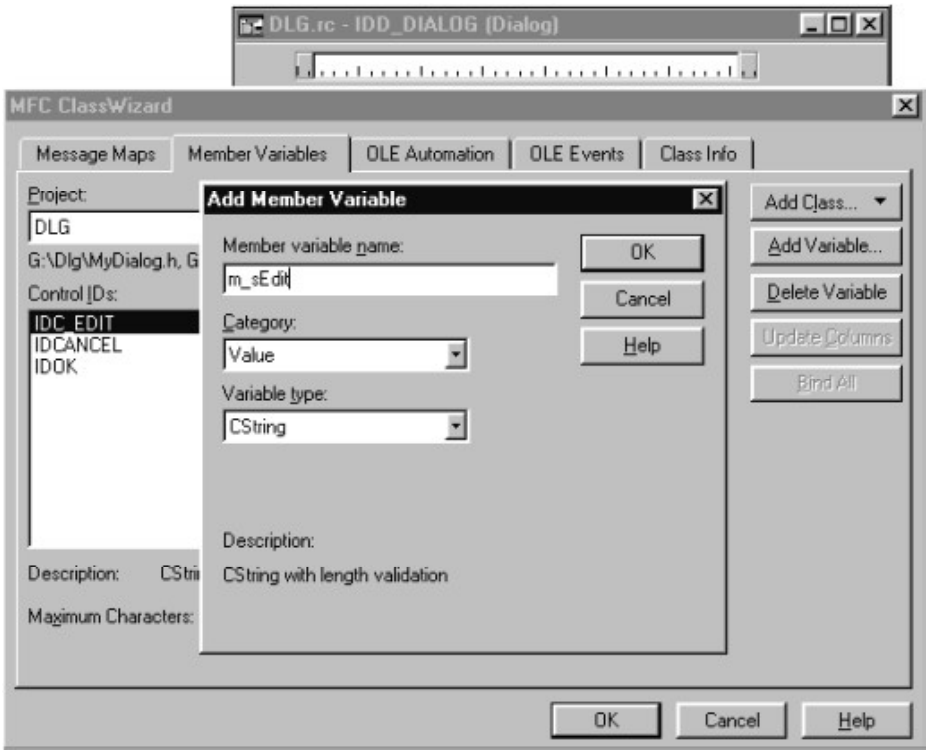


Figure 8.6: The add member variable dialog

If you still have the dialog template resource open for editing, dismiss that window as well. In a moment, we'll begin creating the code that will invoke our new dialog. Before we do that, however, take a look at the code that the ClassWizard has generated for us so far.

8.1.4 Class Wizard Results

The declaration of CMyDialog (in MyDialog.h) is shown in Listing shown below. Part of the class declaration is the declaration of IDD, which identifies the dialog template. The class declaration also contains the member variable m_sEdit, which we created through ClassWizard.

Listing 8.1 CMyDialog Class Declaration

```
class CMyDialog : public CDialog
{
// Construction
public:
```

```
    CMyDialog(CWnd* pParent = NULL);   // standard constructor
// Dialog Data
    //{{AFX_DATA(CMyDialog)
    enum { IDD = IDD_DIALOG };
    CString m_sEdit;
    //}}AFX_DATA
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMyDialog)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);
    //}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CMyDialog)
        // NOTE: the ClassWizard will add member functions here
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Declarations for the constructor function and an override for the DoDataExchange member function are also provided here.

These two functions are defined in MyDialog.cpp (Listing 8.2). Notice that the ClassWizard inserted code into both of them; the member variable m_sEdit is initialized in the constructor and also referred to in DoDataExchange.

Listing 8.2 CMyDialog Member Functions

```
CMyDialog::CMyDialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMyDialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(CMyDialog)
    m_sEdit = _T("");
    //}}AFX_DATA_INIT
}
void CMyDialog::DoDataExchange(CDataExchange* pDX)
```

```
{
   CDialog::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CMyDialog)
    DDX_Text(pDX, IDC_EDIT, m_sEdit);
   //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CMyDialog, CDialog)
   //{{AFX_MSG_MAP(CMyDialog)
      // NOTE: the ClassWizard will add message map macros here
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

DoDataExchange is the function that facilitates data exchange between member variables and dialog controls. It is invoked both when the dialog is constructed and when it is dismissed. The macros inserted by ClassWizard (such as the DDX_Text macro) facilitate data exchange in both directions; the direction is determined by the m_bSaveAndValidate member of the CDataExchange object, pointed to by the pDX parameter. We revisit this function and the various data exchange helper macros shortly.

8.1.5 Invoking the Dialog

Construction of our dialog object is now complete. How are we going to invoke this dialog from our DLG application?

First, we must make a "design decision," if it can be dignified with that phrase: The new dialog will be invoked when the user selects a new menu item, Dialog, from the View menu.

This menu item must first be added to the application's menu using the resource editor (see Figure 8.7).

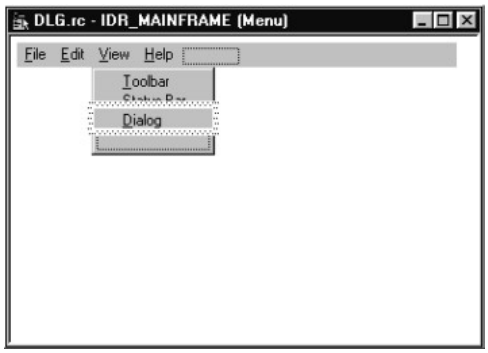


Figure 8.7: Adding the view dialog menu item

To add code that handles the new menu item, invoke the ClassWizard, and add a command handler function for ID_VIEW_DIALOG to the CMainFrame class. (Why CMainFrame? Displaying this dialog has nothing to do with a specific document or any of its views, so CMainFrame appears to be the most logical choice.) This is accomplished most easily by right-clicking on the new Dialog menu item to invoke the ClassWizard, selecting ClassWizard's Message tab, selecting the ID_VIEW_DIALOG identifier, and using the Add Function button.

The implementation of CMainFrame::OnViewDialog is shown in Listing 8.3. After constructing the dialog object, we assign an initial value to the member variable m_sEdit. Next, we invoke the dialog via the DoModal function. After the dialog is dismissed by the user, we examine the new value of m_sEdit by simply displaying it in a message box.

Listing 8.3 The CMainFrame::OnViewDialog Member Function.

```
void CMainFrame::OnViewDialog()
{
   // TODO: Add your command handler code here
   CMyDialog myDialog;
   myDialog.m_sEdit = "Default string";
   myDialog.DoModal();
   MessageBox(myDialog.m_sEdit);
}
```

Note that in order to be able to declare an object of type CMyDialog in CMainFrame::OnViewDialog, it is necessary to include the MyDialog.h header file in MainFrm.cpp.

That's it. The application is ready to be recompiled and run.

8.1.6 Modeless Dialogs

Invoking a dialog through the DoModal member function invokes the dialog as a modal dialog. However, sometimes applications require the use of modeless dialogs. The steps of creating and displaying a modeless dialog are different from the steps taken for modal dialogs.

To convert our dialog in DLG to a modeless dialog, we must first modify the dialog's constructor function. In the constructor, we must make a call to the Create member function in order to construct the dialog box object. We must also call a different version of the base class constructor, as shown in Listing 8.4.

Listing 8.4 Modeless Version of CMyDialog::CMyDialog

```
CMyDialog::CMyDialog(CWnd* pParent /*=NULL*/)
    : CDialog()
{
    Create(CMyDialog::IDD, pParent);
    //{{AFX_DATA_INIT(CMyDialog)
    m_sEdit = _T("");
    //}}AFX_DATA_INIT
}
```

Invocation of the dialog from CMainFrame::OnViewDialog is also different. Instead of calling the dialog's DoModal member function, we just construct the dialog object; the call to Create within the constructor takes care of the rest.

Note that we can no longer construct the dialog box on the stack. Because a modeless dialog box is long lived and continues to exist even after CMainFrame::OnViewDialog returns, we have to allocate the CDialog object differently. This new version of CMainFrame::OnViewDialog is shown in Listing 8.5 (MainFrm.cpp).

Listing 8.5 Constructing a modeless dialog in CMainFrame::OnViewDialog

```
void CMainFrame::OnViewDialog()
{
    // TODO: Add your command handler code here
    CMyDialog *pMyDialog;
    pMyDialog = new CMyDialog;
    pMyDialog->m_sEdit = "Default string";
    pMyDialog->UpdateData(FALSE);
    pMyDialog->ShowWindow(SW_SHOW);
}
```

Why was it necessary to call UpdateData in this function? Because we set the value of m_sEdit *after* the dialog box object has been constructed and initial Dialog Data Exchange took place. By calling UpdateData, we ensure that the controls in the dialog box object are updated to reflect the settings in the member variables of the CDialog object. This is yet another example that should remind us that the C++ object and the Windows object are two different entities.

We must also call the ShowWindow member function to make the new dialog visible. Alternatively, we could have created the dialog box template resource with the WS_VISIBLE style.

How long will this dialog exist? As long as the user does not dismiss it by clicking on the OK or Cancel button. At that time, the default implementations of CDialog::OnOK and CDialog::OnCancel hide the dialog box but do not destroy it. Obviously, we must override these functions to properly destroy the dialog. In both of these functions, a call must be made to the DestroyWindow member function.

We must also override the dialog's OnOK function to ensure that we process whatever the user entered in the dialog. We can no longer rely on the function calling DoModal for this purpose, for the simple reason that DoModal is never called.

Calling the DestroyWindow member function from OnOK and OnCancel ensures that the Windows dialog box object is destroyed; but how will the C++ object be destroyed? The answer to that question is yet another override. You must override the PostNcDestroy member function and delete the CDialog-derived object from within it.

To override the default implementations of OnOK, OnCancel, and PostNCDestroy, you must first add these functions to the CMyDialog class through ClassWizard. Perhaps the simplest way to do this is to open the implementation file, MyDialog.cpp, and use the WizardBar to add the functions.

Implementations of CMyDialog::OnOK, CMyDialog::OnCancel, and CMyDialog::PostNcDestroy are shown in Listing 8.6 (MyDialog.cpp).

Listing 8.6 Member functions in the modeless version of CMyDialog

```
void CMyDialog::OnCancel()
{
   // TODO: Add extra cleanup here
   CDialog::OnCancel();
   DestroyWindow();
}
void CMyDialog::OnOK()
{
   // TODO: Add extra validation here
   MessageBox(m_sEdit);
   CDialog::OnOK();
   DestroyWindow();
}
void CMyDialog::PostNcDestroy()
{
   // TODO: Add your specialized code here and/or call the base class
   CDialog::PostNcDestroy();
```

```
    delete this;
}
```

If your modeless dialog must notify the frame, document, or view, it can do so by calling a member function. The dialog class can have a member variable that stores a pointer to the frame, document, or view object from within which the dialog has been created. Other mechanisms for communication between the modeless dialog and other parts of your application are also conceivable; for example, the dialog may post a message to the application.

8.2 MORE ON DIALOG DATA EXCHANGE

In the preceding example, we have used Dialog Data Exchange to map the contents of an edit control to the contents of a CString member variable in the dialog class. The Dialog Data Exchange mechanism offers many other capabilities for mapping simple variables or control objects to controls in a dialog box.

Note: Although Dialog Data Exchange and Dialog Data Validation are described in the context of dialog boxes, they are not limited in use to dialog boxes only. The member functions discussed, such as DoDataExchange and UpdateData, are actually member functions of CWnd, not CDialog. Dialog Data Exchange is also used outside the context of a dialog box; CFormView and classes derived from it are one example.

8.2.1 Dialog Data Exchange

Dialog Data Exchange takes place in the dialog class's DoDataExchange member function. In this function, calls are made for all member variables that are mapped to controls. The calls that are made are to a family of MFC functions with names that begin with DDX_. These functions are responsible for performing the actual data exchange.

For example, to perform data exchange between an edit control and a member variable of type CString, the following call is made:

```
DDX_Text(pDX, IDC_EDIT, m_sEdit);
```

8.2.2 Dialog Data Validation

In addition to the simple exchange of data between member variables and dialog controls, MFC also offers a data validation mechanism. Data validation is accomplished through calls to functions with names that begin with DDV_. These functions perform the necessary validation and if a validation error is encountered, display a standard error message box and raise an exception of type CUserException. They also call the Fail member function of the CDataExchange object that is passed to DoDataExchange; this object, in turn, sets the focus to the offending control.

An example for a data validation function is DDV_MaxChars, which is used to validate the length of a string typed into an edit control. To validate that a string in an edit control is no longer than 100 characters, you would make the following call:

```
DDV_MaxChars(pDX, m_sEdit, 100);
```

Data validation calls for a given control must immediately follow the data exchange function call for the same control.

8.2.3 Using Simple Types

Dialog Data Exchange with simple types is supported for edit controls, scrollbars, check boxes, radio buttons, list boxes, and combo boxes.

Table 8.1 summarizes the various types supported by Dialog Data Exchange for edit controls.

Table 8.1. Dialog Data Exchange and validation for edit controls.

Control	Data Type	DDX function	DDV function
edit control	BYTE	DDX_Text	DDV_MinMaxByte
edit control	short	DDX_Text	DDV_MinMaxInt
edit control	int	DDX_Text	DDV_MinMaxInt
edit control	UINT	DDX_Text	DDV_MinMaxUnsigned
edit control	long	DDX_Text	DDV_MinMaxLong
edit control	DWORD	DDX_Text	DDV_MinMaxDWord
edit control	float	DDX_Text	DDV_MinMaxFloat
edit control	double	DDX_Text	DDV_MinMaxDouble
edit control	CString	DDX_Text	DDV_MaxChars
edit control	COleDateTime	DDX_Text	
edit control	COleCurrency	DDX_Text	
check box	BOOL	DDX_Check	
radio button	int	DDX_Radio	
list box	int	DDX_LBIndex	
list box	CString	DDX_LBString	
list box	Cstring	DDX_LBStringExact	
combo box	int	DDX_CBIndex	
combo box	CString	DDX_CBString	DDV_MaxChars
combo box	Cstring	DDX_CBStringExact	
scrollbar	int	DDX_Scroll	

The MFC Library provides additional versions of the DDX functions to facilitate data exchange between a dialog box and records in a database. These functions have names that begin with DDX_Field; for example, the database variant of DDX_Text would be named DDX_FieldText.

