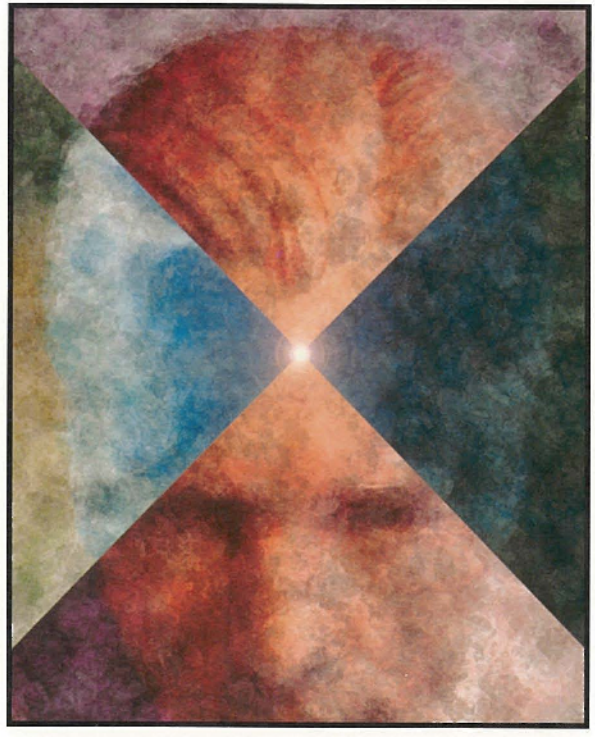
Designing and Using ActiveX Controls

CD-ROM INCLUDED



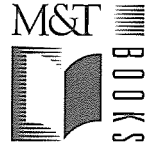
Tom Armstrong

- Implement ActiveX controls in complex program development
- Build your own controls using Microsoft's COM and SDK's
- Make your controls Internet-aware for active Web sites

M&T BOOKS

Designing and Using ActiveX Controls

Tom Armstrong



HENRY HOLT & COMPANY, INC.
NEW YORK

M&T Books
A Division of MIS:Press, Inc.
A Subsidiary of Henry Holt and Company, Inc.
115 West 18th Street
New York, New York 10011
http://www.mispress.com

First Edition—1997

ISBN 1-55851-503-8

10 9 8 7 6 5 4 3 2 1

Associate Publisher: *Paul Farrell*

Executive Editor: *Cary Sullivan*
Editor: *Andrew Neusner*
Copy Edit Manager: *Shari Chappell*
Production Editor: *Anthony Washington*
Technical Editor: *Mark Bramer*
Copy Editor: *Betsy Hardinger*

Contents-in-Brief

Dedication

To Nicole, Jessica, and Eric. Your love and support empower me to achieve things I never dreamed possible.

Contents

Acknowledgments

Acknowledgments for Designing and Using ActiveX Controls

When the M&T gang decided that a new edition of *Designing and Using OLE Custom Controls* would be a good idea, I jumped at the chance. I said to myself, "a second edition will be easy." Well, it wasn't. In many ways, writing a second edition is harder than writing the first. With the software development industry changing so rapidly, second editions are nearly total rewrites, and rewriting is harder than just writing, at least for me. It was an arduous process, and I couldn't have done it without a lot of help. Thanks to my editor, Andy Neusner, for keeping me on schedule but being flexible when emergencies came up. Thanks to Anthony Washington for laying out these pages, and to Betsy Hardinger, who did a great job of transforming my prose into something a real writer would produce.

I'm most thankful for my wife, Nicole. She again allowed me to spend part (actually most) of my evenings writing. She's just about finished with dental school, so her evenings are busy, too. Thanks also to my children, Jessica and Eric, who, when I slipped into a TV-zone and spent too much time on the couch, reminded me to get to work on the book. I love you all dearly.

Thanks to my technical editor, Mark Bramer. Mark and I go way back. All the way back to our first job together when I interviewed him and didn't want to hire him. What a mistake. Smarter heads prevailed and Mark and I have been working together ever since. Mark is a talented developer, editor, and writer. One of these days, Mark, we'll co-author a book.

Thanks to Marc Ritterbusch, who has finally moved over to the Windows development side of our group (better late than never!), for designing the icons for the Chapter 10 tree view example. One of these days, Marc, you're going to catch something when you go fishing.

Thanks to Richard "Doogie" Clark. Doogie read through each chapter and provided a lot of valuable feedback. In particular, he kept me honest when I started getting lazy. Thanks also to Bruce Allen. Bruce, I'm going to need your help on my next book now that you're an expert on Internet Explorer and connection points. Don't forget.

Thanks to the students who helped work out some of the presentation bugs in the examples. Thanks to the members of the AWD development team, Chuck Reeves and Mark Clobes in particular, who spent the week of October 21 learning about COM, OLE, and ActiveX controls. Thanks also to Dan Weiss, President of Step 1 Training, where I teach an occasional ActiveX/MFC class.

Thanks to all the readers of the first edition, many of whom have provided valuable feedback that has been incorporated into this edition. Specifically, thanks to Stuart Bessler, John Wood, Rick Anderson, and Bob Wilkins. Bob provided a great example of using ActiveX controls within another ActiveX control (it's included on the CD-ROM).

Acknowledgments for Designing and Using OLE Custom Controls

I have to first thank my editor, Judy Brief, who took a chance on a first-time writer by responding to my proposal letter in a record three business hours. Thanks to the whole team at M&T Books: Stephanie Doyle who designed these pages (and spent her weekends doing it) and Karen Tongish who transformed my doggerel into something readable. Thanks also to my technical editor, John Elsbree of Microsoft.

I am thankful for the love and understanding provided by my beautiful wife, Nicole, and my wonderful children, Jessica and Eric. They endured my absence on the weekends, in the evenings, and even on the summer vacation while I pounded away on these pages. I promise to love and support each of them in whatever they choose to pursue.

To the members of the EnCorr development team: To Rob Alumbaugh, a brilliant young developer who reviewed some of the chapters and kept me from getting too "low-level." Thanks Rob. Thanks to Mark Bramer for his help with the design of the clock control and for providing his considerable editing prowess even when he had only a few hours to review a chapter. Thanks to Jim Crespino, who had many ideas and suggestions regarding the control examples, particularly the pipe control of Chapter 11. Thanks to Steve and Doris Stava for reviewing some of the chapters, providing control ideas, and the continual encouragement. And to Roy Lambright, Bob Rench, and Marc Ritterbusch: Thanks for your friendship and humor. When are you going to start developing for a real operating system and forget that OS/2 stuff?

Thanks to various individuals that I've had the pleasure to attend school with or work with in my career who have provided opportunities; stimulated intellectual growth through discussion; provided moral support, friendship, and encouragement; or a combination: Jim Phelan, Jim Kurtenbach, David Bridges, Chuck Reeves, Devin Sherry, Phil Brennaman, Steve Gray, Steve Luke, Danny Hughes, Mike Hudgins, and David Cloud.

To my mother Maureen, my late father Tom Sr., my mother-in-law Millie Koschmeder, my late father-in-law Fred Koschmeder, and my grandparents Cloyd and Josephine Jenkins: Thank you for the unconditional love and support that you have always provided.

To those teachers who have taught me more than the requisite material, in particular Graham Glass, an instructor at the University of Texas at Dallas: Graham you were the one teacher who affected my life profoundly. Your enthusiasm, voracious thirst for knowledge, and general love of life inspired me to approach learning (and life) in a different way. Thank you.

And finally, to that which inspires each of us to push beyond our sphere of contentment. We all have the ability, creativity, and stamina to change the world, but we must endeavor to do it. In the words of Gandhi: "We must become the change we seek in the world."

Preface

This book is for those software developers who want to participate in the burgeoning field of component software development. Software components, specifically those based on Microsoft's Component Object Model (COM), are having a broad impact on the development of Windows-based software. In order for developers to remain productive and competitive, it is imperative that they understand and apply this new technology.

Today, the Internet is having a major impact on the software development industry. To maintain its lead in PC-based environments, Microsoft has radically altered its approach to software development. Microsoft has embraced the new Internet-based development technologies and is moving rapidly to provide developers with state-of-the-art tools and techniques. ActiveX is at the center of this movement. This book covers one of the most important new technologies: ActiveX controls.

What Is Expected from the Reader

To build and understand the sample applications, the reader is expected to have a programming background that includes work in Windows software development, using C, C++, or visual tools such as Visual Basic. This book is not a complete tutorial on any of these languages or tools, but rather is about how to use Visual C++, MFC, COM, OLE, and ActiveX to build robust software components, particularly ActiveX controls. An understanding of object-oriented techniques is valuable but not necessary.

The reader is not expected to understand Microsoft's Component Object Model or ActiveX, but this book cannot cover every aspect of COM. Only those areas important to development of component software, automation, and specifically ActiveX controls are discussed. Other areas of COM will be discussed only as they pertain to the understanding of our primary topic: ActiveX controls.

Hardware and Software Required for the Sample Application

The examples were developed on a 90-MHz Pentium with 24 megabytes of memory running Windows NT 4.0. The software tools required to build and test the samples include: Microsoft's Visual C++ version 4.0 or higher running on Windows 95 or Windows NT 3.51 or higher. The screen shots are of Visual C++ 4.2.

Some control examples use features of Microsoft's ActiveX SDK and the August 1996 Win32 SDK, but by the time you read this, full support for these SDKs will be available in Visual C++ 5.0. If you have any questions as to the requirements to build the sample controls, check out my Web site at `http://www.widgetware.com`

Comments and Bug Reports

I welcome and encourage comments, suggestions, and bug reports at the email address given at the end of this Preface. The Visual C++ compiler is updated three times a year, and as it is changes, I update the example programs and controls. You can contact me via email or through my Web site URL. The site contains any updated examples, FAQs, pointers to other OLE/ActiveX sites, discussions, and other material concerning COM, MFC, and ActiveX technology.

email: `toma@sky.net or tom@widgetware.com`

URL: `http://www.sky.net/~toma/ or http://www.widgetware.com`

Introduction

Software development is becoming more and more complex. New paradigms are needed to decompose this complexity so that the process of developing software can be improved. Large software applications must be broken down into smaller, more manageable pieces. During the last few years, object-oriented languages have helped reduce this complexity explosion. The C++ language in particular has garnered a lot of industry support and is currently enjoying healthy growth in many areas of software development. Smalltalk has become a viable development language in several environments as well, and Java, with the popularity of the Web driving its advance, is quickly becoming an important development language. However, even with these advancements in development paradigms and languages, software development is still a very complex task.

Examples of where software complexity has overwhelmed the development process are easy to find. Severe problems with the software that controlled the baggage system at Denver's new airport caused months of delays and cost billions of dollars. Microsoft's Windows 95 was delayed by nearly two years because (among other reasons) of the complexities of developing a robust operating system with more than 4 million lines of source code. While these examples are extreme, they illustrate the problems that exist in the software development industry today.

Object-oriented paradigms and their primary implementation language, C++, cannot solve this complexity explosion alone. Additional technologies and techniques are required. Microsoft's Component Object Model (COM), OLE, and ActiveX are the technologies that we will explore in this book. These technologies are having a major impact on the development of Windows software, specifically in the area of component software development.

Component Software

The concept of *component software* is not new. Software developers have long hoped for a technology that would enable them to assemble software in a manner similar to that used by hardware engineers. For more than 40 years, computer hardware designers have constructed complex hardware systems by combining off-the-shelf hardware components. These components, or integrated circuits (ICs), can be assembled in endless patterns to make practically anything (electronic) imaginable. When viewed independently, RAM, ASICs, JK flip-flops, and other chips perform rather insignificant functions, but when combined with other IC components, complex hardware systems can be built.

Each IC can be treated, from a design perspective, as a *black box*. The user provides the black box with a set of inputs, and depending on the behavior of the black box, a specific result will be provided as the output. The designer needs absolutely no understanding of *how* the black box performs its functions; he only needs an understanding of the input and output behavior. The inputs and outputs are stringently defined, and this rigid structure allows the easy combining of hundreds of IC components, working together to define complex hardware systems.

Component software is an effort to apply the hardware paradigm to software. This approach would provide software *ICs* that could be assembled in various configurations to quickly produce robust software systems. C++ and other OOP languages were supposed to provide this capability, but, for various reasons, they have not achieved this goal. Microsoft's Component Object Model and its companions, OLE and ActiveX, provide the technology to make component software a reality.

Twenty years ago, computer hardware was very expensive while software was relatively cheap. Today the reverse is true. The reason: Massive reuse of well-specified, discrete hardware components. This leap in productivity is currently occurring in software development, with Visual Basic custom controls providing the impetus.

Custom Controls

Visual Basic finally provided a platform that supported component development, not from the aspect of building software components (ICs), but by providing the *breadboard* on which to assemble the various components into usable applications. If we continue the hardware analogy, Visual Basic custom controls (VBXes) are the ICs and any Visual Basic source code is the *bus* or wires that connect these discrete components.

Custom controls provide specific, well-defined functionality. They do not provide all the advanced features of true OOP languages, such as inheritance and polymorphism, but this simplicity may explain their success. Custom controls cannot function independently and can only provide value when coupled with other custom controls by a controlling entity (usually Visual Basic). This controlling entity is called a *container* in OLE parlance; it provides the *glue* that ties the components together. It also provides the bus over which the controls can communicate.

Discrete hardware components and custom controls share many characteristics. They have a focused purpose, documented behavior, and a well-defined interface (the inputs and outputs). The user of a control only needs to understand its behavior and how to affect this behavior. Examples include simple entry fields that validate dates as they are entered or more complex controls that encapsulate the functionality of a complete editor. Industry-specific controls are also important. A vendor in the health care industry may provide a custom control that implements an interface to a medical device. Because the control hides any proprietary logic, he can freely distribute the control to end users. The examples and uses are myriad, as the healthy market for third-party controls demonstrates.

ActiveX Controls

While Visual Basic custom controls provided the stimulus that eventually validated component development, its VBX architecture has two major problems. The VBX architecture is inextricably tied to the Windows 16-bit environment, and Microsoft does not publish the details of how VBXes can be used within other applications or tools. Microsoft's Component Object Model, OLE, and ActiveX standards now provide a solid foundation on which to build software components. ActiveX controls have replaced the VBX and have added crucial functionality in the process. ActiveX controls can expect a much larger market, relative to that provided to VBX developers today, because ActiveX controls are supported by many more applications and software development tools. ActiveX controls have also been enlisted to become the central software element in Microsoft's new Web-based strategy.

ActiveX Controls and the Internet

On March 6, 1996, Microsoft unveiled its new ActiveX development strategy. Central to this new strategy is the ActiveX control: ActiveX controls can now be embedded in Web-based pages and accessed by browsers such as Internet Explorer. The component-based software revolution has now come to the Internet, and ActiveX controls are a major part of this new environment.

Automation

Another area of COM/OLE that enhances the viability of component software development is automation. Automation allows Windows applications to *expose*, or make available, their functionality to other Windows applications. In this way, applications like Microsoft Word become software components that can be used by other smaller, larger, or more complex applications. For example, say a developer is faced with the prospect of spending a few man-years developing a word processing package for his application. His users indicate their need for this capability and want him to "make it something like Word" if he can. Today, thanks to COM and OLE, he can choose either to spend the man-years developing a word processing package or to incorporate Microsoft Word directly into his existing application. His users gain the use of something familiar, and the developer can add Microsoft Word functionality with a minimum of work (measured in man-weeks). COM, OLE, and ActiveX, with their component and application integration technologies, make this possible.