8.2.4 Using Control Data Types

In addition to assigning a member variable to a control representing the control's value, it is also possible to assign member variables that represent the control object itself. For example, it is possible to assign a variable of type CEdit to an edit control.

The Dialog Data Exchange mechanism uses the DDX_Control function to exchange data between a dialog control and a CWnd-derived control object.

A control object can be used concurrently with a member variable representing the control's value. For example, it is possible to assign both a CString object representing the control's value and a CEdit object representing the control itself to an edit control in a dialog.

Why would you use a control object? Through such an object, you can implement much greater control over the appearance and behavior of dialog controls. For example, as control objects are CWnd-derived, your application can use CWnd member functions to change the control's size and position. Through the control object, it is also possible to send messages to the control.

In the case of many control types (including the new common controls) you must use a control object for Dialog Data Exchange. The use of a simple data type is meaningless and not supported.

8.2.5 Implementing Custom Data Types

Versatile as the Dialog Data Exchange mechanism is, it would not be sufficient in many situations were it not for the capability to extend it for custom data types. Fortunately, the ClassWizard offers the capability to handle custom DDX and DDV routines.

The steps required to implement custom DDX/DDV support are time consuming and may only be beneficial for data types that you frequently reuse. That said, it is possible to add custom DDX/DDV support to a specific project, or to all projects, by modifying either your project's CLW file, or the ddx.clw file in your msdev\bin subdirectory.

8.3 DIALOGS AND MESSAGE HANDLING

CDialog-derived objects are, as you might expect from CWnd-derived objects, capable of handling messages. In fact, in all but the simplest cases, it is necessary to add message handlers to your CDialog-derived dialog class.

Message handling in a dialog is no different from message handling in a view or frame window. Message handler functions can be easily added to the dialog class's message map using ClassWizard. In the earlier examples we have already done that when we added override versions of the OnOK and OnCancel member functions. These member functions are handlers of WM_COMMAND messages. (The third override function we implemented, PostNcDestroy, is not a message handler; however, it is called from within the handler for WM_NCDESTROY messages, OnNcDestroy.)

The most frequently used message handlers in a dialog class correspond to messages sent to the dialog window by one of its controls. These include BN_CLICKED messages sent by a button; the variety of CBN_ messages sent by a combo box; EN_ messages sent by an edit control; and so on. Another set of message that dialog classes often handle consists of WM_DRAWITEM and WM_MEASUREITEM for owner-draw controls.

Owner-draw controls bring up an interesting issue. Should you handle such a situation from within your dialog class, or should you assign an object of a class derived from a control class to the control and handle it there? For example, if you have an owner-draw button, you have the choice of adding a handler for WM_DRAWITEM messages to your dialog class, or deriving a class from CButton and overriding its DrawItem member function.

8.4 PROPERTY SHEETS

Property sheets are several overlapping dialogs in one. The user selects one of the dialogs, or property pages, by clicking on the corresponding tab in a tab control.

MFC supports property sheets through two classes: CPropertySheet and CPropertyPage. CPropertySheet corresponds to the property sheet; CPropertyPage-derived classes correspond to the individual property pages within the property sheet.

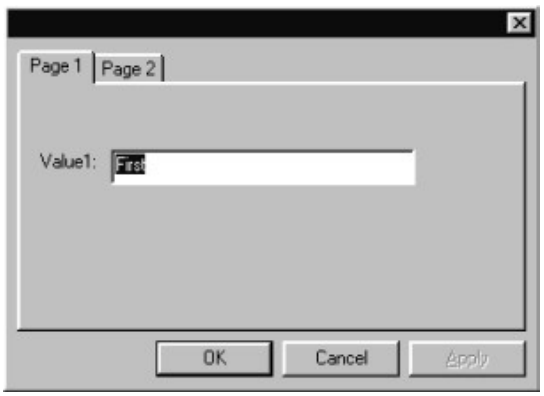


Figure 8.8: A sample property sheet

Using a property sheet requires several steps. First, the property pages must be constructed; next, the property sheet must be created.

The following simple example reviews this process. A new application, PRP, is used to display a property sheet, as shown in Figure 8.8. Like our earlier application, DLG, PRP is also a standard SDI application created by AppWizard.

8.4.1 Constructing Property Pages

Constructing a property page is very similar to constructing dialogs. The first step is to construct the dialog template resource for every property page that you wish to add to the property sheet.

There are a few special considerations when constructing a dialog template resource for a property page object:

1. The dialog's caption should be set to the text that you wish to see appear in the tab corresponding to the dialog.
2. The dialog's style should be set to child.
3. The dialog's border style should be set to thin.
4. The Titlebar style should be checked.
5. The Disabled style should be checked.

Although the property pages in a property sheet will overlap, it is not necessary to create them with the same size. The MFC Library will automatically adjust the size of property pages to match the size of the largest property page.

In this example we construct two property pages for our application—nothing fancy, just a simple text field in both of them. The first property page, titled "Page 1," is shown in Figure 8.9. To insert a blank property page template similar to the one shown here, use the IDR_PROPPAGE_SMALL subtype of the Dialog resource type in the Insert Resource dialog. Afterwards, you can add the controls as shown.

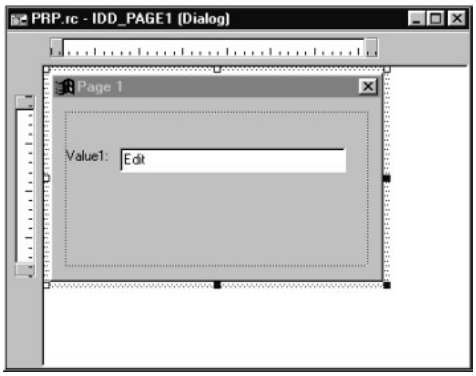


Figure 8.9: Constructing a property page

The identifier of the dialog template resource should be set to IDD_PAGE1; the identifier of the edit control should be set to IDC_EDIT1. Make sure that the dialog template's properties are set correctly. To set the dialog's caption, double click on the dialog to invoke the Dialog Properties property sheet (Figure 8.10).

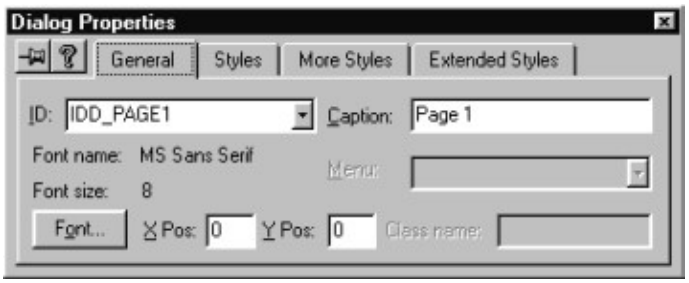


Figure 8.10: Property page dialog resource caption setting

To set the style, border style, and titlebar setting, select the Styles tab in the property sheet of the dialog resource (Figure 8.11).

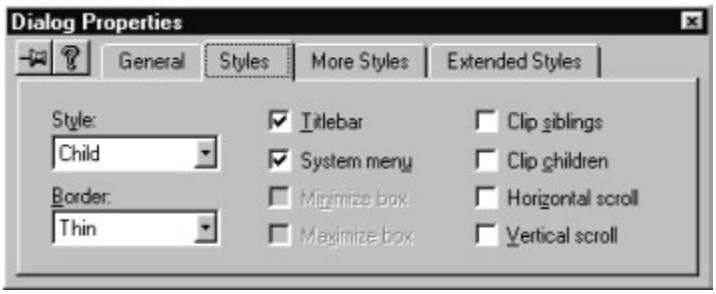


Figure 8.11: Property page dialog resource styles

To set the Disabled style of the dialog resource, use the More Styles tab in the dialog resource property sheet (Figure 8.12).

Figure 8.12: Setting the property page dialog resource to disabled

The second property page in our simple example is, for the sake of simplicity, nearly identical to the first. In fact, you can create the dialog resource for this second property page by simply making a copy of the first. Make sure that the identifier of the new dialog resource is set to IDD_PAGE2 and that the identifier of the edit control within it is IDC_EDIT2. (It would be perfectly legal to use the same identifier for controls in separate property pages; they act and behave like separate dialogs. Nevertheless, I prefer to use distinct identifiers; this helps reduce the possibility for errors.)

Once both property page dialog resources have been constructed, it is time to invoke the ClassWizard and construct classes that correspond to these property pages. To do so, invoke the ClassWizard while the focus is on the first property page dialog resource while it is open for editing. As with dialogs, the ClassWizard will recognize that the dialog template has no corresponding class and offer you the opportunity to create a new class.

In the Create New Class dialog, specify a name for the class corresponding to the dialog template (for example, CMyPage1). More importantly, make sure that this new class is based on CPropertyPage (and not the default CDialog). Once the correct settings have been entered, create the class.

While in ClassWizard, you should also add a member variable that corresponds to the edit control in the property page. Name this variable m_sEdit1.

These steps should be repeated for the second property page. The class for this property page should be named CMyPage2, and the member variable corresponding to its edit control should be named m_sEdit2.

Construction of our property pages is now complete. Take a brief look at the code generated by ClassWizard. The declaration of CMyPage1 is shown in Listing 8.7 (the declaration of CMyPage2 is virtually identical).

Listing 8.7 CMyPage1 declaration

```
class CMyPage1 : public CPropertyPage
{
    DECLARE_DYNCREATE(CMyPage1)
// Construction
public:
    CMyPage1();
    ~CMyPage1();
// Dialog Data
    //{{AFX_DATA(CMyPage1)
    enum { IDD = IDD_PAGE1 };
    CString m_sEdit1;
```

```
    //}}AFX_DATA
// Overrides
    // ClassWizard generate virtual function overrides
    //{{AFX_VIRTUAL(CMyPage1)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);
    //}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CMyPage1)
        // NOTE: the ClassWizard will add member functions here
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

As you can see, there is very little difference between this declaration and the ClassWizard-generated declaration of a CDialog-derived dialog class. Most importantly, CPropertyPage-derived classes can use Dialog Data Exchange functions just as classes derived from CDialog.

The implementation of CMyPage1 member functions (Listing 8.8) is also no different from the implementation of similar functions in a CDialog-derived class. Perhaps the one notable difference is that this class has been declared as dynamically creatable with the help of the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros.

Listing 8.8 CMyPage1 implementation

```
IMPLEMENT_DYNCREATE(CMyPage1, CPropertyPage)
CMyPage1::CMyPage1() : CPropertyPage(CMyPage1::IDD)
{
    //{{AFX_DATA_INIT(CMyPage1)
    m_sEdit1 = _T("");
    //}}AFX_DATA_INIT
}
CMyPage1::~CMyPage1()
{
}
```

```
void CMyPage1::DoDataExchange(CDataExchange* pDX)
{
   CPropertyPage::DoDataExchange(pDX);
   //{{AFX_DATA_MAP(CMyPage1)
    DDX_Text(pDX, IDC_EDIT1, m_sEdit1);
   //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CMyPage1, CPropertyPage)
   //{{AFX_MSG_MAP(CMyPage1)
      // NOTE: the ClassWizard will add message map macros here
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

As its declaration, the implementation of CMyPage2 is virtually identical to that of CMyPage1.

8.4.2 Adding a Property Sheet Object

Now that the property pages have been constructed, the one remaining task is to create the property sheet. Again, we need to invoke the new property sheet when the user selects a new menu command, Property Sheet, from the application's View menu. Add this command to the menu using the resource editor, and invoke the ClassWizard to add a corresponding member function, CMainFrame::OnViewPropertysheet, to the CMainFrame class.

In this member function, we have to perform a series of tasks. First, a property sheet object must be constructed. Next, the property pages must be added to it using the AddPage member function; and finally, the property sheet must be invoked using the DoModal member function.

Listing 8.9 contains the implementation of CMainFrame::OnViewPropertysheet that performs all these tasks.

Listing 8.9 The CMainFrame::OnViewPropertysheet function

```
void CMainFrame::OnViewPropertysheet()
{
   // TODO: Add your command handler code here
    CPropertySheet myPropSheet;
   CMyPage1 myPage1;
   CMyPage2 myPage2;
    myPage1.m_sEdit1 = "First";
```

```
    myPage2.m_sEdit2 = "Second";
    myPropSheet.AddPage(&myPage1);
    myPropSheet.AddPage(&myPage2);
    myPropSheet.DoModal();
}
```

Do not forget to include the header files MyPage1.h and MyPage2.h in MainFrm.cpp; otherwise, you will not be able to declare objects of type CMyPage1 or CMyPage2 and the function in Listing 8.9 will not compile.

At this time, the application is ready to be compiled and run.

Although in this example we made no use of the property page member variables after the property sheet is dismissed, we could access them simply through the property page objects myPage1 and myPage2.

8.4.3 CPropertyPage Member Functions

Our simple example did not utilize many of the advanced capabilities of the CPropertyPage class.

For example, in a more realistic application, you may wish to override the CancelToClose member function whenever a change is made to a property page. This member function changes the OK button to Close and disables the Cancel button in the property sheet. This function is best used after an irreversible change has been made in a property page.

Another frequently used property page function is the SetModified function. This function can be used to enable the Apply Now button in the property sheet.

Other property page overridables include OnOK (called when the OK, Apply Now, or Close button is clicked in the property sheet), OnCancel (called when the cancel button is clicked in the property sheet), and OnApply (called when the Apply Now button is clicked in the property sheet).