Visual C++ and the MFC Libraries

Visual C++ and its Microsoft Foundation Class (MFC) libraries are today's most advanced development tools available for Windows. The MFC libraries provide an *application framework* that removes much of the tedious work involved with developing Windows and COM-based applications. While a lot of the tedium is removed, there is still a lot of complexity left behind. In particular, the incorporation of COM, OLE, and

ActiveX support within the MFC libraries adds another dimension that developers must assimilate. Nevertheless, Visual C++ and the MFC libraries help significantly in the development of COM-based components, as this book demonstrates.

The Chapters

The chapters in this book are probably best read in succession, although the first three chapters can be skimmed if you are already familiar with C++, Visual C++, and the MFC libraries. Chapter 1 discusses the issues facing software developers today: how to achieve reuse with today's software tools, why object-oriented development languages haven't solved more development problems, and the concept of component software. Chapter 2 provides an overview of the C++ language and how it can be used to effectively build software components. Chapter 3 introduces the Visual C++ development environment and details the workings of the Microsoft Foundation Class framework.

Chapters 4 focuses on Microsoft's Component Object Model and how this system-level technology enables the creation of robust software components. Chapter 5 builds on Chapter 4 by detailing the MFC implementation of COM and ActiveX. Chapter 6 covers the use of these technologies by wrapping C++ classes with automation.

The last half of the book focuses exclusively on the development of ActiveX controls. First the architecture is examined, and then various ActiveX control types are explored. Chapters 8, 9, 10, 11, and 12 each detail the development of a specific type of ActiveX control. A graphical clock control is developed in Chapter 9. Chapter 10 covers development of controls that subclass standard Windows controls and the new Windows 95 common controls. A nonvisual control that encapsulates the services provided by the Win32 named pipes API is developed in Chapter 11. Chapter 12 details what is required to develop and implement an Internet-aware control. And finally, Chapter 13 focuses on answering the most frequently asked questions concerning automation and ActiveX control development.

Chapter 1

Component-Based Development, the Web, and ActiveX

Component-based software development is changing the way Windows applications are developed. In this chapter we will look at what component software is and why it is having a tremendous impact on the software development industry. We will discuss the definitions of software objects and components as well as the benefits they can provide the application developer. We will also describe Microsoft's *Component Object Model* (COM), the primary technology we will use to construct and connect these components.

Within the past year, the popularity of the Internet has caused a major shift in the development of software. The way software is developed, deployed, and supported has been radically altered by this entity, called the Web. Component software—from ActiveX controls and Java applets to Netscape plug-ins—abounds in Web-based environments.

Web technologies are also changing the corporate software environment. Corporate *intranets*, a much larger and more lucrative market than the commercial Internet, are quickly moving to Web-based development tools and technologies. Microsoft's ActiveX is one of the most important new Web-based technologies.

ActiveX is a broad term that covers Microsoft's Web-based strategy. It covers a large number of applications, tools, and technologies, all of which use COM heavily in their implementation. For now, ActiveX can be thought of as the underpinning of Microsoft's Web-based technologies. In reality, the term *ActiveX* has basically replaced one that we all are familiar with: OLE. The term *OLE* now connotes a small subset of Microsoft's component technologies—specifically, the old compound document technology that has been available for several years. In this first chapter, we'll talk about COM and ActiveX in the context of component-based development.

This chapter is intended to whet your appetite by showing you the benefits that component-based software can provide the software industry and explaining why you, the software developer, should begin to study and adopt these methods. Trust me—component software is going to make our jobs much more fun.

If you don't really care how we got here or why and want to get your hands dirty, jump ahead to Chapter 2. But promise me that when you get time, you'll come back and read this chapter, because it explains why software components are going to change the world.

The Changing Development Landscape

Application development techniques have always changed rapidly, but within the last few years, for various reasons, profound changes have occurred. Users are more sophisticated and demanding, and business in general has become more competitive. And now, with the enormous growth in the popularity of the Internet, software is developed and deployed at a frenetic pace. Effective development and deployment of software systems is essential to the survival of most businesses. Only a few years ago, information technology was viewed as a back office requirement that meant little to the bottom line. Today, the opposite is true.

With the realization that information technology is crucial to the competitiveness of any business entity, there has been an increasing demand to develop and deploy mission-critical applications quickly and effectively. Business environments change more rapidly than they did only a few years ago, and competitiveness is difficult to maintain in this fast changing environment. Many books have been written on the information age, so I will not expound on it here. Suffice it to say that to be competitive, companies, business units, teams, and individuals must be willing to change and adapt.

In the Introduction, I described the concept of component software development. With its ability to support third-party, custom controls, the Visual Basic development environment has dramatically changed Windows software development. The practice of combining discrete software components into solid Windows applications has also increased the productivity of many software developers. Broad reuse of software components is one reason for this increase in productivity. Software reuse is of paramount importance if we, as developers, are to advance the state of software development. And now, with the addition of Web-based technologies, the job of making small, robust, and distributable components has become much easier.

Reuse Is What Counts

It's a waste of time to build various components if they are never reused. If a similar application needs to be built and if the components we've constructed are not generic enough or are not carefully delineated, they cannot be reused effectively. If there is no reuse, there is no saving of time in either the development or the building of other applications. Reuse is the primary focus of this book and the goal of component builders everywhere. Without it, component building would provide little benefit.

Reusable software has many forms and definitions. *Reuse*, for our purposes, means the ability to use a particular component many times, either within the same application or in various applications, *without modification* of the original component. Object-oriented programming languages, such as C++, profess to be replete with reuse. Although that may be true, there are other methods to obtain consistent reuse without resorting to building every piece of an application in C++.

Types of Reuse

There are many ways to reuse software when you're developing applications. I'll discuss two ways that are in wide use today: reuse through the inheritance mechanisms provided by object-oriented programming languages such as C++, and reuse through the development of discrete, language-independent components. The first method, reuse via inheritance, typically requires access to the implementing source code. Source code is required because few standards exist that allow the sharing of objects between language compilers (such as Watcom and Borland). Reuse with inheritance typically requires that the inheritor use the same language used in the original implementation. In other words, you cannot reuse, via inheritance, a C++ class in Smalltalk or Java. The second method, reuse by building and using components, does not depend on the implementing language, relying instead on component standards that allow sharing across languages and environments.

Inheritance allows a developer to extend an existing code module by augmenting the original base class code. A great amount of functionality may already be provided by the base class, so implementation of additional functionality requires only minor changes to the new subclass. This new class provides significant functionality with only a small change or augmentation. Inheritance used in this manner provides significant reuse of existing code. The principal problem with this approach is that to continue to augment and reuse, you must continue to use the same development language. Today's C++ compilers do not allow the sharing of C++ objects developed using different compilers, primarily because there is no standard method to mangle, or decorate, the exposed function points. *Mangling* is a method used by C++ compilers to construct unique names for public functions. C++ allows multiple definitions of functions using the same function name but different argument types. Mangling "mangles" these names (by attaching an encoding of the argument types) so that the linker can resolve the specific function at link time. This process is described in more detail in the next chapter. Smalltalk objects are similar in that no binary standard exists to describe the various Smalltalk language objects. You can overcome these problems by using one of the new *wrappering* technologies, which allow you to wrap an existing C++ or Smalltalk class with an external, language-independent interface. The two primary standards are IBM's System Object Model (SOM) and Microsoft's COM.

Reuse through inheritance is wonderful when the dynamics of a project allow for the building of applications using only one primary language, but in larger projects that require multiple languages and are staffed by individuals with diverse programming backgrounds, this arrangement can be difficult to implement. Another problem with inheritance-based reuse is that it is difficult to manage in large projects. Inheritance-based reuse creates a large hierarchical structure of classes that can complicate the process of code changes. If a change is made to a base level class, every class that derives from it will be affected. This can cause massive recompiles and relinks as the changes cascade through the project.