Property sheets can also be used to implement wizard-like behavior; that is, behavior similar to the behavior of the ubiquitous wizards that can be found in many Microsoft applications. Wizard mode can be enabled by calling the SetWizardMode member function of the property sheet; in the property pages, override the OnWizardBack, OnWizardNext, and OnWizardFinish member functions.

8.4.4 Modeless Property Sheets

Using the DoModal member function of a property sheet implies modal behavior. As is the case with dialogs, it is also possible to implement a modeless property sheet.

To accomplish this, it is first of all necessary to derive our own property sheet class. This is important because at the very least, we must override its PostNcDestroy member

function to ensure that objects of this class are destroyed when the modeless property sheet is dismissed.

The new property sheet class can be created using ClassWizard. Create a new class derived from CPropertySheet, and name it CMySheet. While in ClassWizard, add the PostNcDestroy member function.

The declaration of CMySheet (in the file MySheet.h), as generated by ClassWizard, is shown in Listing 8.10.

Listing 8.10 CMySheet declaration

```
class CMySheet : public CPropertySheet
{
   DECLARE_DYNAMIC(CMySheet)
// Construction
public:
   CMySheet(UINT nIDCaption, CWnd* pParentWnd = NULL,
         UINT iSelectPage = 0);
   CMySheet(LPCTSTR pszCaption, CWnd* pParentWnd = NULL,
         UINT iSelectPage = 0);
// Attributes
public:
// Operations
public:
// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(CMySheet)
   protected:
   virtual void PostNcDestroy();
   //}}AFX_VIRTUAL
// Implementation
public:
   virtual ~CMySheet();
   // Generated message map functions
protected:
   //{{AFX_MSG(CMySheet)
      // NOTE - the ClassWizard will add and remove member
```

```
        // functions here.
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

In the implementation file, MySheet.cpp, it is necessary to modify the PostNcDestroy member function to destroy not only the property sheet object, but also any property pages associated with it. The implementation of this function, together with other, ClassWizard-supplied member function implementations for the CMySheet class, is shown in Listing 8.11.

Listing 8.11 CMySheet declaration

```
/////////////////////////////////////////////////////////////////////////
// CMySheet
IMPLEMENT_DYNAMIC(CMySheet, CPropertySheet)
CMySheet::CMySheet(UINT nIDCaption, CWnd* pParentWnd,
                   UINT iSelectPage)
    :CPropertySheet(nIDCaption, pParentWnd, iSelectPage)
{
}
CMySheet::CMySheet(LPCTSTR pszCaption, CWnd* pParentWnd,
                   UINT iSelectPage)
    :CPropertySheet(pszCaption, pParentWnd, iSelectPage)
{
}
CMySheet::~CMySheet()
{
}
BEGIN_MESSAGE_MAP(CMySheet, CPropertySheet)
    //{{AFX_MSG_MAP(CMySheet)
 // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////
// CMySheet message handlers
void CMySheet::PostNcDestroy()
{
```

```
    // TODO: Add your specialized code here and/or call the base class
    CPropertySheet::PostNcDestroy();
    for (int i = 0; i < GetPageCount(); i++)
        delete GetPage(i);
    delete this;
}
```

A modeless property sheet does not have OK, Cancel, and Apply Now buttons by default. If any buttons are required, these must be added by hand. We are not going to worry about these now; the modeless property sheet can still be dismissed by closing it through its control menu.

How is the modeless property sheet invoked? Obviously, we have to modify the OnViewPropertysheet member function in our CMainFrame class, as using DoModal is no longer appropriate. Nor is it appropriate to create the property sheet or any of its property pages on the stack, as we do not want them destroyed when the OnViewPropertysheet function returns. The new OnViewPropertysheet function is shown in Listing 8.12.

Listing 8.12 Invoking a modeless property sheet