I'm not implying that inheritance-based languages are bad, just that they have their place and purpose. We will use C++ throughout this book to develop components. But the inheritance feature will be used internally by the components being developed and not as the primary reuse mechanism.

The other important type of reuse is that provided by software components. A component can be another application such as Microsoft's Word for Windows, a database management package, an ActiveX control, an Automation server, a Java applet, and so on. These components offer reuse on a different scale than that offered by language-based inheritance. This book focuses on building these types of components.

Software components can be built with various languages. The important characteristic that makes a component reusable is that it has a well-defined binary standard interface. A *binary standard* provides sharing, or interoperation, of objects developed using disparate languages and tools. We'll cover this in more detail soon. Components should also be generic and configurable. As mentioned earlier, the *interface*, or exposed functionality, is the most important aspect. Today the primary interface implementation mechanisms are IBM's SOM and Microsoft's COM. We will use COM (which includes OLE and ActiveX) because this book describes the development of Windows-based software and because COM is the de facto component technology on Windows machines.

Reuse and Portability

Another important aspect of reuse is portability. *Portability*, as used in this book, refers primarily to the easy movement of source code among the various Windows platforms: Windows 3.x, Windows NT, and Windows 95. Portability on a larger scale, including non-Microsoft platforms, is a valid goal, but we will touch on this topic only briefly.

Microsoft has recently stated that it will provide multiplatform support for most basic ActiveX technologies, including ActiveX controls. Versions of Internet Explorer 3 will be provided for 16-bit Windows, 32-bit Windows, the Macintosh, and certain flavors of UNIX. Portability among these operating systems will be provided with Visual C++ and the MFC libraries. Microsoft has also licensed the technology to other companies that are providing support on other platforms (such as IBM's MVS).

The code that we will develop will be portable across the various Microsoft Windows platforms. We will use the MFC libraries, an application framework that abstracts many of the complexities of Windows development and hides the differences among 16-bit Windows, 32-bit Windows, and Macintosh platforms. The 32-bit version supports Windows 95, Windows NT, and the Macintosh. Source code is portable among these operating environments with minor changes and a recompile and relink. Details and caveats will be discussed in later chapters as the various items are encountered.

New Development Methodologies

Many development methodologies have come and gone within the last few years. Some have confirmed their worth and enjoy broad acceptance. Examples include structured programming, client-server–based development, and the ubiquitous object-oriented analysis, design and development. Others, such as computer aided software engineering (CASE), expert systems, and artificial intelligence (AI), have proven useful only in well-defined, restrictive settings and do not yet show promise for broad application. Component software development is proving itself to be a viable and cost-effective method of developing applications for the Windows environment.

There are two primary methods of developing Windows applications. The first method is based on the visual combining of discrete software components. The most obvious example of this paradigm is Visual Basic, but there are many others, including Borland's Delphi, Computer Associates' Realizer, and IBM's Visual Age products. The more prevalent method is the use of object-oriented languages, typically C++.

Usually, an application framework (either supplied by a vendor, as with MFC, or developed in-house) is used to help manage complexity in large Windows applications. Because of the dynamics of the PC market, Windows developers must choose at least one of these methods to remain competitive. In reality, nearly all good development tools provide a mixture of both techniques.

Visual Programming

Initially, Windows development was done using the Windows Software Development Kit (SDK) and the C language. These tools were the only ones available to build professional, efficient Windows applications. Then, in 1991, Microsoft introduced Visual Basic, a radically different method of developing Windows applications. With Visual Basic you can write a "Hello World" program with zero lines of code. Even though there were certain deficiencies in Visual Basic development, it provided a truly *visual* development environment in which developers could drag-and-drop controls (or components), modify their properties, and have a fairly robust application with only a few hundred lines of Visual Basic code. Compare this to the thousands of lines of C/SDK code that would be required to provide similar functionality.

Visual Basic also provided a Control Development Kit (CDK) that allowed third parties to develop generic and specialized controls that could be incorporated seamlessly into the Visual Basic development environment. This proved to be a major event in the history of component software.

Visual Basic provided the genesis for software component building and combining. The use of the Visual Basic custom controls (VBX)—and now the incorporation of COM, OLE, and ActiveX support—has made Visual Basic the standard for building prototypes and commercial software. Even though the VBX specification did not provide an open standard for component development and was restricted to its 16-bit heritage, VBXs became the de facto "components" in many Windows development environments. Support for VBX controls is now included in most 16-bit C++ compilers and other visual development tools. Many vendors had to reverse-engineer the VBX architecture to enable it to work within their products, but the large number of VBX components made this a requirement. With ActiveX control, Microsoft now has an open standard for visual component development. Vendors can easily incorporate support for components developed with this standard, and today nearly all of them have done so.

The new Web-based technologies add another dimension to this environment. Web developers and designers must produce and publish material very quickly. Microsoft's ActiveX control technology makes it easy for these developers to incorporate dynamic content into their products. Design and development tools provide an efficient method of assembling these various components—HTML, ActiveX controls, and VBScript—to produce effective, dynamic Web pages.

Object-Oriented Languages

Object-oriented languages such as Smalltalk, Java, and C++ have recently received a great deal of attention (and use) within PC and workstation development environments. All of these languages greatly ease the task of building complex and reusable software. Various factors have contributed to the success of these languages. C++ is a superset of the widely used C language and lets developers progress to object-oriented methods as time permits. Smalltalk has taken longer to gain momentum primarily because its

interpreted, run-time binding performance requirements make it slower. Smalltalk is a pure object-oriented language that started in academia for instructional purposes. Although Smalltalk has been around for more than 20 years, only now is the hardware sufficiently fast to allow its use in typical application development environments.

Smalltalk will not be used in this book; nevertheless, it is an important language of the future—many people believe it to be the COBOL of the '90s. Instead, we will use Visual C++ and the MFC libraries to develop COM-based software components. Visual C++ and MFC provide sufficient speed as well as a level of abstraction from the underlying complexities of both the Win32 and COM APIs.

Java is the newest object-oriented language, and its popularity has soared along with the popularity of the Web. Java is a product of the new component-based software model. Its initial purpose was to facilitate the development of small, portable *applets*, which are perfect for Web environments. Java removes most of the development difficulties and complexity of C++, and this is one of its main selling points. Unlike C++, Java is a new language unencumbered by a long history of modifications. It isn't perfect, however, because it still has some growing to do.

Although object-oriented languages provide significant help with software development problems, they are deficient in areas that are important to the broad application of the object-oriented technology. C++ and Smalltalk do not provide a binary standard that would allow the sharing of objects across languages or environments. Each language has *proprietary*, or nonstandard, methods of implementing specific features of the object-oriented paradigm. Because we're focusing on C++, let's review why it currently has such limitations.

C++

The C++ language was supposed to solve many of the problems caused by the increasing complexity of building large or complex applications, and to a large extent it has succeeded. C++ is useful when you're implementing large applications, frameworks, and systems. A good example is Microsoft's Foundation Class libraries. Visual C++, coupled with MFC, allows much of the complexity of Windows and COM software development to be hidden. Application frameworks such as MFC provide robust, well-tested application code. The developer doesn't have to initially understand all the underlying complexity. A developer can become productive quickly, gradually developing an understanding of the underlying architecture. There are problems with this approach (aren't there always?), and we'll discuss them in Chapter 3.

C++ has failed in its ability to provide a standard method of describing the produced objects in a binary fashion. For C++ objects to be effectively shared or reused across development platforms, hardware platforms, or even compilers, the original source code is required. There are no compiler-independent standards for decorating or mangling C++ function names, so to share these objects, the secondary user requires either the original source code or platform-specific (Windows, OS/2) and compiler-specific (Microsoft, Borland, Symantec) binaries (**.LIB** or **.OBJ** files). This doesn't mean that C++ isn't an effective language. It is. But more is required to take objects developed in C++ to the next level, where they can be shared and reused across environments and languages.