```
void CMainFrame::OnViewPropertysheet()
{
    // TODO: Add your command handler code here
    CMySheet *pMyPropSheet;
    CMyPage1 *pMyPage1;
    CMyPage2 *pMyPage2;
    pMyPropSheet = new CMySheet("");
    pMyPage1 = new CMyPage1;
    pMyPage2 = new CMyPage2;
    pMyPage1->m_sEdit1 = "First";
    pMyPage2->m_sEdit2 = "Second";
    pMyPropSheet->AddPage(pMyPage1);
    pMyPropSheet->AddPage(pMyPage2);
    pMyPropSheet->Create();
}
```

In order for CMainFrame::OnViewPropertysheet to compile in its new form, it is necessary to add the include file MySheet.h to MainFrm.cpp; otherwise, the attempt to declare an object of type CMySheet will fail.

The application is now ready to be recompiled and run.

Summary

In MFC, dialogs are represented by classes derived from CDialog.

The steps of constructing a dialog that is part of an MFC application are as follows:

1. Create the dialog template resource.
2. Invoke ClassWizard and create the dialog class corresponding to the resource.
3. Through ClassWizard, add member variables corresponding to controls.
4. Still using ClassWizard, add message handlers if necessary.
5. Add code to your application that constructs a dialog object, invokes it (through the DoModal member function), and retrieves results.

MFC applications can also have modeless dialogs. These dialogs are constructed differently. The constructor function in your dialog class should call the Create member function; it should also call the modeless version of the constructor of the CDialog base class. The modeless dialog must also explicitly be made visible by calling the ShowWindow member function.

Classes that correspond to modeless dialogs should override the OnOK and OnCancel member functions and call the DestroyWindow member function from within them. They should also override PostNcDestroy and destroy the C++ object (using delete this, for example).

Controls in a dialog are often represented by member variables in the corresponding dialog class. To facilitate the exchange of data between controls in the dialog box object and member variables in the dialog class, the Dialog Data Exchange mechanism can be used. This mechanism provides a simple method for matching member variables to controls. Member variables can be of simple value types or can represent control objects. It is possible to simultaneously use a member variable of a simple type to obtain the value of a control while using a control object to manage other aspects of the control. The Dialog Data Exchange mechanism also offers data validation capabilities.

For frequently used nonstandard types, it is possible to extend the ClassWizard's ability to handle Dialog Data Exchange. New data exchange and validation routines can be added either on a per project basis or to your overall Visual C++ configuration.

Property sheets represent several overlapping dialogs, or property pages, which the user can choose by clicking on corresponding tabs in a tab control.

Creating a property sheet is a two-phase process. First, property pages must be created; second, a property sheet object must be constructed, the property pages must be added to it, and the property sheet must be displayed.

Construction of property pages involves the same steps as construction of a dialog:

1. Create the dialog template resource for every property page; ensure that the resources have the Child style, Thin border style, Titlebar style, Disabled style, and that their caption is set to the text that is desired in the corresponding tab.
2. Invoke ClassWizard and create a class derived from CPropertyPage corresponding to every dialog template resource.
3. Through ClassWizard, add member variables corresponding to controls in each property page.
4. Still using ClassWizard, add message handlers if necessary.

Once the property pages have been constructed, you can proceed with the second phase:

1. Construct a CPropertySheet object or an object of a class derived from CPropertySheet.
2. Construct a property page object for every property page you wish to add to the property sheet.
3. Add the property pages to the property sheet by calling AddPage.
4. Invoke the property sheet by calling DoModal.

It is also possible to create modeless property sheets. To implement modeless property sheets, it is necessary to derive a class from CPropertySheet and override its PostNcDestroy member function to delete not only the property sheet object, but also all of its property pages. The modeless property sheet should be invoked via the Create member function instead of DoModal.

REVIEW EXERCISE

1. Explain the CDialog Class in details.
2. What are the basic steps in constructing a dialog in VC++?
3. What are Modeless Dialogs?
4. Explain the Dialog Data Exchange mechanism.
5. What are Custom Data Types in VC++?
6. What are Property sheets in VC++? How they are constructed?
7. How to create modeless property sheets?

MULTIPLE CHOICE QUESTIONS C++

1. What is a Constructor?

(*a*) A function called when an instance of a class is initialized.

(*b*) A function that is called when an instance of a class is deleted.

(*c*) A special function to change the value of dynamically allocated memory.

(*d*) A function that is called in order to change the value of a variable.

2. A class is_____

(*a*) Data Type. (*b*) Abstract Type.

(*c*) User Defined Type. (*d*) All of these options.

3. Can two classes contain member functions with the same name?

(*a*) No.

(*b*) Yes, but only if the two classes have the same name.

(*c*) Yes, but only if the main program does not declare both kinds.

(*d*) Yes, this is always allowed.

4. In object orientated programming a class of objects can _____ properties from another class of objects

(*a*) Utilize. (*b*) Borrow.

(*c*) Inherit. (*d*) Adapt.

5. Object Oriented Technology's use of _____ facilitates the reuse of the code and architecture and its _____ feature provides systems with stability, as a small change in requirements does not require massive changes in the system:

(*a*) Encapsulation; inheritance (*b*) Inheritance; polymorphism

(*c*) Inheritance; encapsulation (*d*) Polymorphism; abstraction

6. Which of the following are class relationships?

(*a*) Is-a relationship. (*b*) Part-of relationship.

(*c*) Use-a relationship. (*d*) All of these options.

7. Which of the following is true?

(*a*) Class is an object of an object. (*b*) Class is meta class.

(*c*) Class cannot have zero instances. (*d*) None of these options.

8. The design of classes in a way that hides the details of implementation from the user is known as:

(*a*) Encapsulation. (*b*) Information hiding.

(*c*) Data abstraction. (*d*) All of these options.

9. Which are the main three features of OOP language?

(*a*) Data Encapsulation, Inheritance and Exception handling.

(*b*) Inheritance, Polymorphism and Exception handling.

(*c*) Data Encapsulation, Inheritance and Polymorphism.

(*d*) Overloading, Inheritance and Polymorphism.

10. Can two classes contain member functions with the same name?

(*a*) No.

(*b*) Yes, but only if the two classes have the same name.

(*c*) Yes, but only if the main program does not declare both kinds.

(*d*) Yes, this is always allowed.

11. Suppose that the Test class does not have an overloaded assignment operator. What happens when an assignment a=b; is given for two Test objects a and b?

(*a*) The automatic assignment operator is used. (*b*) The copy constructor is used.

(*c*) Compiler error. (*d*) Run-time error.

12. Which of the operators cannot be overloaded?

(*a*) '+' operator. (*b*) '<<' operator.

(*c*) '++' operator. (*d*) '.' operator.

(*e*) "::" operator.

13. Operator overloading

(*a*) Improves the visibility and adds simplicity. (*b*) Helps in encapsulation.

(*c*) Helps in Polymorphism. (*d*) Helps in inheritance.

14. In function overloading, functions having the same name must differ in

(*a*) Return type. (*b*) Number of arguments.

(*c*) Type of arguments. (*d*) Any of a,b,c are possible.

15. Which of the following can be overloaded?

(*a*) Functions. (*b*) Operators.

(*c*) Constructors. (*d*) All of the above.

16. Using inheritance, which of the following is not allowed

(*a*) Changing implementation of operation in parent by the subclass.

(*b*) Using implementation of operation in parent class by the subclass.

(*c*) Using attributes in parent class by the subclass.

(*d*) Having operations is subclass which do not exist in parent class.

(*e*) None.

17. Inheritance is the mechanism of class by which we can inherit properties of base class to derived class. The different forms of inheritance are as follows:

(*a*) Single. (*b*) Multiple.
(*c*) Multilevel. (*d*) Multipath.
(*e*) Hybrid. (*f*) Hierachica l.
(g) All.

18. Which of the following are class relationships?

(*a*) is-a relationship. (*b*) Part-of relationship.
(*c*) Use-a relationship. (*d*) All of these options.

19. What is inheritance?

(*a*) It is same as encapsulation. (*b*) Aggregation of information.
(*c*) Generalization and specialization. (*d*) All of these options.

20. Object orientated programming allows for extension of an object function or of class function. This ability within OOP is called _____ .

(*a*) extendibility. (*b*) expansion capacity.
(*c*) virtual extension. (*d*) scalability.

21. The ability to reuse objects already defined, perhaps for a different purpose, with modification appropriate to the new purpose, is referred to as

(*a*) Information hiding. (*b*) Inheritance.
(*c*) Redefinition. (*d*) Overloading

22. What is a base class?

(*a*) An abstract class that is at the top of the inheritance hierarchy.
(*b*) A class with a pure virtual function in it.
(*c*) A class that inherits from another class
(*d*) A class that is inherited by another class, and thus is included in that class.

23. **Statement I:** All the non-private members of the base class can be accessed from the derived class as if they were members of the derived class.

Statement II: The protected data members can be accessed in the same class or in its derived class

(*a*) Both are true. (*b*) Both are false.
(*c*) Statement I is true, statement II is false. (*d*) Statement I is false, statement II is true.

24. A derived class

(*a*) Inherits data members and member functions from base class.
(*b*) Inherits constructors and destructor.
(*c*) Object can access protected members with the dot operator.
(*d*) Inherits data members and member functions from base class as well as inherits constructors and destructor.

25. How do you define an abstract class? In other words, what makes a class abstract?

(*a*) The class must not have method definitions.

(*b*) The class must have a constructor that takes no arguments.

(*c*) The class must have a function definition equal to zero.

(*d*) The class may only exist during the planning phase.

(e) all.

26. Interface is also known as _____.

(*a*) Virtual class. (*b*) Dependent class.

(*c*) Pure abstract class. (*d*) None of these options.

ANSWERS

1. (*a*)	**2.** (*c*)	**3.** (*c*)	**4.** (*c*)	**5.** (*c*)	**6.** (*a*)	**7.** (*b*), (*c*)	**8.** (*d*)
9. (*c*)	**10.** (*c*)	**11.** (*b*)	**12.** (*d*), (*e*)	**13.** (*a*), (*c*)	**14.** (*d*)	**15.** (*d*)	**16.** (*e*)
17. (*g*)	**18.** (*d*)	**19.** (*c*)	**20.** (*a*)	**21.** (*b*)	**22.** (*d*)	**23.** (*c*)	**24.** (*d*)
25. (*e*)	**26.** (*c*).						

MULTIPLE CHOICE QUESTIONS ON MFC

1. CFile directly supports the following type of input/output

(*a*) Buffered (*b*) Unbuffered

(*c*) Both buffered and unbuffered (*d*) None of above

2. File input/output in text mode can be done using

(*a*) CFile (*b*) CStdioFile

(*c*) CMemFile (*d*) CArchive

3. CFile directly supports the following type of input/output

(*a*) Buffered (*b*) Unbuffered

(*c*) Both buffered and unbuffered (*d*) None of above

4. File input/output in text mode can be done using

(*a*) CFile (*b*) CStdioFile

(*c*) CMemFile (*d*) CArchive

5. Which resource can be present only once in an MFC application

(*a*) Accelerator table (*b*) Menu

(*c*) Icon (*d*) String table

6. CRuntimeClass is a

(*a*) Structure (*b*) Union

(*c*) Class (*d*) Macro

7. An application's EXE in debug target will be _____ than release target

(*a*) Smaller (*b*) Bigger

(*c*) Same size (*d*) Depends upon the application

8. In document/view architecture which object translates mouse and keyboard messages

(*a*) CWinApp (*b*) CFrameWnd

(*c*) The class you derive from CFrameWnd (*d*) CView

9. Which of the following class is not derived from CObject

(*a*) CCmdTarget (*b*) CRuntimeClass

(*c*) CWinApp (*d*) CView

10. What is the return value from AfxMessageBox when it is dismissed?

(*a*) 0

(*b*) Any negative value

(*c*) Any positive value

(*d*) Value corresponding to the ID of the button clicked by user

11. Arrange the following in order they are searched when a DLL is loaded memory:

(*a*) Process current directory

(*b*) Directory containing exe

(*c*) Windows directory

(*d*) Windows system directory

(*a*) C, D, A, B

(*b*) A, C, B, D

(*c*) B, A, C, D

(*d*) B, A, D, C

12. InitInstance is a member of which class.

(*a*) CWinThread (*b*) CWinApp

(*c*) CWnd (*d*) CObject

13. In an MDI application how many menu resources are there?

(*a*) At least One (*b*) At least Two

(*c*) At the most One (*d*) At the most Two

14. Which type of dialog box overlaps all other windows, including that of other applications when displayed?

(*a*) System modal (*b*) Modal

(*c*) Modeless

15. You override OnInitDialog() in your class derived from Cdialog then when should you call the base class OnInitDialog() inside your version?

(*a*) As the first statement (*b*) As the last statement

(*c*) Anywhere (*d*) You don't have to call it

16. A class derived directly from which of the following would not be able to receive command messages

(*a*) CObject (*b*) CCmdTarget

(*c*) CWnd (*d*) CDocument

17. For the purpose of constructing a modeless dialog box you use following constructor

(*a*) CDialog(LPCTSTR , CWnd* = NULL);

(*b*) CDialog(UINT nIDTemplate, CWnd* pParentWnd = NULL);

(*c*) CDialog();

(*d*) Any of the above can be used.

18. A modeless dialog box object should not be

(*a*) Created on heap (*b*) Created on stack

(*c*) Global (*d*) Static local

19. To replace the default message loop and provide your own customized version you override

(*a*) OnIdle (*b*) PreTranslateMessage

(*c*) InitInstance (*d*) Run

20. When a dll is loaded its second parameter has a value:

(*a*) DLL_ATTACH

(*b*) DLL_PROCESS_ATTACH

(*c*) DLL_ATTACH_PROCESS

(*d*) DLL_THREAD_ATTACH

21. Which of the following message will undergo command routing in document view architecture?

(*a*) WM_CHAR

(*b*) WM_COMMAND

(*c*) WM_LBUTTONDOWN

(*d*) WM_PAINT

22. Worker threads are recognized by their

(*a*) Class (*b*) Function

(*c*) Name

23. Which of the following objects cannot be used for synchronization of threads across process boundary?

(*a*) Semaphore (*b*) Mutex

(*c*) Event (*d*) Critical section

24. The third parameter that goes into CMultiDocTemplate corresponds to

(*a*) CMdiChildWnd (*b*) CMdiFrameWnd

(*c*) CView (*d*) CDocument

25. The first parameter that CSingleDocTemplate takes corresponds to the ID of which of the following resources:

(*a*) Main menu (*b*) Icon

(*c*) Resource string (*d*) All the above

26. Which of the following class is specially meent for text input and output?

(*a*) CFile (*b*) CMemFile

(*c*) CSocketFile (*d*) CStdioFile

27. Which of the following are not added to a class due to inclusion of DECLARE_ DYNCREATE macro?

(*a*) CreateObject() (*b*) GetRuntimeClass()

(*c*) CRuntimeClass object (*d*) IsKindOf()

28. Which of the following is not a function of CObject class?

(*a*) Dump() (*b*) AssertValid()

(*c*) IsDerivedFrom() (*d*) CreateObject()

29. Which of the following classes don't overload insertion operator?

(*a*) CDumpContext

(*b*) CArchive

(*c*) CObject

(*d*) CFile

30. To make an extension dll which macro should be used with a class.

(*a*) AFX_DLL_EXT (*b*) AFX_EXT_DLL

(*c*) AFX_DLL (*d*) AFX_EXT

31. Which calling convention would you used if you want to use a DLL in VB.cdecl?

(*a*) stdcall (*b*) fastcall

(*c*) Any calling convention would work.

32. How many maximum rows and columns can a static splitter window contain?

(*a*) 3 × 3 (*b*) 2 × 2

(*c*) 16 × 16 (*d*) There is no such limit.

33. Which class is used to have more than one view of a single document in an SDI application?

(*a*) CFrameWnd (*b*) CView

(*c*) CSplitterWnd (*d*) CWnd

34. Which class encapsulated individual page of a Property-Sheet?

(*a*) A class derived form CPropertySheet (*b*) A class derived form CDialog

(*c*) A class derived form CWnd (*d*) A class derived form CPropertyPage

35. If you override OnOk () in your class derived from CDialog then when should you call the base class OnOk() inside your version?

(*a*) As the first statement (*b*) As the last statement

(*c*) Anywhere (*d*) You don't have to call it

36. In an MDI application, which window owns the toolbar?

(*a*) CMdiFrameWnd derived window (*b*) CMdiChildWnd derived window

(*c*) CView derived window (*d*) CScrollWnd derived window

37. If you want to perform certain initializations, anytime a document is created in SDI, you should put the code for initialization in

(*a*) Document class constructor (*b*) OnOpenDocument

(*c*) OnDocumentFile (*d*) OnNewDocument

38. On which of the following synchronization object can you call Lock() multiple times without causing the threads to block?

(*a*) CEvent (*b*) CCriticalSection

(*c*) CMutex (*d*) CSemaphore

39. Which of the following class does not appears in a typical dialog based application?

(*a*) CDialog derived (*b*) CFrameWnd derived

(*c*) CWinApp derived (*d*) CDocument derived

40. A dynamic splitter window can be created using

(*a*) Create (*b*) CreateStatic

(*c*) CreateDynamic (*d*) Any of the above

41. Which class represents connection to a data source?

(*a*) CRecordSet (*b*) CDatabase

(*c*) CConnection (*d*) CRecordView

42. Which of the following statement is false regarding CObject?

(*a*) Private constructor (*b*) Privately overloaded = operator

(*c*) Overloaded new and delete (*d*) Publicly overloaded = operator

43. Which of the following is not a resource?

(*a*) Accelerator table (*b*) Bitmap

(*c*) Toolbar (*d*) File

UNSOLVED QUESTIONS

1. The _____ member function of a class is overriden for serialization

(*a*) OnSerialize( ) (*b*) OnNewDocument( )

(*c*) OnClose ( ) (*d*) OnWrite ( )

(*e*) None of the above.

2. _____ macro calls the function AssertValid( ).

(*a*) ASSERT_VALID ( ) (*b*) TRACE2 ( )

(*c*) VERIFY ( ) (*d*) ASSERT ( )

(*e*) None of the above (*f*) a and d

3. DECLARE_MESSAGE_MAP ( ) is used in a class to

(*a*) Recieve and Handle messages

(*b*) To be able to send messages to other windows

(*c*) inorder for the window to be resizable

(*d*) None of the above

4. RUNTIME_CLASS( ) macro is used to

(*a*) RTTI (*b*) Serialization

(*c*) Memory Management (*d*) Debug and Diagnostics

(*e*) None of the above (*f*) b and c.

5. To fill the client area of the window displayed on screen on should trap the

(*a*) OnCreate ( ) (*b*) OnCreateClient ( )

(*c*) OnRun ( ) (*d*) OnPaint ( )

6. CSingleLock utility class is used for thread synchronization

(*a*) For more than two threads accessing the same data

(*b*) For two threads accessing two sets of different data

(*c*) For multiple documents
(*d*) None of the above

7. DeleteContents ( ) is a member function of
(*a*) CMyDocument (*b*) CWinThread
(*c*) CDocument (*d*) CWinApp

8. Which function in the View is first called when the View is attached to the Document?
(*a*) OnUpdate ( ) (*b*) UpdateAllViews ( )
(*c*) OnInitialUpdate ( ) (*d*) OnNewDocument ( )

9. One must pass the _____ parameter to the function UpdateAllViews ( ) in order to send notification for all the views to be updated
(*a*) 1 (*b*) NULL
(*c*) this (*d*) None of the above.

10. Attach ( ) is a function which takes in a _____ parameter
(*a*) Pointer to the object (*b*) Pointer to the Device context
(*c*) Handle to a Window (*d*) None of the Above.

11. One should normally create an object of the CPaintDC only in
(*a*) OnDraw ( ) (*b*) OnCreate ( )
(*c*) PreCreateWindow ( ) (*d*) OnPaint ( )
(*e*) None of the above.

12. The OnCreate ( ) member function of the CFrameWnd calls _____ function of the same class to create the view
(*a*) OnCreateView ( ) (*b*) OnCreateControl ( )
(*c*) OnCreateApplication ( ) (*d*) OnCreateClient ( )
(*e*) b and c.

13. SetModified ( ) is a member function of the class
(*a*) CPropertySheet (*b*) CDialog
(*c*) CPropertyPage (*d*) None of the above.

14. UpdateData (TRUE) is used to transfer the values from
(*a*) Control to variable (*b*) variable to control
(*c*) Variable to View (*d*) b and c
(*e*) None of the above

15. In order to terminate an appliction from the OnCreate ( ) member function, one must return
(*a*) 0 (*b*) 1
(*c*) –1 (*d*) 2

ANSWERS

1. (*a*)	**2.** (*b*)	**3.** (*a*)	**4.** (*b*)	**5.** (*d*)	**6.** (--)	**7.** (*b*)	**8.** (*d*)
9. (*b*)	**10.** (*d*)	**11.** (*d*)	**12.** (*a*)	**13.** (*b*)	**14.** (*a*)	**15.** (*a*)	**16.** (*a*)
17. (*c*)	**18.** (*b*)	**19.** (*d*)	**20.** (*b*)	**21.** (*b*)	**22.** (*b*)	**23.** (*d*)	**24.** (*a*)
25. (*d*)	**26.** (*d*)	**27.** (*d*)	**28.** (*c*)	**29.** (*c*)	**30.** (*b*)	**31.** (*b*)	**32.** (*c*)
33. (*c*)	**34.** (*d*)	**35.** (*b*)	**36.** (*a*)	**37.** (*d*)	**38.** (*d*)	**39.** (*b*), (*d*)	**40.** (*a*)
41. (*b*)	**42.** (*d*)	**43.** (*d*)					

FILL IN THE BLANKS

1. _____ is the function for creating a modal dialog box on screen.

2. _____ function in a class derived from CDialog is modified by the class-wizard to provide you actual data transfer functionality between any control and the corresponding member variable.

3. In order to be able to receive a command message _____ class must be derived from which M.F.C. given class.

4. _____ virtual member function in the above class do we need to override in any of its derived classes in order to change the default command route followed in a doc-view architecture based application.

5. _____ function of CSplitterWnd class do we need to call in order to create a static splitter on the mainframe window?

6. The three letters AFX in MFC means _____ and the prefix Afx (as in AfxGetMainWnd) to any method in MFC indicates _____ of the method.

7. To open modeless dialog window in MFC one calls _____ function.

8. _____ can be used for locking on a Mutex object.

9. The pointers of the view created in a Document View Architecture are maintained by the _____.

10. OnIdle is a member function of _____ .

11. Inorder to receive any messages a class should be derived from _____ .

12. The four functionalities provided by the class CObject are _____, _____, _____ & _____.

13. To create a modeless dialog box one should use the _____ API.

14. _____ Macro is used to Dump the values in the Debug pane window.

15. In order for a class to be serializable one should add _____ in the header file of the class and _____ in the implementation (i.e. .cpp) file.

16. _____ should be added in the header file & _____ should be added in the .cpp i.e the implementation file inorder to achieve Runtime Type Information.

17. m_pMainWnd originates from the _____ top most class.

18. OnUpdate( ) function is a member function of the class _____ .

SOLVED FILL IN THE BLANKS

19. Afx prefixed to any method in MFC indicates **global** function.
20. For a class to receive command message it should be derived from **CCmdTarget**.
21. For a class to be support serialization it must be derived from **CObject**.
22. In Windows 32 bit O.S. every **thread** has a message queue.
23. **CWinThread** class in MFC represents the above O.S component.
24. **OnCmdMsg** virtual member function of CCmdTarget needs to be overridden to change the default message route followed in doc-view architecture based application.
25. **Run** member function of CWinThread class provides implementation of message loop.
26. When message queue is found empty windows call **OnIdle** member function of CWinThread class.
27. **CGdiObject** is the base class for all the GDI objects.
28. **RUNTIME_CLASS** macro can be used to get the pointer to a CRuntimeClass structure given its class name.
29. **Alt** and **F10** are the two keys that generate WM_SYSKEYUP.
30. **DrawMenuBar** function of CWnd class is used to redraw the menu bar to reflect any changes.
31. To open modeless dialog window in MFC on calls **Create** function.
32. The function to show modal dialog box on screen is **DoModal**.
33. **CpropertySheet:: AddPage()** function is used to add pages to a property sheet.
34. An application cannot have **String table** resource multiple times.
35. MFC uses **LoadFrame** function of CWnd class to load menu and other resources all in a single go in document/view architecture.
36. **(WM_COMMAND)** and **(WM_UI_COMMAND_UPDATE)** macros correspond to the messages that are subject to routing in an MFC application.
37. An MDI uses **CMultiDocTemplate** while an SDI uses **CSingleDocTemplate** to create a document template.
38. An MDI's top-level window is derived from **CMdiFrameWnd** and its child windows are derived from **CMdiChildWnd**.
39. A dynamic splitter window can contain a maximum of **16** rows and **16** columns.
40. Static splitter window is created using **CreateStatic** member function of CSplitterWnd class.
41. **AfxBeginThread** global function can be used for creating a thread.
42. **CPaintDC** class can be used for handling WM_PAINT message in MFC.
43. In a simple MFC application main thread of execution is provided by **CWinApp** object.
44. Critical Section, Event, Mutex are basically **kernal** objects provided by operating system.
45. **1Lock** member function of CEvent class is called to block on an event.
46. **Manual reset** and **auto reset** are two types of CEvent objects.
47. **Unlock ()** member function of CMutex is used to release a mutex.

48. **LoadAcclTable ()** function is used to load an accelerator table.
49. **StretchBlt ()** function of CDC class can be used to increase or decrease the size of image.
50. **DllMain** is the entry point in the dll.
51. The size of the extension dll is **smaller** than that of regular dll of same type.
52. An extension dll links **statically** to the code in MFC library.
53. CSocket class is derived from **CAsyncSocket**.
54. Accept function of CSocket is **blocking** in nature while that of CAsyncSocket is **non-blocking** in nature.
55. To connect to server from a client we use **Connect** function of CSocket class.
56. To accept a connection from a client on a server we use **Accept** function of CSocket class.
57. To support serialization CFile takes the help of **CArchive** class.
58. In MFC **CFile** class is used to handle input/output from a file.
59. If the schema number of the object on the disk does not match the schema number of the class in memory, MFC throws **CArchiveException** exception.
60. **IsStoring** function of CArchive class is used to determine whether the object is being stored or retrieved.
61. **Seek ()** function of CFile is used to move to a specific position within a file.
62. AssertValid function is available is **debug** mode of application development.
63. To have a scroll bar in the main window of your application you use **CScrollView** class.
64. When user double clicks an entry in a ListBox, **(LBN_DBLCLK)** notification is sent to the parent window.
65. The third parameter that goes into CMultiDocTemplate corresponds to the **child** frame of the application.
66. **ProcessShellCommand** is used to dispatch commands specified on the command- line in the InitInstance function.
67. **SetModifiedFlag (TRUE)** member function of CDocument is called to indicate to the view that the document has changed.
68. Any time a new document is created or opened in SDI, **eleteContent()** function
69. _____is called by the framework to delete previous data.
70. **Detach()** of CWnd class can be used to dissociate a menu from a CMenu object.
71. In a document view architecture an object of **CDocument** is used to store data and _____an object of **CView** derived class is used to render the output.
72. Clicking a Toolbar item produces **(WM_COMMAND)** message.
73. CEvent, CCriticalSection, CSemaphore and CMutex classes are all derived from **CSyncObject**.
74. Two types of CEvent objects are **manual reset** and **auto reset**.
75. The two functions which are used for enumerating files and folders are ::FindFirstFile and :: **FindNextFile**.

TRUE OR FALSE

1. All MFC classes are derived from CObject.
2. Toolbar is child of the view window.
3. An applications top level menu can be changed at runtime.
4. A class supporting Serialization has to use DECLARE_SERIAL and IMPLEMENT_SERIAL macros.
5. AfxBeginThread can create only worker threads.
6. OnPaint and OnDraw member functions serve the same purpose.
7. Runtime class information supported by CObject is same as RTTI of C++.
8. InitInstance is member of CWinThread class.
9. DoDataExchange() of CDialog calls UpdateData() function.
10. To create a propertysheet on screen we need to call the CreatePropertySheet() function.
11. In a document-View based SDI application which object creates the frame window object.
12. CreateStatic is a member function of the CFrameWnd class.
13. We can access the pointer of the document in the constructor of the view.
14. In the view class OnDraw function is called by OnInitialUpdate().
15. Object of CCreateContext class is passed as parameter to PreCreateWindow.
16. The VERIFY macro is available for release build of MFC.
17. The CreateThread is a global function.
18. AfxSetResourceHandle is used for message dispatching.
19. MFC classes can be exported out of regular DLLs.
20. WaitForSingleObject can be used for locking on a Critical section object.
21. The view is created in the OnNewDocument of Document object.
22. It's a must to override the function OnIdle.
23. OnSetActive is a member function of CPropertySheet.
24. CSyncObject is the base class for CSingleLock.
25. One can have more than one global object for the Application class in a project.
26. The Run( ) function of the CWinThread class can be overriden.
27. There can be more than one document class present in a SDI (Single Document Interface) application.
28. Is it necessary to create the view immediately after you create the static splitter window.
29. OnDraw ( ) is a member function also present in the CDocument class.
30. If you want to create a set of buttons on the view, you would trap the OnCreate ( ) function of the class CMainFrame.
31. The code to be executed for a thread, is placed in the constructor of the class which is derived from CWinThread.
32. afx_msgGetApp( ) is the API used to get a pointer to the object representing the Application.

33. If the macro DECLARE_SERIAL ( ) is added in a class, you can also add other macros like DECLARE_DYNAMIC ( ) & DECLARE_DYNCREATE( ) in the same class.

34. GetDocument( ) is a member function, overriden in your document class which returns a pointer to the Active Document in an SDI application.

35. There can be only one CWinApp object in an MFC application. [True]

36. Runtime class information supported by CObject is same as RTTI of C++. [False]

37. MFC uses virtual functions to implement messages. [False]

38. All MFC classes are derived from CObject. [False]

39. CObject class is also available outside MFC framework. [True]

40. CRuntimeClass is derived from CObject. [False]

41. OnPaint and OnDraw functions serve the same purpose. [False]

42. You can't call MessageBox() function inside InitInstance (). [True]

43. A window in windows application gets all the mouse messages that are generated by the user. [False (Only command messages.)]

44. Drop-down menus are actually pop-up menus. [True]

45. If two or more items in the same menu are assigned the same shortcut key it will generate an error when that key is pressed. [False (Focus toggles between items.)]

46. DoModal doesn't returns until the dialog box is dismissed. [True]

47. Modeless dialog box is dismissed using EndDialog. [False (Using DestroyWindow.)]

48. Property sheets support DDX and DDV. [True]

49. In document/view architecture application object and document object can receive all types of messages. [False (Only Command messages)]

50. Toolbar is a child of view window. [False. (Frame window.)]

51. An application top-level menu can be changed at runtime. [True]

52. In an MDI application we can open several different types of documents simultaneously. [True]

53. You can omit individual sub string in a resource document string in do document/view architecture. [True]

54. In document/view architecture OnDraw has to be overridden. [True]

55. In an SDI for each new view, a new document object is created and attached to the view. [False. (Views are reused.)]

56. A document object can process a keyboard and mouse messages. [False]

57. An MDI can have one or many menu resources. [False (Cannot have one menu resource)]

58. In an MDI application we can have different icons for mainframe window and the child form windows. [True]

59. In an MDI application different child frame window can have different icons. [True]

60. A static splitter must have at least one sharing scroll bar. [True]

61. If we have two panes in a static splitter window, housing two views of a document, then change in the document data automatically changes the two views.
[False. (We need to do it using CDocument::UpdateAllViews)]

62. SDI supports only one document type. [True]

63. In a document/view architecture the application object is created dynamically. [True]

64. The drag and drop support is provide through the function RegisterShellFileTypes (). [True]

65. You can call AddDocTemplete multiple times in an SDI. [False]

66. Both Windows and MFC treat UI and worker threads differently.
[False. (Windows makes no distinction between these threads)]

67. Both UI and worker threads have message loops. [False. (Only UI threads)]

68. Two or more threads may have same thread function. [True]

69. A thread function is a callback function. [True]

70. AfxBeginThread can create only worker threads. [False]

71. Critical Sections can be used for synchronization within same or different process.
[False. (Only within same process.)]

72. Events can be used for synchronization within same or different process. [True]

73. When a priority is specified for a thread it is relative to all other threads in the operating system.
[False. (Only relative to the threads in the same process)]

74. Bitmaps can be selected in any device context. [False. (Only memory device context.)]

75. When a memory device context is first created its size can be anything.
[False. (Size of one monochrome pixel.)]

76. A file that is dynamically linked can have any extension. [True]

77. Global variables in a dll are same for all the processes that are linked to that dll. [False. (Private to each process.]

78. DlllMain () is called only once in the lifetime of a dll. [False. (Four times.)]

79. In static linking the object code of the library does not becomes part of the executable but in dynamic linking it does. [False. (Vice-versa)]

80. Every dll must contain a DllMain () provided by the user.
[False. (The framework provides a dummy DllMain () if u don't provide one.)]

81. An extension dll can't be use by visual basic clients. [True]

82. It is necessary that a dll used by the client program must either be present in the client program' s directory or in the Windows' system directory.
[False. (Can also be present in directory specified by 'path' environment variable or current directory.)]

83. Every time you make a change to dll you have to recompile the applications using the dll.
[False. (Only if you have made changes to prototypes of exported functions.)]

84. CFile class directly supports both binary and text input and output.
[False. (Only binary I/O is directly supported. Its derived classes are used for text I/O.)]

85. Same CArchive object is used for both storing and loading an object during serialization throughout the application lifetime. [False. (Each time a new object is created.)]

86. The CScrollView class does not support scrolling from the keyboard by default. [True]

87. Every MFC application that has a UI must have a class derived from CView. [False. (Dialog based application don't have to.)]

88. When loading DLL windows searches the directory defined in the path environment variable before it searches the current directory. [False. (At the end.)]

89. Dialog box messages are not passed to the main window of application. [True]

90. A tool bar is also a window that is child of the window hosts it. [True]

91. The repeat count parameter for a key-up message can never be anything but one. [True]

92. Operating system gives higher priority to UI threads as compared to worker threads.

[False (O.S makes no distinction between UI and Worker threads. The programmer decides the priority if required.)]

93. Toolbars can also contain items that do not appear in menu. [True]

94. Toolbars cannot have Texts. [False]

95. "It important to forward non-client area and system keyboard messages to base class if you process them". [True]

QUESTIONS

1. SetModifiedFlag() is a member function of which class and why do we need to use it?

2. DeleteContents() is a member function of which class and why do we need to override it?

3. Which class serves as a helper class for implementing RTTI mechanism?

4. Write the preprocessor directive used to differentiate between the release and debug modes while writing any M.F.C. application.

5. Which function can be used for getting the address of the main frame window pointer anywhere in a M.F.C. application?

6. Write the M.F.C. given global function for creating a thread.

7. Write any two classes, which can be used for synchronizing multiple threads in M.F.C.

8. Which member function of the frame class is actually responsible for loading the resources for the main frame window?

9. Write the member function of the CPropertySheet class, which returns a pointer to the current active page in it.

10. Which device context class can be used for handling WM_PAINT message in M.F.C?

11. What is the purpose of the two parameters in the BEGIN_MESSAGE_MAP (,) macro?

12. What does MFC do when you call the function DoModal() to open modal dialog window

13. What does "AFX" stand for?

14. How can we call Win32 API functions in MFC program?

15. In an MDI application, which window owns the toolbar?

16. What is the difference between "grayed" and "disabled" menu item?

17. Why must a serializable class have default constructor?

18. Write the preprocessor directive used to differentiate between debug and release modes when writing any MFC application.

19. Void OnDraw (CDC *pDC) is a member of which class and on what occasions does the framework call it?

20. How can we call Win32 API functions in MFC program?

21. Which virtual function should you override if you want to filter some messages and what is its class?

22. Name the six GDI objects.

23. What is the purpose of ON_COMMAND_RANGE macro?

24. What is the purpose of resource.h file?

25. Why is it necessary to override OnOk and OnCancel for modeless dialog box?

26. How can you set the focus on a control programmatically in a dialog box?

27. How can on iterate through all the views associated with a document? Give the function names and their required parameters.

28. Why do we need to override OnInitialUpdate member function of CView sometimes?

29. In what order do application, frame widow, document and view objects get created?

30. Why is it not necessary to have message map entry for the "File | Save" menu item?

31. Why is AfxBeginThread instead of ::CreateThread recommended in MFC for creating a thread.

32. How can main thread and the one that it spawns communicate?

33. What is an import library?

34. Is it necessary to override InitInstance() in any window based MFC application? Why

35. What is the use of _T macro?

36. Why should you call base class OnIdle version if you override it?

37. What is command routing?

38. How you can define and use your own messages?

39. What are owner drawn controls?

40. How to display text in status bar?

41. How to give effect of blinking to a line?

42. How to handle communication using MFC?

43. Why MFC extension Dll's are used?

44. How to handle application resources in Dll's?

45. What are accelerator keys?

46. How to loadbitmaps on buttons using MFC?

47. How to draw circle using MFC?

48. What are different types of dialog boxes?

49. What is a modal dialog box?

50. What is a modeless dialog box and how do you create it?

51. What are document templates?

52. What code is written in InitInstance?

53. How we load libraries dynamically?

54. How one can add a addtitional toolbar?

55. How one can change a cursor shape?

56. When in WM_PAINT message issued?

57. Why Invalidate function of CWnd used?

58. How you can change the size of a window using MFC?

59. How you can change the application title?

60. How can you display an image in the MDI client window?

61. How can you convert a SDI application framework to a MDI appliction framework?

62. Will deleting a modeless dialog box without destroying it, give memory leaks? (creating on heap & stack)

63. When should we use CWnd's 'DestroyWindow' member function?

64. How are memoryleaks detected in debug mode of an application?

65. What is the difference between internal & external make files?

66. Can U acccess a memeber of application class in View class?

(Ans: AfxGetApp() : returns the pointer to the application object)

67. Why is MFC class 'CRecordSet' used? (aans: for database iteraction)

APPENDIX I

EXCEPTION HANDLING

The Win32 API supports *structured exception handling*. Through this mechanism, applications can handle various hardware- and software-related conditions. Structured exception handling is not to be confused with the concept of exceptions in the C++ language, which is a feature of that language. The Win32 exception handling mechanism is not dependent on its implementation in a specific language. To avoid confusion, I decided to follow the conventions used in Microsoft documentation and use the term "C exception" to refer to Win32 structured exceptions, and "C++ exception" to refer to the typed exception handling mechanism of the C++ language.

Exception Handling in C and C++

Microsoft provides a set of extensions to the C language, which enable C programs to handle Win32 structured exceptions. This exception handling mechanism is markedly different from the typed exceptions in the C++ language. This section offers a review of both mechanisms in the context of exceptions in the Win32 environment.

C Exceptions

What is, indeed, an exception? How do exceptions work? In order to understand the exception handling mechanism, first take a look at the program shown in Listing 1.

Listing 1 A Program that Generates an Exception

```
void main(void)
{
   int x, y;
   x = 5;
   y = 0;
   x = x / y;
}
```

Needless to say, an integer division by zero is likely to cause a program to terminate abnormally. If you compile the above program and run it under Windows 95, it generates the dialog shown in Figure 17.1.

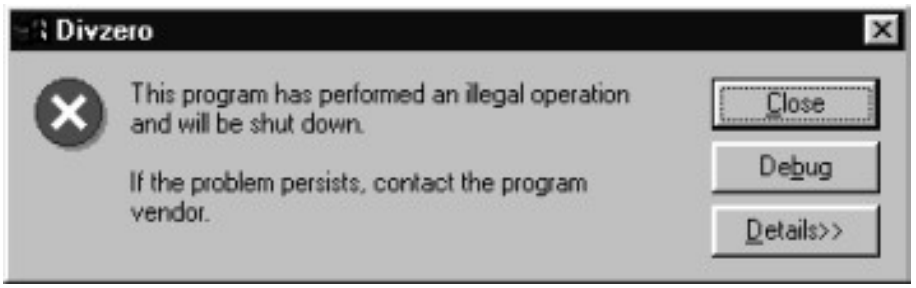


Figure 1: Division by zero error

What exactly happened here? Obviously, when you attempt to divide by zero, the processor will generate an error condition (the actual mechanism is hardware dependent and not of our concern). This error condition is detected by the operating system, which looks for an *exception handler* for the specific error condition. As no such handler was detected, the default exception handling mechanism took over, displaying the dialog.

Using the C exception handling mechanism, it is possible for us to *catch* this exception and handle the divide by zero condition gracefully. Consider the program shown in Listing 2.

Listing 2. Handling the divide by zero exception