We're discussing only static objects; we haven't yet raised the issue of sharing actual run-time instantiations of these objects or components. There are major problems here as well, particularly in the sharing of objects across processes on local and distributed processors.

What is needed is a higher level of object orientation or a standard that encapsulates the underlying objects to allow many of the object-oriented features to be used across development environments. Such standards or technologies are described as object-oriented systems.

Object-Oriented Systems

Object-oriented languages provide significant leverage in the development of software, but to incorporate or share the developed objects across disparate languages, processes, and environments requires additional software. Object-oriented systems enable the sharing, or *distribution*, of these objects across the mentioned boundaries. Two primary standards address these requirements: Microsoft's COM, on which OLE and ActiveX are based, and IBM's SOM and Distributed SOM (DSOM). Each standard has strengths and weaknesses, but we will focus on what I feel is the de facto, Windows-based standard: Microsoft's Component Object Model.

A Binary Standard

If you haven't yet heard the phrase "provides a binary standard" (I've mentioned it a few times), you soon will. A *binary standard*, in the context of object and component sharing, provides the means by which objects and components, developed using various languages, from disparate vendors, running on heterogeneous platforms, can interoperate without any changes to the binary or executable. Whew. Let's try that again with pictures.

Figure 1.1 illustrates a hypothetical problem that we need to solve. We've written a wonderful text editor in Visual Basic, but we don't feel like implementing the spell checker, text search functions, or paragraph formatting in Visual Basic. So we locate specialized objects through our favorite object supplier or from a group within our organization. Great—this will save time. But wait—how are we going to get all the pieces to work together? Visual Basic knows nothing about the binary format of Borland's C++ objects, nor does it know about Visual C++ or Watcom binaries. In other words, there is no binary standard that describes how different languages should store the machine language representation of the implemented language algorithms that would allow other languages to access this implementation.

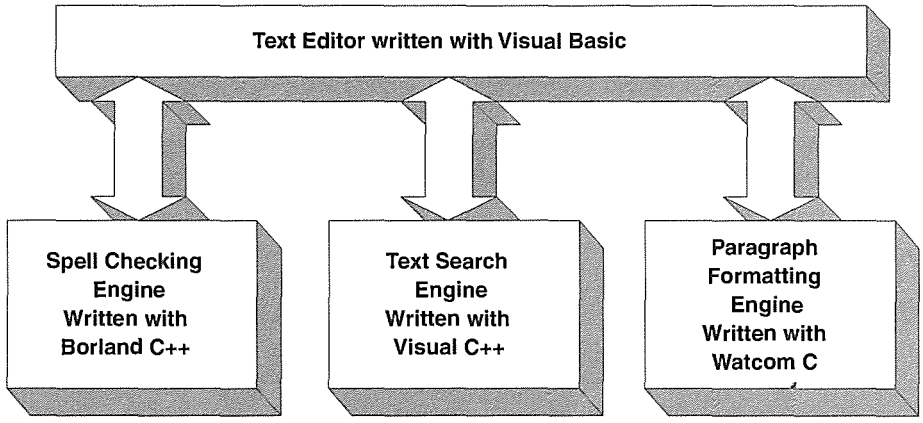


Figure 1.1 A hypothetical application.

Even without a binary standard, there are certain things we can do to solve this problem. If we have the source code to each module, we can determine which functions we would like to call from Visual Basic; then we remove the C++ mangling by declaring them `extern "C"` and recompile the source. To access these functions from Visual Basic, they must be implemented in a Windows DLL; they cannot exist in an executable (EXE).

Using a Windows DLL solves the initial problem by letting us use these functions within Visual Basic. We must, however, explicitly declare the functions from Visual Basic. Let's use something like the following:

```
Declare Function SpellCheck lib "spell.dll" (...) as long
Declare Function TextSearch lib "search.dll" (...) as long
```

Now we can access the various functions in the component "engines" to perform the needed tasks, and we complete the editor. This approach solves the problem, but only temporarily. Later, as enhancements and bug fixes are added to the C++ objects, we will be responsible for implementing them in our source code. Because we initially had to modify the source, we must also maintain any enhancements and bug fixes ourselves. As you can see, the C++ objects are no longer generic, but have become specific to our application.

Another problem with our approach is that the names of our DLLs are hard-coded within the Visual Basic `Declare` statements. Later if we want to plug in another DLL, possibly having a different name, we must touch the Visual Basic source and re-create the executable. A binary standard should allow a developer to plug in different components without modifying the original code.

This is a simple illustration of the problems with existing object-oriented language solutions. There are ways to get around interoperability problems in software development, but what we need is a standard way to do it, one that is prescribed and extensible so that vendors can build highly interoperable software. Microsoft's COM provides this binary standard.

Objects Versus Components

To help in the understanding of what objects and components are, we need to define identifying characteristics for the two types. The term *object* will be used for the traditional software module that is produced using object-oriented languages; the binary form of this module is typically an OBJ or LIB file. The term *component* will be used for items that have a binary standard wrapping, or interface, implemented around a language-dependent (usually C++) object, allowing the component to be reused across language environments. A component's binary form is either a DLL or EXE. Following are more detailed definitions.

Software Objects

In this book, the term *object* will primarily pertain to C++ or Smalltalk objects as they are used within their respective development environments. Object-oriented language objects have the following characteristics:

- **Reuse through inheritance**: As we've discussed, the primary method of reuse with object-oriented languages is through the use of inheritance.
- **Language dependency**: The objects produced are dependent on the language or compiler in which they are developed.
- Often, objects are dependent on other objects within their environment for part or most of their functionality.
- No current standard exists for use of objects outside of the programming language—that is, for proprietary methods of implementing certain object-oriented techniques.
- Reuse requires an understanding of the underlying or dependent objects.
- Objects are normally pieces of a particular application and can be difficult to extract for reuse.
- To reuse objects requires substantial knowledge of the underlying programming language.

Software Components

What is a software component? There are many definitions, but here I will define the components that we will build and use in this book. The type of component that we will build does not necessarily have all the traits of a normal object-oriented programming language object. *Component* objects typically will not have the popular characteristics of inheritance or polymorphism, but they will have what I think is the most important object-oriented characteristic of all: reusability. Inheritance and polymorphism are important but not nearly as important as the eventual reusability of the produced object. Purists will say that inheritance is important to—and basically provides for—reusable objects. I agree, but only when you are working within the framework of a single, inheritance-based language, such as C++. Few development organizations do this.

Our definition of a software component provides the following characteristics:

- **Internal implementation details are completely hidden**: Components *encapsulate*, or hide, to the fullest extent the details of how their functionality is implemented.

- **Independent of other components**: Components are self-contained and shouldn't depend on other components to provide supporting services.
- **A well-defined interface**: Components should provide a well-documented set of services.
- **Use of a binary standard to expose external services**: Components will use a language-independent, industry-standard method of exposing services. We will use Microsoft's COM, OLE, and ActiveX.
- Reusability is achieved through binary reuse of the discrete component and not through language-based inheritance.
- Components should not require *user* understanding of the implementing language and, if possible, should require minimal programming expertise to use.
- The component's behavior, and not its implementation, is what is important to the component user.

Applications as Components

Given our definition of software components, you can see that a large Windows application such as Microsoft's Word or Corel's Quattro Pro can be a component of another application. The key is that these component applications provide a well-defined, external interface to their underlying application functions (sounds like a binary standard). By accessing these exposed functions, other applications can harness the built-in functionality and provide it to their users.

Today, many such applications provide functions that most client applications will not need. But in the future, smaller, more specialized applications (or components) will be developed for use by a broad range of component-based applications. In later chapters, we will build both business-specific and generic application-based components. As you can see, the distinction between applications and components can be hard to discern. It will eventually become subjective, significant only in the context of the specific development project.

The Component Interface

How a component implements what it does is of no concern to the component user. The component user asks, "What can this component do for me?" A component is completely described by its interface to the outside world. The interface details what the component can and cannot do under various conditions. That's the beauty of encapsulation. No one need know how the dirty work is done. The component user selects components solely by their advertised behavior. The only implementation detail that matters to the component user is performance: whether the component is fast enough.