```
#include "windows.h"
```

```
void main(void)
{
   int x, y;
   __try
   {
      x = 5;
      y = 0;
      x = x / y;
   }
   __except (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
            EXCEPTION_EXECUTE_HANDLER :
            EXCEPTION_CONTINUE_SEARCH)
   {
      printf("Divide by zero error.\n");
   }
}
```

Running this program no longer produces the dialog shown in Figure 1; instead, the message "Divide by zero error." is printed and the program terminates gracefully.

The block of statements following the __try instruction is often called a *guard block*. This block of statements is executed unconditionally. When an exception is raised within the guard block, the expression following the __except statement (often called the *filter expression*) is evaluated. This expression should be an integer expression yielding one of the following values:

Table 1: Filter expression values

Symbolic constant	Value	Description
EXCEPTION_CONTINUE_EXECUTION	–1	Continue execution at the location where exception was raised
EXCEPTION_CONTINUE_SEARCH	0	Pass control to next exception handler
EXCEPTION_EXECUTE_HANDLER	1	Execute exception handler

If the filter expression's value is -1 (EXCEPTION_CONTINUE_EXECUTION), execution continues at the location where the exception was raised. That is, *at* the location, not *after—w*hich means that the offending piece of code may get executed again. Whether it actually does get executed or not depends on the type of the exception. For example, in the case of an integer division by zero, it does; in the case of a floating-point division by zero, it does not. In any case, care should be taken to avoid

creating an infinite loop by returning control to the point where the error occurs without eliminating the conditions which caused the exception in the first place.

In the other two cases, the first thing that happens is that the guard block goes out of scope. Any function calls that might have been interrupted by the exception are terminated and the stack is unwound.

If the filter expression evaluates to 1 (EXCEPTION_EXECUTE_HANDLER), control is transferred to the statement block following the __except statement.

The third filter value, 0 (EXCEPTION_CONTINUE_SEARCH), hints at the possibility of nested exceptions. Indeed, consider the program shown in Listing 17.3. In this program, two exceptions are generated, one for a floating-point division by zero, one for an integer division by zero. The two exceptions are handled very differently.

Listing 3. Nesting exception handlers

```
#include <stdio.h>
#include <float.h>
#include <windows.h>
unsigned int divzerofilter(unsigned int code, int *j)
{
   printf("Inside divzerofilter\n");
   if (code == EXCEPTION_INT_DIVIDE_BY_ZERO)
   {
      *j = 2;
      printf("Handling an integer division error.\n");
       return EXCEPTION_CONTINUE_EXECUTION;
   }
   else return EXCEPTION_CONTINUE_SEARCH;
}
void divzero()
{
   double x, y;
   int i, j;
   __try
   {
      x = 10.0;
      y = 0.0;
      i = 10;
```

```
        j = 0;
        i = i / j;
        printf("i = %d\n", i);
        x = x / y;
        printf("x = %f\n", x);
    }
    __except (divzerofilter(GetExceptionCode(), &j))
    {
    }
}
void main(void)
{
    _controlfp(_EM_OVERFLOW, _MCW_EM);
    __try
    {
        divzero();
    }
    __except (GetExceptionCode() == EXCEPTION_FLT_DIVIDE_BY_ZERO ?
            EXCEPTION_EXECUTE_HANDLER :
            EXCEPTION_CONTINUE_SEARCH)
    {
        printf("Floating point divide by zero error.\n");
    }
}
```

When an exception is raised inside the divzero function, the filter expression is evaluated. This results in a call to the divzerofilter function. The function checks if the exception was an integer division by zero exception; if so, it corrects the value of the divisor (j) and returns the EXCEPTION_CONTINUE_EXECUTION value, which causes the exception handling mechanism to return control to the point where the exception was raised. In the case of any other exceptions, divzerofilter returns EXCEPTION_CONTINUE_SEARCH; this causes the exception handling mechanism to seek another exception handler.

This other exception handler has been installed in the main function. This handler handles floating-point division by zero exceptions. Instead of returning to the point where execution was interrupted, it simply prints an error message.

Running this program produces the following output:

```
Inside divzerofilter

Handling an integer division error.

i = 5

Inside divzerofilter

Floating point divide by zero error.
```

As you can see, both times an exception is raised, the exception filter installed in the function divzero is activated. However, in the case of the floating-point division, the exception remains unhandled; therefore, the exception is propagated to the next level, the exception handler installed in the main function.

NOTE: To handle floating-point exceptions, it was necessary to call the _controlfp function. This function can be used to enable floating-point exceptions. By default, floating-point exceptions on the Intel architecture are disabled; Instead, the floating-point library generates IEEE-compatible infinite results.

A discussion of C exception handling would not be complete without a list of some of the commonly occurring C exceptions. These exceptions are shown in Table 2.

Table 2. Filter expression values.

Symbolic constant	Description
EXCEPTION_ACCESS_VIOLATION	Reference to invalid memory location
EXCEPTION_PRIV_INSTRUCTION	Attempt to execute privileged instruction
EXCEPTION_STACK_OVERFLOW	Stack overflow
EXCEPTION_FLT_DIVIDE_BY_ZERO	Floating-point division
EXCEPTION_FLT_OVERFLOW	Floating point result too large
EXCEPTION_FLT_UNDERFLOW	Floating point result too small
EXCEPTION_INT_DIVIDE_BY_ZERO	Integer division
EXCEPTION_INT_OVERFLOW	Integer result too large

In addition to system-generated exceptions, applications can raise software exceptions using the RaiseException function. Windows reserves exception values with bit 29 set for user-defined software exceptions.

C Termination Handling

Closely related to the handling of C exceptions is the topic of C termination handling. To better understand the problem of which termination handling provides a solution, consider the program shown in Listing 4.

Listing 4. Resource allocation problem.

```
#include <stdio.h>
#include <windows.h>
void badmem()
{
   char *p;
   printf("allocating p\n");
   p = malloc(1000);
   printf("p[1000000] = %c\n", p[1000000]);
   printf("freeing p\n");
   free(p);
}
void main(void)
{
   __try
   {
      badmem();
   }
   __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER :
            EXCEPTION_CONTINUE_SEARCH)
   {
      printf("An access violation has occurred.");
   }
}
```

In this program, the function badmem allocates the p character array. However, its execution is interrupted when it refers to an invalid array element. Because of this, the function never has a chance to free up the allocated array, as demonstrated by its output:

allocating p

An access violation has occurred.

This problem can be solved by installing a termination handler in the badmem function, as shown in Listing .5.

Listing 5. A termination handler

```
#include <stdio.h>
#include <windows.h>
void badmem()
{
   char *p;
   __try
   {
      printf("allocating p\n");
      p = malloc(1000);
      printf("p[1000000] = %c\n", p[1000000]);
   }
   __finally
   {
      printf("freeing p\n");
      free(p);
   }
}
void main(void)
{
   __try
   {
      badmem();
   }
   __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER :
            EXCEPTION_CONTINUE_SEARCH)
   {
      printf("An access violation has occurred.");
   }
}
```

Running this program produces the desired result:

```
allocating p
```

freeing p

An access violation has occurred.

As you can see, the instructions in the badmem function are now enclosed in a __try block, which is now followed by the __finally keyword. The __finally keyword is special in that the instruction block that follows it is *always* executed, no matter under what circumstances the function terminates. So when badmem goes out of scope due to the exception, the instructions in the __finally block are given a chance to clean up any resources that might have been allocated within this function.

C++ Exception Handling

The Win32 exception handling mechanism uses the GetExceptionCode function to determine the nature of the exception. In contrast, C++ exception handling is type-based; the nature of the exception is determined by its type.

Most examples that demonstrate C++ exception handling do so in the context of a class declaration. This is not necessary, and in my opinion often hides the simplicity of C++ exception handling. Consider the simple example in Listing 17.6. (When you compile this example or any other program that uses C++ exceptions, do not forget to add the -GX switch to the cl command line.)

Listing 6. C++ Exception handling.

```
#include <iostream.h>
int divide(int x, int y)
{
   if (y == 0) throw int();
   return x / y;
}
void main(void)
{
   int x, y;
   try
   {
      x = 5;
      y = 0;
```

```
        x = divide(x, y);
    }
    catch (int)
    {
        cout << "A division by zero was attempted.\n";
    }
}
```

In this example, the function divide raises an exception of type int when a division by zero is attempted. This exception is caught by the exception handler in main.

Termination Handling in C++

C++ exceptions can also be used for termination handling. For termination handling, a C++ program can wrap a block of code using a "catchall" exception handler, and perform resource cleanup before propagating all exceptions to a higher level handler by using throw. Consider the example in Listing 7, which is a C++ variant of the program shown in Listing 7.

Listing 7. Termination handling with C++ exceptions

```
#include <stdio.h>
#include <windows.h>
void badmem()
{
    char *p;
    try
    {
        printf("allocating p\n");
        p = (char *)malloc(1000);
        printf("p[1000000] = %c\n", p[1000000]);
    }
    catch(...)
    {
        printf("freeing p\n");
        free(p);
        throw;
```

```
    }
}
void main(void)
{
    try
    {
        badmem();
    }
    catch(...)
    {
        printf("An exception was raised.");
    }
}
```

Running this program produces the following output:

allocating p

freeing p

An exception was raised.

The exception handler in the function badmem plays the role of the __finally block in the C exception handling mechanism.

Although these examples demonstrate the power of C++ exception handling with C-style code, the use of classes in exception handling has some obvious advantages. For example, when the exception is thrown, an object of the type of the exception is actually created; thus it is possible to provide additional information about the exception in the form of member variables. Also, appropriate use of constructors and destructors can replace the relatively inelegant resource cleanup mechanism shown in Listing 7.

C++ Exception Classes

Visual C++ Version 4.0 provides an implementation of the exception class hierarchy, as put forward in the draft ANSI C++ standard. This hierarchy consists of the exception class and derived classes representing various conditions, such as runtime errors. The exception class and derived classes are declared in the header file stdexcpt.h. Because these classes are based on an evolving draft standard, it is possible that they will change with future releases of Visual C++.

Mixing C and C++ Exceptions

While the C compiler does not support C++ exceptions, the C++ compiler supports both C++ exceptions and the Microsoft extensions for C exceptions. Sometimes it is

necessary to mix these two in order to use the C++ exception syntax while catching Win32 structured exceptions. There are basically two methods for this: You can use an ellipsis handler, or you can use a translator function.

The Ellipsis Handler

In the termination handling example shown in Listing 7, we already made use of the *ellipsis handler*. This catchall handler, which has the form

```
catch(...)
{
}
```

can be used to catch exceptions of any type, including C exceptions. This offers a simple exception handling mechanism like the one used in Listing 7. Unfortunately, the ellipsis handler does not have any information about the actual type of the structured exception.

This should be easy, you say. (Well, I certainly said that when I first read about the differences between C and C++ exception handling.) Why not just catch an exception of type unsigned int (after all, the Microsoft Visual C++ documentation states that this is the type of C exceptions) and examine its value? Consider the program in Listing 8:

Listing 8. Failed attempt to catch C exceptions as C++ exceptions of type unsigned int

```
#include <windows.h>
#include <iostream.h>
void main(void)
{
   int x, y;
   try
   {
      x = 5;
      y = 0;
      x = x / y;
   }
   catch (unsigned int e)
   {
       if (e == EXCEPTION_INT_DIVIDE_BY_ZERO)
```

```
        {
            cout << "Division by zero.\n";
        }
        else throw;
    }
}
```

Alas, this elegant solution is no solution at all. C exceptions can only be caught by an ellipsis handler. But not all is lost just yet; could we not simply use the GetExceptionCode function in the C++ catch block and obtain the structured exception type? For example, consider the program in Listing 9.

Listing 9. C++ exception handlers cannot call GetExceptionCode

```
#include <windows.h>
#include <iostream.h>
void main(void)
{
    int x, y;
    try
    {
        x = 5;
        y = 0;
        x = x / y;
    }
    catch (...)
    {
        // The following line results in a compiler error
        if (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)
        {
            cout << "Division by zero.\n";
        }
        else throw;
    }
}
```

As they say, nice try but no cigar. The function GetExceptionCode is implemented as an intrinsic function and can only be called as part of the filter expression in a C

__except statement. It seems that some other mechanism is necessary to differentiate between C exceptions in C++ code.

There is yet another possible solution. We could create a C exception handler to catch all C exceptions and throw a C++ exception of type unsigned int with the value of the C exception code. An example program for this is shown in Listing 10.

Listing 10. Raising C++ exceptions in a C exception filter

```
#include <windows.h>
#include <iostream.h>
int divide(int x, int y)
{
   try
   {
      x = x / y;
   }
   catch(unsigned int e)
   {
      cout << "Inside C++ exception.\n";
       if (e == EXCEPTION_INT_DIVIDE_BY_ZERO)
      {
         cout << "Division by zero.\n";
      }
      else throw;
   }
   return x;
}
unsigned int catchall(unsigned int code)
{
   cout << "inside catchall: " << code << '\n';
   if (code != 0xE06D7363) throw (unsigned int)code;
    return EXCEPTION_CONTINUE_SEARCH;
}
void main(void)
{
   int x, y;
```

```
    __try
    {
        x = 10;
        y = 0;
        x = divide(x, y);
    }
    __except(catchall(GetExceptionCode())) {}
}
```

This approach has but one problem. When the catchall function throws a C++ exception that is *not* handled by a C++ exception handler, it is treated as yet another C exception, resulting in another call to catchall. This would go on forever, were it not for the test for the value 0xE06D7363, which appears to be a magic value associated with C++ exceptions. But we are getting into seriously undocumented stuff here; there has to be another solution!

At this point, you might ask the obvious question: if C++ programs can use the Microsoft C exception handling mechanism, why go through this dance at all? Why not just use __try and __except and get it over with? Indeed, this is a valid solution; however, to improve code portability, you may want to use the C++ exception handling mechanism when possible, and localize and dependence on Microsoft extensions as much as possible.

Translating C Exceptions

Fortunately, the Win32 API provides a function that allows a much more elegant solution for translating a C exception into a C++ exception. The name of the function is _set_se_translator. Using this function, one can finally obtain an elegant, satisfactory solution for translating C exceptions to C++ exceptions. An example for this is shown in Listing 11.

Listing 11. Using _set_se_translator to translate C exceptions

```
#include <windows.h>
#include <iostream.h>
#include <eh.h>
int divide(int x, int y)
{
    try
    {
        x = x / y;
```

```
    }
    catch(unsigned int e)
    {
        cout << "Inside C++ exception.\n";
         if (e == EXCEPTION_INT_DIVIDE_BY_ZERO)
        {
            cout << "Division by zero.\n";
        }
        else throw;
    }
    return x;
}
void se_translator(unsigned int e, _EXCEPTION_POINTERS* p)
{
    throw (unsigned int) ;
}
void main(void)
{
    int x, y;
    _set_se_translator(se_translator);
    x = 10;
    y = 0;
    x = divide(x, y);
}
```

Summary

Win32 programmers using the C++ language must face two separate, only partially compatible exception handling mechanisms. Win32-structured exceptions are often generated by the operating system. These exceptions are not dependent on any language-specific implementation and are used to communicate a condition to the application's exception handler using a 32-bit unsigned value.

In contrast, C++ exceptions are typed expressions; the nature of the exception is often derived from the type of the object that is used when the expression is thrown.

C programs can use the __try and __except keywords (which are Microsoft extensions to the C language) to handle structured exceptions. It is possible for exception handlers

to be nested. The type of the expression is obtained by calling the GetExceptionCode function in the __except filter expression. Depending on the value of the filter expression, an exception may be handled by the exception handler, execution may continue at the point where the exception occurred, or control can be transferred to the next exception handler. An unhandled exception causes an application error.

C programs can also use termination handlers. These handlers, installed using the __try and __finally keywords, can ensure that a function which is abnormally terminated by an exception is given a chance to perform cleanup.

C++ programs use the C++ try and catch keywords to handle exceptions. The type of the exception is declared following the catch keyword. The catch keyword with an ellipsis declaration (...) can be used to catch all exceptions; one possible use of this construct is to act as a termination handler, analogous to the __finally block in C exception handling.

As C++ programs can also use C exceptions, it is possible to mix the two exception handling mechanisms. C++ programs can catch C exceptions using an ellipsis handler. Unfortunately, this method does not allow C++ programs to obtain the exception code. However, C++ programs can install an exception translator function, which can be used to translate C structured exceptions into C++ typed exceptions.

APPENDIX II

C++ TEMPLATES

INTRODUCTION

Many C++ programs use common data structures like stacks, queues and lists. A program may require a queue of customers and a queue of messages. One could easily implement a queue of customers, then take the existing code and implement a queue of messages. The program grows, and now there is a need for a queue of orders. So just take the queue of messages and convert that to a queue of orders (Copy, paste, find, replace????). Need to make some changes to the queue implementation? Not a very easy task, since the code has been duplicated in many places. Re-inventing source code is not an intelligent approach in an object oriented environment which encourages re-usability. It seems to make more sense to implement a queue that can contain any arbitrary type rather than duplicating code. How does one do that? The answer is to use type parameterization, more commonly referred to as templates.

C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T can be replaced with actual types, for example, Queue<Customers>, and C++ will generate the class Queue<Customers>. Changing the implementation of the Queue becomes relatively simple. Once the changes are implemented in the template Queue<T>, they are immediately reflected in the classes Queue<Customers>, Queue<Messages>, and Queue<Orders>.

Templates are very useful when implementing generic constructs like vectors, stacks, lists, queues which can be used with any arbitrary type. C++ templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code.

C++ provides two kinds of templates: class templates and function templates. Use function templates to write generic functions that can be used with arbitrary types. For example, one can write searching and sorting routines which can be used with any arbitrary type. The Standard Template Library generic algorithms have been implemented as function templates, and the containers have been implemented as class templates.

CLASS TEMPLATES

Implementing a class template

A class template definition looks like a regular class definition, except it is prefixed by the keyword template. For example, here is the definition of a class template for a Stack.

```
template <class T>
class Stack
{
public:
        Stack(int = 10) ;
        ~Stack() { delete [] stackPtr ; }
        int push(const T&);
        int pop(T&) ;
        int isEmpty()const { return top == -1 ; }
        int isFull() const { return top == size - 1 ; }
private:
        int size ;  // number of elements on Stack.
        int top ;
        T* stackPtr ;
} ;
```

T is a type parameter and it can be any type. For example, Stack<Token>, where Token is a user defined class. T does not have to be a class type as implied by the keyword class. For example, Stack<int> and Stack<Message*> are valid instantiations, even though int and Message* are not “classes”.

Implementing Class Template Member Functions

Implementing template member functions is somewhat different compared to the regular class member functions. The declarations and definitions of the class template member functions should all be in the same header file. The declarations and definitions need to be in the same header file. Consider the following.

//B.Htemplate <class t>class b{public: b() ; ~b() ;} ; // B.CPP#include "B.H"template <class t>b<t>::b(){}template <class t>b<t>::~b(){}/ / MAIN.CPP#include "B.H"void main(){ b<int> bi ; b <float> bf ;}

When compiling B.cpp, the compiler has both the declarations and the definitions available. At this point the compiler does not need to generate any definitions for template classes, since there are no instantiations. When the compiler compiles main.cpp, there are two instantiations: template class B<int> and B<float>. At this point the compiler has the declarations but no definitions!

While implementing class template member functions, the definitions are prefixed by the keyword template. Here is the complete implementation of class template Stack:

```
//stack.h
#pragma once
template <class T>
class Stack
{
public:
        Stack(int = 10) ;
        ~Stack() { delete [] stackPtr ; }
        int push(const T&);
        int pop(T&) ;  // pop an element off the stack
        int isEmpty()const { return top == -1 ; }
        int isFull() const { return top == size - 1 ; }
private:
        int size ;  // Number of elements on Stack
        int top ;
        T* stackPtr ;
} ;
//constructor with the default size 10
template <class T>
Stack<T>::Stack(int s)
{
```

```
        size = s > 0 && s < 1000 ? s : 10 ;
        top = -1 ;  // initialize stack
        stackPtr = new T[size] ;
}
 // push an element onto the Stack
template <class T>
int Stack<T>::push(const T& item)
{
        if (!isFull())
        {
                stackPtr[++top] = item ;
                return 1 ;  // push successful
        }
        return 0 ;  // push unsuccessful
}

// pop an element off the Stack
template <class T>
int Stack<T>::pop(T& popValue)
{
        if (!isEmpty())
        {
                popValue = stackPtr[top—] ;
                return 1 ;  // pop successful
        }
        return 0 ;  // pop unsuccessful
}
```

Using a Class Template

Using a class template is easy. Create the required classes by plugging in the actual type for the type parameters. This process is commonly known as “Instantiating a class”. Here is a sample driver class that uses the Stack class template.

```
#include <iostream>
#include “stack.h”
using namespace std ;
```

```
void main()
{
        typedef Stack<float> FloatStack ;
        typedef Stack<int> IntStack ;
        FloatStack fs(5) ;
        float f = 1.1 ;
        cout << "Pushing elements onto fs" << endl ;
        while (fs.push(f))
        {
                cout << f << ' ' ;
                f += 1.1 ;
        }
        cout << endl << "Stack Full." << endl
        << endl << "Popping elements from fs" << endl ;
        while (fs.pop(f))
                cout << f << ' ' ;
        cout << endl << "Stack Empty" << endl ;
        cout << endl ;
        IntStack is ;
        int i = 1.1 ;
        cout << "Pushing elements onto is" << endl ;
        while (is.push(i))
        {
                cout << i << ' ' ;
                i += 1 ;
        }
        cout << endl << "Stack Full" << endl
        << endl << "Popping elements from is" << endl ;
        while (is.pop(i))
                        cout << i << ' ' ;
        cout << endl << "Stack Empty" << endl ;
}
```

Program Output

```
Pushing elements onto fs
1.1 2.2 3.3 4.4 5.5
Stack Full.
Popping elements from fs
5.5 4.4 3.3 2.2 1.1
Stack Empty
Pushing elements onto is
1 2 3 4 5 6 7 8 9 10
Stack Full
Popping elements from is
10 9 8 7 6 5 4 3 2 1
Stack Empty
```

In the above example we defined a class template Stack. In the driver program we instantiated a Stack of float (FloatStack) and a Stack of int(IntStack). Once the template classes are instantiated you can instantiate objects of that type (for example, fs and is.)

A good programming practice is using typedef while instantiating template classes. Then throughout the program, one can use the typedef name. There are two advantages:

- **typedef's** are very useful when "templates of templates" come into usage. For example, when instantiating an STL vector of int's, you could use:
- typedef vector<int, allocator<int> > INTVECTOR ;
- If the template definition changes, simply change the typedef definition. For example, currently the definition of template class vector requires a second parameter.
  - typedef vector<int, allocator<int> > INTVECTOR ;
  - INTVECTOR vi1 ;

    In a future version, the second parameter may not be required, for example,

    typedef vector<int> INTVECTOR ;

    INTVECTOR vi1 ;

Imagine how many changes would be required if there was no **typedef**!

Function Templates

To perform identical operations for each type of data compactly and conveniently, use function templates. You can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically

instantiates separate object code functions to handle each type of call appropriately. The STL algorithms are implemented as function templates.

Implementing Template Functions

Function templates are implemented like regular functions, except they are prefixed with the keyword template. Here is a sample with a function template.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
```

Using Template Functions

Using function templates is very easy: just use them like regular functions. When the compiler sees an instantiation of the function template, for example: the call max (10, 15) in function main, the compiler generates a function max(int, int). Similarly the compiler generates definitions for max(char, char) and max(float, float) in this case.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
void main()
{
        cout <<" max(10, 15) = " << max(10, 15) << endl ;
        cout <<" max('k', 's') = " << max('k', 's') << endl ;
        cout <<" max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
}
```

Program Output

max(10, 15) = 15

max('k', 's') = s

max(10.1, 15.2) = 15.2

Template Instantiation

When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.

- A class generated from a class template is called a generated class.
- A function generated from a function template is called a generated function.
- A static data member generated from a static data member template is called a generated static data member.

The compiler generates a class, function or static data members from a template when it sees an implicit instantiation or an explicit instantiation of the template.

1. Consider the following sample. This is an example of implicit instantiation of a class template.