Figure 1.2 depicts a simple component. Its primary capability or behavior is the evaluation of algebraic expressions. Its single interface provides four functions: `SetExpression`, `Verify`, `GetExpression`, and `Evaluate`. These functions encompass all the behavior this component exposes. Within the implementation, there are many other functions as well as internal structures to provide the external functionality, but the component user neither has nor needs any knowledge of these details.

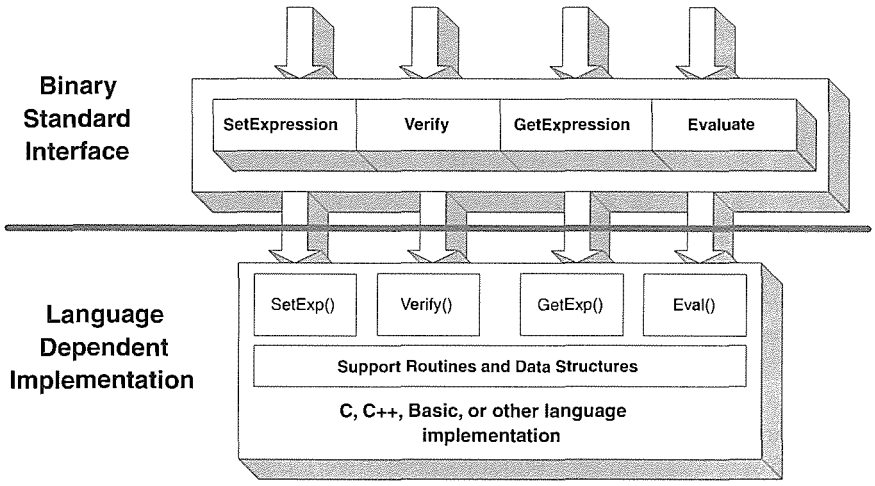


Figure 1.2 Component interface and implementation.

COM: The Facilitator

The technology that allows users to build component-based Windows software is Microsoft's Component Object Model standard. COM provides a system-level, well-defined set of services that standardizes the sharing of data and functionality. COM provides a language-independent way of exposing capabilities from your software, making software reusable at the binary level. COM and ActiveX provide an open, well-defined standard that applications can use to expose functionality to other applications or scripting languages such as Visual Basic.

A Brief Overview of OLE and ActiveX

The next few sections will briefly describe COM, OLE, and ActiveX. It would require more than one book to completely document COM in all its forms. If you need more information than this book provides, I recommend that you pick up a copy of Kraig Brockschmidt's book *Inside OLE* from Microsoft Press. It is required reading for anyone doing serious work with COM and OLE.

OLE is constructed as a layer above the Component Object Model and provides various services to application developers. The most familiar OLE concept is that of the *compound document*. OLE compound document services let you embed component objects created by other applications within an application's document. An example is embedding a Microsoft Excel spreadsheet within a Microsoft Word document. One of the most important OLE technologies for component development is *OLE automation*, or just *Automation*.

Like OLE, ActiveX is based on COM. Until April 1996, the term *ActiveX* did not exist; OLE was used to denote all the application-level services built on top of COM. Today, however, OLE has been relegated to use in those technologies that relate to Microsoft's compound documents (or object linking and embedding) services, and ActiveX has taken over as the primary name for most of the technologies based on COM. In particular, it refers to those services that are tied to Microsoft's Web-based technologies.

Automation and ActiveX Controls

Although COM, OLE, and ActiveX encompass a large area of application use and development, we will focus primarily on two areas of COM-based development. The first is Automation, which makes it easy for applications or components to expose various features for use by other applications. Let's say you're developing an application that needs to fax a document. You could develop the routines to access the fax modem, dial the phone, and so on, but a much easier approach would be to use an existing fax application to do the work for you. There are many ways to do this. One is to let Microsoft Word do most of the work (along with fax hardware and software that includes a Windows print driver). Using Automation, you can drive Word externally, tell it to load your document, perform a document conversion if necessary, and then print it directly to the fax modem. You're depending on software that you have no developmental control over, but it works well, and you need write only about 50 lines of code. That is reuse.

The other software component that we'll focus on is the ActiveX control. Although Automation is used primarily for nonvisual components, ActiveX controls provide the same type of reuse that Automation provides, but for visual components such as Windows entry fields, command buttons, and so on. ActiveX controls are also an important part of Microsoft's Web-based strategy. They play a major role in the majority of Microsoft's new technologies.

Monolithic versus Component-Based Applications

Most Windows applications can be called *monolithic* applications. A monolith is defined by the *Random House College Dictionary* as having a "uniform, massive, or intractable quality or character." When applied to software, this term describes applications that have grown massive with options and features, few of which the average user will ever use. The application is trying to be all things to all people. This type of Windows software will eventually become uncommon. It will be replaced by software based on combinations of individual, specific, and highly efficient components.

Figure 1.3 illustrates the differences between a monolithic application and a similar application built using COM-based components.

Almost every major Windows software package is currently implemented using the monolithic model. Microsoft Word, Excel, and Project are examples. Even though Microsoft Word and Excel may share various pieces, or software modules (maybe the spell checker), this sharing is implemented using closed, proprietary mechanisms. These mechanisms aren't meant to be proprietary, but the current C++ compilers do not intrinsically support interoperability standards. As I've mentioned, existing C++ compilers cannot share objects at the binary level.

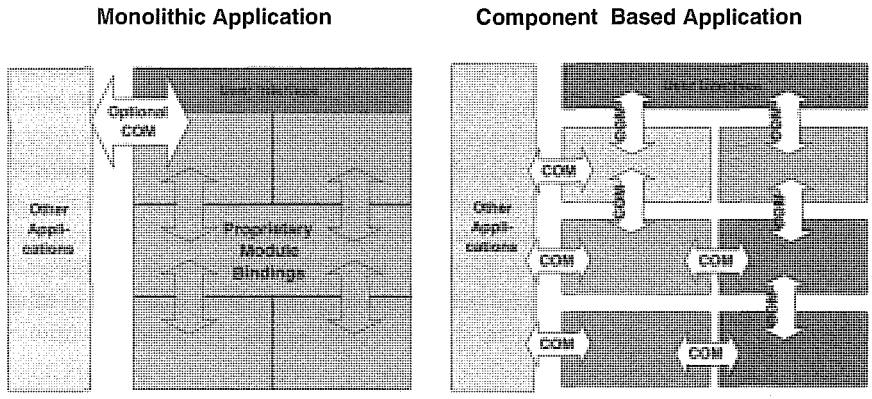


Figure 1.3 Monolithic versus component-based application.

There are also examples of small, highly functional software applications written using the new component-based approach. Microsoft's browser, Internet Explorer 3.0, is written using this approach. The main executable, **IEXPLORE.EXE**, is only 15KB in size. The majority of its functionality is provided by an ActiveX control.

What's in It for Me (the Developer)?

Why should software developers be excited about the future of component software? Many developers spend their days solving the same old problems. Sometimes they may solve the old problem in a new, creative way, but under normal circumstances, it's just the same old algorithm that we've used a hundred times. How many ways are there to implement a linked list? In a nutshell, components allow developers to solve new problems instead of continually solving the old ones.

With the advent of component software and object-oriented components, developers are free to tackle the real problems in software development. They can focus their creative energy on the combining of robust existing components into much more functional (and fabulous) applications. This technique is currently in use in pure C++ environments. In these C++ environments, experienced and astute developers are many times more productive than average programmers. Their productivity is enhanced through the use and understanding of the available C++ objects and tools. They never spend time writing a linked-list routine. Instead, they solve real problems.