```
template <class T>
class Z
{
public:
Z() {} ;
~Z() {} ;
void f(){} ;
void g(){} ;
} ;

int main()
{
Z<int> zi ;   //implicit instantiation generates class Z<int>
Z<float> zf ; //implicit instantiation generates class Z<float>
return 0 ;
}
```

18. Consider the following sample. This sample uses the template class members Z<T>::f() and Z<T>::g().

```
template <class T>
```

```
20. class Z
21. {
22. public:
23. Z() {} ;
24. ~Z() {} ;
25. void f(){} ;
26. void g(){} ;
27. } ;
28.
29. int main()
30. {
31. Z<int> zi ; //implicit instantiation generates class Z<int>
32. zi.f() ;    //and generates function Z<int>::f()
33. Z<float> zf ; //implicit instantiation generates class Z<float>
34. zf.g() ;      //and generates function Z<float>::g()
35. return 0 ;
36. }
```

This time in addition to the generating classes Z<int> and Z<float>, with constructors and destructors, the compiler also generates definitions for Z<int>::f() and Z<float>::g(). The compiler does not generate definitions for functions, nonvirtual member functions, class or member class that does not require instantiation. In this example, the compiler did not generate any definitions for Z<int>::g() and Z<float>::f(), since they were not required.

37. Consider the following sample. This is an example of explicit instantiation of a class template.

```
38. template <class T>
39. class Z
40. {
41. public:
42. Z() {} ;
43. ~Z() {} ;
44. void f(){} ;
45. void g(){} ;
46. } ;
47.
48. int main()
```

```
{
template class Z<int> ; //explicit instantiation of class Z<int>
template class Z<float> ; //explicit instantiation of
                          //class Z<float>
return 0 ;
}
```

Consider the following sample. Will the compiler generate any classes in this case? The answer is NO.

```
template <class T>
class Z
{
 public:
Z() {} ;
~Z() {} ;
void f(){} ;
void g(){} ;
} ;

int main()
{
Z<int>* p_zi ; //instantiation of class Z<int> not required
Z<float>* p_zf ; //instantiation of class Z<float> not required
return 0 ;
}
```

This time the compiler does not generate any definitions! There is no need for any definitions. It is similar to declaring a pointer to an undefined class or struct.

Consider the following sample. This is an example of implicit instantiation of a function template.

```
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
    return a > b ? a : b ;
}
void main()
```

```
80. {
81. int I ;
82. I = max(10, 15) ; //implicit instantiation of max(int, int)
83. char c ;
84. c = max('k', 's') ; //implicit instantiation of max(char, char)
85. }
```

In this case the compiler generates functions max(int, int) and max(char, char). The compiler generates definitions using the template function max.

```
86. Consider the following sample. This is an example of explicit instantiation of a
    function template.
87. template <class T>
88. void Test(T r_t)
89. {
90. }
91.
92. int main()
93. {
94. //explicit instantiation of Test(int)
95.     template void Test<int>(int) ;
96.     return 0 ;
97. }
```

Note: Visual C++ 5.0 does not support this syntax currently. The above sample causes compiler error C1001.

Class Template Specialization

In some cases, it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization. The following example defines a template class specialization for template class stream.

```
#include <iostream>
using namespace std ;
template <class T>
class stream
{
public:
```

```
void f() { cout << "stream<T>::f()"<< endl ;}
} ;
template <>
class stream<char>
{
public:
void f() { cout << "stream<char>::f()"<< endl ;}
} ;
int main()
{
        stream<int> si ;
        stream<char> sc ;
        si.f() ;
        sc.f() ;
        return 0 ;
}
```

Program Output

```
stream<T>::f()
stream<char>::f()
```

In the above example, stream<char> is used as the definition of streams of chars; other streams will be handled by the template class generated from the class template.

Template Class Partial Specialization

You may want to generate a specialization of the class for just one parameter, for example

```
//base template class
template<typename T1, typename T2>
class X
{
} ;

//partial specialization
template<typename T1>
class X<T1, int>
```

```
{
} ; //C2989 here
int main()
{
        // generates an instantiation from the base template
        X<char, char> xcc ;
        //generates an instantiation from the partial specialization
X<char, int> xii ;
        return 0 ;
}
```

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list.

Note: Visual C++ 5.0 does not support template class partial specialization. The above sample causes compiler error C2989: template class has already been defined as a non-template class.

Template Function Specialization

In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization. The following example demonstrates a situation where overriding the template generated code would be necessary:

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements of type T, where T is a
//class or data type for which operator> is defined.
template <class T>
T max(T a, T b)
{
   return a > b ? a : b ;
}
int main()
{
   cout << "max(10, 15) =" << max(10, 15) << endl ;
   cout << "max('k', 's') =" << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) =" << max(10.1, 15.2) << endl ;
```

```
    cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin", "Jasmine") <<
endl ;
    return 0 ;
}
```

Program Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Aladdin", "Jasmine") = Aladdin
```

Not quite the expected results! Why did that happen? The function call max("Aladdin", "Jasmine") causes the compiler to generate code for max(char*, char*), which compares the addresses of the strings! To correct special cases like these or to provide more efficient implementations for certain types, one can use template specializations. The above example can be rewritten with specialization as follows:

```
#include <iostream>
#include <cstring>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
    return a > b ? a : b ;
}
// Specialization of max for char*
template <>
char* max(char* a, char* b)
{
    return strcmp(a, b) > 0 ? a : b ;
}
int main()
{
```

```
    cout << "max(10, 15) = " << max(10, 15) << endl ;
    cout << "max('k', 's') = " << max('k', 's') << endl ;
    cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
    cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin", "Jasmine") << endl ;
    return 0 ;
}
```

Program Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Aladdin", "Jasmine") = Jasmine
```

Template Parameters

1. C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T can be replaced with actual types, for example, Queue<Customers>, and C++ will generate the class Queue<Customers>. For example,
2. template <class T>
3. class Stack
4. {
5. } ;

Here T is a template parameter, also referred to as type-parameter.

6. C++ allows you to specify a default template parameter, so the definition could now look like:
7. template <class T = float, int elements = 100> Stack { ....} ;

Then a declaration such as

Stack<> mostRecentSalesFigures ;

would instantiate (at compile time) a 100 element Stack template class named mostRecentSalesFigures of float values; this template class would be of type Stack<float, 100>.

Note, C++ also allows non-type template parameters. In this case, template class Stack has an int as a non-type parameter.

If you specify a default template parameter for any formal parameter, the rules are the same as for functions and default parameters. Once a default parameter is declared all subsequent parameters must have defaults.

8. Default arguments cannot be specified in a declaration or a definition of a specialization. For example,

```
template <class T, int size>
class Stack
{
} ;

//error C2989: 'Stack<int,10>' : template class has already been
//defined as a non-template class
template <class T, int size = 10>
class Stack<int, 10>
{
} ;

int main()
{
Stack<float,10> si ;
return 0 ;
}
```

26. A type-parameter defines its identifier to be a type-name in the scope of the template declaration, and canot be re-declared within its scope (including nested scopes). For example,

```
template <class T, int size>
class Stack
{
int T ; //error type-parameter re-defined.
void f()
{
char T ; //error type-parameter re-defined.
    }
} ;

class A {} ;
```

```
38. int main()
39. {
40. Stack<A,10> si ;
41. return 0 ;
42. }
```

Note: VC++ 5.0 or SP1 compiles this sample without any errors. It does not flag the re-definition of type-parameter as an error.

43. The value of a non-type-parameter cannot be assigned to or have its value changed. For example,

```
44. template <class T, int size>
45. class Stack
46. {
47. void f()
48.     {
49. //error C2105: '++' needs l-value
50. size++ ; //error change of template argument value
51. }
52. } ;
53.
54.
55. int main()
56. {
57. Stack<double,10> si ;
58. return 0 ;
59. }
```

60. A template-parameter that could be interpreted as either a parameter-declaration or a type-parameter, is taken as a type-parameter. For example,

```
61. class T {} ;
62. int i ;
63.
64. template <class T, T i>
65. void f(T t)
66. {
67. T t1 = i ; //template arguments T and i
68. ::T t2 = ::i ; //globals T and i
```

```
69. }
70.
71.
72.
73. int main()
74. {
75. f('s') ; //C2783 here
76. return 0 ;
77. }
```

Note: Compiling the above sample using VC++ 5.0 and SP1 causes compiler error C2783: could not deduce template argument for 'i'. To workaround the problem, replace the call to f('s') with f<char, 's'>('s').

```
class T {} ;
int i ;

template <class T, T i>
void f(T t)
{
   T t1 = i ; //template arguments T and i
::T t2 = ::i ; //globals T and i
}

int main()
{
   f<char, 's'>('s') ; //workaround
   return 0 ;
}
78. A non-type template parameter cannot be of floating type. For example,
79. template <double d> class X ; //error C2079: 'xd' uses
80. //undefined class 'X<1.e66>'
81. //template <double* pd> class X ; //ok
82. //template <double& rd> class X ; //ok
83.
84. int main()
85. {
```

```
86. X<1.0> xd ;
87. return 0 ;
88. }
```

Static Members and Variables

1. Each template class or function generated from a template has its own copies of any static variables or members.
2. Each instantiation of a function template has it's own copy of any static variables defined within the scope of the function. For example,

```
3. template <class T>
4. class X
5. {
6.     public:
7.     static T s ;
8. } ;
9.
10. int main()
11. {
12.     X<int> xi ;
13.         X<char*> xc ;
14. }
```

Here X<int> has a static data member s of type int and X<char*> has a static data member s of type char*.

15. Static members are defined as follows.

```
16. #include <iostream>
17. using namespace std ;
18.
19. template <class T>
20. class X
21. {
22.     public:
23.     static T s ;
24. } ;
25.
26. template <class T> T X<T>::s = 0 ;
27. template <> int X<int>::s = 3 ;
```

```
28. template <> char* X<char*>::s = "Hello" ;
29.
30. int main()
31. {
32.     X<int> xi ;
33.     cout << "xi.s = " << xi.s << endl ;
34.
35.       X<char*> xc ;
36.     cout << "xc.s = " << xc.s << endl ;
37.
38.     return 0 ;
39. }
```

Program Output

```
xi.s = 10
xc.s = Hello
```

40. Each instantiation of a function template has it's own copy of the static variable. For example,

```
41. #include <iostream>
42. using namespace std ;
43.
44. template <class T>
45. void f(T t)
46. {
47.     static T s  = 0;
48.     s = t ;
49.     cout << "s = " << s << endl ;
50. }
51.
52. int main()
53. {
54.     f(10) ;
55.     f("Hello") ;
56.
57.     return 0 ;
58. }
```

Program Output

s = 10
s = Hello

Here f<int>(int) has a static variable s of type int, and f<char*>(char*) has a static variable s of type char*.

Templates and Friends

Friendship can be established between a class template and a global function, a member function of another class (possibly a template class), or even an entire class (possible template class). The table below lists the results of declaring different kinds of friends of a class.

Class Template	Friend declaration in class template X	Results of giving friendship
template class <T> class X	freind void f1();	makes f1() a friend of all instantiations of template X. For example, f1() is a friend of X<int>, X<A>, and X<Y>.
template class <T> class X	friend void f2(X<T>&);	For a particular type T for example, float, makes f2(X<float>&) a friend of class X<float> only. f2(x<float>&) cannot be a friend of class X<A>.
template class <T> class X	friend A::f4(); // A is a user defined class with a member function f4();	makes A::f4() a friend of all instantiations of template X. For example, A::f4() is a friend of X<int>, X<A>, and X<Y>.
template class <T> class X	friend C<T>::f5(X<T>&);//C is a class template with a member function f5	For a particular type T for example, float, makes C<float>::f5(X<float>&) a friend of class X<float> only. C<float>::f5(x<float>&) cannot be a friend of class X<A>.
template class <T> class X	friend class Y;	makes every member function of class Y a friend of every template class produced from the class template X.
template class <T> class X	friend class Z<T>;	when a template class is instantiated with a particular type T, such as a float, all members of class Z<float> become friends of template class X<float>.

INDEX

O

P

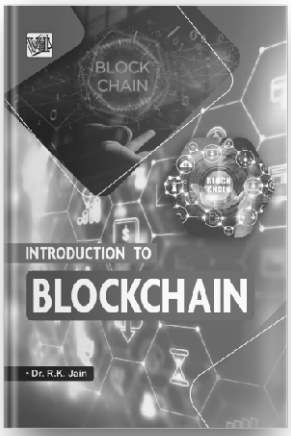
R

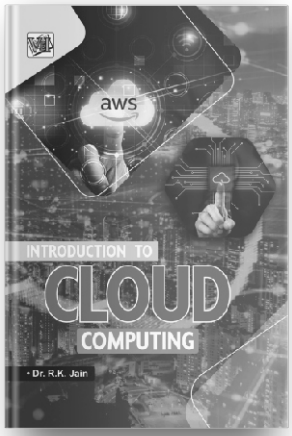
S

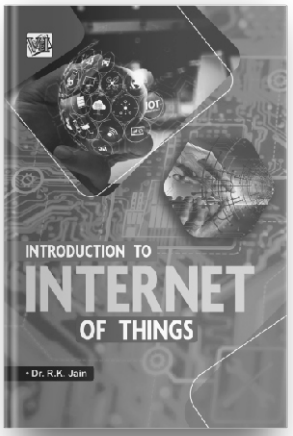
T

SPECIAL BONUS!

Want These 3 Bonus Books for free?







Get FREE, unlimited access to these and all of our new books by joining our community!

SCAN w/ your camera TO JOIN!

OR Visit

freebie.kartbucket.com