Buy versus Build (or "Not Invented Here")

If we are to develop applications rapidly, it is important to use applications or components that are already available. Why spend ten worker-years developing a word processor when Microsoft Word is available for the price of a few hours' labor? Would you develop a database management system from scratch if your

project required database services? Of course not—but that mentality is prevalent in many corporations. For whatever reasons, many organizations think that if a product isn't developed in-house, it's not worth using. In certain situations, proprietary development of software is necessary because it provides total control—you're not at the mercy of another vendor's bugs. But I bet these situations aren't nearly as common as most corporate developers think.

I once worked for a company that developed a Lotus 1-2-3 work-alike, because a certain client wanted spreadsheet capabilities. Later, we developed a messaging, or middleware, system even though a superior product—one that many developers were already familiar with—already existed. This "not invented here" mentality costs businesses millions of dollars each year. If they are to compete with other companies that embrace component-based development, they had better change. In today's environment of robust, inexpensive application frameworks, C++ source libraries, and components, it is ludicrous to continue to reinvent the wheel on a daily basis.

Component Builders versus Component Assemblers

Component development and assembly requires two types of application developers. There are those who create the components that can be assembled into various applications, and others who assemble these components into the finished applications. Many times these two development tasks are handled by one individual, but in the future these responsibilities will be assigned to separate developers, each focusing on the specifics of the job at hand. Each developer has a very important job, and in most cases these jobs are quite different.

The *component builder* focuses on building reusable components that provide either well-specified generic functionality, specific business functionality, or methods of gluing these components together effectively. Component builders will typically be computer science graduates with a low-level understanding of the problem and skills in various programming languages.

The *component assembler* focuses on the business problems to be solved. He or she analyzes the problem, chooses from the various components, and then assembles them using a high-level glue or scripting application. Typically a business major, the component assembler should have a business or analysis background in the field in which the program is being applied.

Although other individuals are also involved in building applications for the business environment, the focus of this book is to provide component builders and assemblers with techniques needed to put this process to work.

Component-Based Development and the Web

Development of Web pages and related software is a perfect application for component-based software techniques. The client-side Web browser is an ideal medium in which to use this technology. Small, easy-to-use components such as Java applets, ActiveX controls, and Netscape plug-ins make it easy to quickly add functionality to Web pages. The majority of the functionality will be encapsulated within these components, and a small amount of script coding such as VBScript will be used to tie the components together.

The Web server will benefit from the component-based paradigm as well. As we'll see in Chapter 12, ActiveX controls and VBScript can be used effectively here. The server-side architecture uses ActiveX controls to provide nonvisual services.

In short, the Web will hasten our move to the component-based model of development. The Web will be full of components. With the maturity of technologies such as Java and ActiveX, the Web page designer will become an assembler of discrete components: a Java applet here, a couple of ActiveX controls there, some HTML, and finally some VBScript to tie it all together. The Web user will be a downloader of components. As the user browses the Web, each page may install a number of small components on the local system. As all the pieces work together, a dynamic Web page is produced that not only is great to look at but also provides quite a bit of functionality.

Summary

There are two general methods of developing Windows software: object-oriented and visual. Both methods are effective, but visual programming enables quick and effective development by more individuals, including nonprogrammers. Our focus will be on developing software components (using the object-oriented method) that can be used and reused in both object-oriented and visual environments.

For corporations and developers to remain competitive, the reuse of newly developed and existing software is paramount. Reuse can be achieved in various ways. Inheritance, the principal method used in object-oriented languages, is a powerful feature, but isn't a requirement for reusable software. Software components achieve effective reusability without it. The primary difference between a well-written C++ object (or other language module) and a software component is the addition of a binary standard wrapper that allows its use across various languages. Figure 1.4 illustrates this concept. For our purposes, Microsoft's COM/OLE/ActiveX combination, particularly Automation and ActiveX controls, will provide these wrappering services.

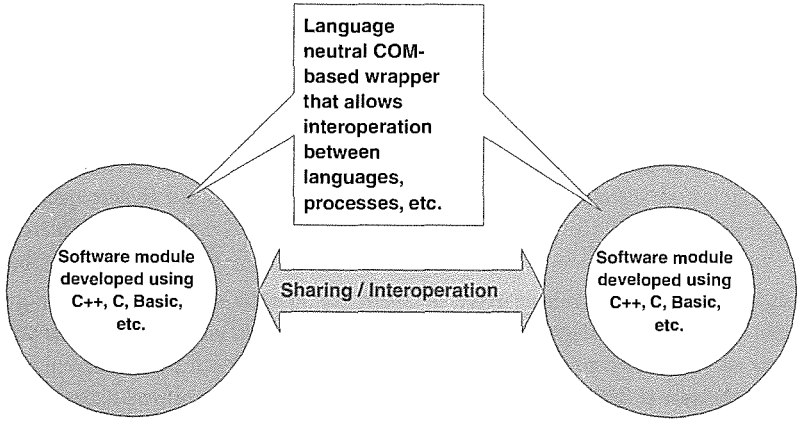


Figure 1.4 Binary standard wrapper.

Chapter 2

Designing Reusable Components with C++

In Chapter 1, we discussed the importance of reuse in software development. Software objects and their binary standard companions, software components, were offered as an effective method to achieve reuse in the development cycle, within both C++ and other development environments. In this chapter we'll begin the process of developing software components.

The first step is to create C++ classes that are themselves reusable or generic. These classes should not be specific to a particular task but instead should be general enough to be used in multiple areas of an application as well as across applications. This concept is far from new and has been used since the advent of structured programming. But C++ provides a fertile field in which to design and develop reusable modules or objects.

Not all classes can be designed to be widely reusable. Determining when a class might be reusable across the application and across projects is an art. In many cases, the added overhead of designing a class to be reusable is not worth the effort it requires. Designing reusable classes is like almost everything else in life: it's best not to have too much or too little of anything. Keep this in mind. Some organizations try too hard to make all the code they write reusable, and this added overhead can have a detrimental effect on the development process.

In this chapter, we'll go through the process of developing a general class that is composed of other general classes. Throughout this process, keep in mind that the potential user of the classes may not have access to the C++ language. If we design the classes with a multiple language focus, they can easily be adapted for use outside the C++ environment. In later chapters we will use these C++ objects to develop binary standard software components.

Before we begin developing C++ objects, we must be fairly proficient in the language, so this chapter begins with a quick tour of C++. This overview is far from comprehensive, because we focus only on those areas of C++ that we will use in the various projects and those that are necessary to understand various idioms used in Microsoft's Foundation Class (MFC) libraries. Use of MFC is described in more detail in Chapter 3, although minor references to certain MFC classes are made in this chapter. For a complete review

of the C++ language, several excellent books are listed in the Bibliography. (One of my favorites is the *Effective C++* series by Scott Meyers.)

C++, the Language

This book uses the C++ language exclusively in the examples and sample programs. Although you may not need the quick refresher course provided here, you might as well plod through it, because I make a few comments on how MFC uses certain features of C++. The overview is by no means a course on C++; it only hits the high points and provides what is necessary to understand the C++ presented in this book. There isn't a long discourse on deep vs. shallow copies, and so on.

C++ is a supercharged version of the C language. C++ includes all the features of C and provides a way for C programmers to ease into the age of objects. I've programmed in C for more than eight years and in C++ for about three years. Daily, I become more aware of the power of C++. It is a profound language. It has many levels that developers can attain as they gain experience. An experienced C++ developer who is intimately familiar with a powerful framework and a solid class library (either self-built or commercial) is a programming machine. The productivity of such a developer is orders of magnitude greater than that of a beginning or slightly experienced C++ developer. This also can be said of developers who use other, non-C++ languages. Intimate knowledge of any good language and good supporting tools makes all of us productive.

This book will help you in your efforts to become a proficient C++ developer, but that's far from its main purpose. As component developers, our goal is to make our users—visual developers—as productive (although at a higher level) as seasoned C++ developers in much less time.

Classes (Encapsulation)

C++ endows the C language with the object-oriented characteristics of encapsulation, inheritance, and polymorphism. The primary device for these new features is the C++ class. Classes allow the encapsulation of data and of the functions that affect the data. The goal of a class is to capture and encapsulate the essence of a particular thing or object and then expose to users of the class only those items that are important for its use. The details of the internal implementation are encapsulated, or hidden, from the class user.

To make sure we're using the same terminology, let's go over some C++ terms. A *class* is a definition of a C++ object and is a compile-time construct. Once compiled, a class can be *instantiated* to create a run-time instance of the class definition. This run-time entity is typically deemed an *object*. There can be zero, one, or many of these instantiated objects during the execution of an application. They are quite dynamic.

Items within a class are referred to with several different terms. *Member variables* typically describe the class's internal variables. The MFC libraries, and programs written using them, regularly use the convention `m_variable`, where `variable` is the actual name used to identify member variables. Another widely used term for member variables is *property*. This term describes the relationship of member variables to the class; size and color are properties of a fruit. The terms *data members* and *attributes* are also used to describe member variables.

```
class Fruit
{
   // Member variables, or properties, or data members, or attributes
   CString  m_strName;
   long     m_lWeight;
   CString  m_strColor;
   ...
   // Member functions, or methods
public:
   Cstring  GetColor();
   long     GetWeight();
};
```

Member functions and *methods* are used to describe functions declared within a class. The term *method* comes from its use in Smalltalk, where objects respond to *messages*, or methods. When discussing C++ classes and objects, I'll use *member function* and *member variable*. Later, when we're discussing ActiveX components, the terms *property* and *method* will be used with a slightly different meaning. This is appropriate, because when we're discussing these items at the component level, a component property may be implemented as either a member variable or a member function.

Now let's talk about what encapsulation is and how it can help developers. Encapsulation is important for C++ developers as well as developers who use components. Software components provide rigid encapsulation of functionality. A component user cannot access the source code or any of the internal structures used to implement the component's functionality. They can use only what is exposed by the component developer.

C++ allows the encapsulation within a class of data and functions that operate on that data. The class can hide or protect many of these elements from external users. *Users* in this context and throughout this book refers to the class library user or component user and not the end user of an application. Encapsulation inside classes is implemented by the `public`, `protected`, and `private` keywords. We won't discuss the `protected` keyword until after the inheritance section. For now, `public` and `private` will suffice. Examine the following class declaration:

```
class Expression
{
private:
   CString   m_strExp;
   Stack     m_Stack;
   Tokenizer m_Tokenizer;

   void      InfixToPostFix();
   int       Precedence();

public:
   void     SetExpression( CString str );
```

```
    CString  GetExpression();
    BOOL     Validate();
    long     Evaluate();
};
```

Here we have an `Expression` class. It appears to support expression conversion and evaluation, but the key is that the user of the class does not know how the conversion and evaluation occur. The class members and functions that follow the `private` keyword cannot be accessed by the class user. The user can access only those items designated as `public`. The implementation of the conversion and evaluation routines is hidden from the user. Later, if the class developer finds a more effective method of converting expressions, the developer can change the implementation without affecting the class user. We will use this example again later but, as you can see, encapsulation enables the hiding of implementation details. Other non–object-oriented languages can also do this. C functions, Fortran modules, and other language features allow encapsulation of functionality. But C++ provides additional features, such as the `public`, `private`, and `protected` keywords, that make encapsulation more effective.


NOTE

As you've probably noticed, a C++ class declaration looks very similar to a C structure declaration. In C++, classes and structures are nearly identical. The C++ language adds all the capabilities of classes to C structures with one minor difference. C++ class members default to `private`, whereas C structure members default to `public`.

Constructors

When a C++ class is instantiated, or created, an instance of the class is created. An *instance* of a class is an actual chunk of memory in the address space of an executing application process. A class is the definition of the characteristics of that chunk of memory. A C++ class is of no use unless instances of the class (objects) are created and used at run time. Whenever a class is instantiated, its *constructor* is called. The constructor's primary purpose is to initialize member data. Contrast this with the C technique of initializing structured data.

```
struct Fruit

{

    char* pszName;
    long  lWeight;
    char* pszColor;
};
typedef struct Fruit Fruit;
```

If we need to create and use a `Fruit` structure, we use the following code:

```
void SomeFunction()
{
   // Let's create an apple on the stack
   Fruit Apple;

   // The structure contains members that must be
   // allocated and initialized. This memory is allocated on the heap.
   Apple.pszName = (char*) malloc( sizeof( "Apple" ) + 1 );
   strcpy( Apple.pszName, "Apple" );
   Apple.lWeight = 6;
   Apple.pszColor = (char*) malloc( sizeof( "Red" ) + 1 );
   strcpy( Apple.pszColor, "Red" );

   // Do something with Apple...

   // Now deallocate any non-stack allocated memory
   free( Apple.pszName );
   free( Apple.pszColor );

   // As the Apple structure goes "out of scope" only its memory
   // is deallocated
}
```

To do the same using C++ classes and constructors, you might do this:

```
class Fruit
{
private:
   CString m_strName;
   long    m_lWeight;
   CString m_strColor;

public:
   // Provide a default constructor that initializes
   // the class members. This cannot be done using the
   // C structure method above.
   Fruit()
   {
     m_strName = m_strColor = "";
     m_lWeight = 0;
   }

   // Overloaded constructor. This constructor takes parameters
   // that initialize the members of the new fruit object.
```

```
    Fruit( CString strName, long lWeight, CString strColor )
    {
      m_strName = strName;
      m_lWeight = lWeight;
      m_strColor = strColor;
    }
};
```

Now when we need to create an instance of an apple, all we have to do is this:

```
void SomeFunction()
{
  // Let's create an apple on the stack.
    // We provide the parameters to initialize our
    // new apple. The constructor handles the allocations
    // for us. Contrast this with the C example.
    Fruit Apple( "Apple", 6, "Red" );

    // Do something with Apple...

    // As the Apple variable goes "out of scope"
    // C++ takes care of cleaning up any memory for us.
    // It does this with a destructor that we'll discuss
    // in a moment.
}
```

C++ makes the creation of user-defined types very easy. You code the logic in the constructor, and it is used whenever an object of that type is created, whether it is on the stack or on the heap. You could also do this in C by using a function that takes the particular parameters for the structure and a similar function for the deallocation of the dynamically allocated memory. In this case, you must ensure that you call the function for deallocation before the structure goes out of scope. As you will see in a moment, C++ provides for automatic deallocation ("destruction") of objects as they go out of scope.

Dynamic creation of structures and classes is a fundamental element of both C and C++ programming. Constructors allow the orderly initialization of dynamic elements as they are created. Constructors also remove much of the tedium involved in allocating dynamic objects. You code the constructor once, and it does the work from then on.

The syntax for declaring a constructor is the class name itself with zero or more parameters. If you do not specify any constructors for your class, the compiler will provide a default constructor that takes no arguments. On the other hand, if you specify any constructors at all, the compiler will not provide the default constructor. You must implement any constructors that you will need. Constructors can be *overloaded*, which provides the ability for a function to be multiply declared, and the compiler will deter-

mine, by matching the parameters, which function to call. Our previous example contained two constructors, each with the same function name. We will discuss overloading in more detail in a later section.

The new and delete Operators

In the previous example, which contrasted the dynamic creation of structures in C and C++, the objects were initially allocated on the stack. C++ provides two additional operators—`new` and `delete`—to help in the dynamic creation of objects on the heap. In C++, `new` should be used instead of the C `malloc` function, and instead of using the C `free` function you use the C++ `delete` operator.

```
// Instead of this
void TheCWay()
{
   char* pszColor;
   if ( ( pszColor = malloc( 128 )) == NULL )
   {
      // Error handling code...
   }
   // Do something with pszColor...

   free( pszColor );
}

// do this
void TheCppWay()
{
   char* pszColor = new char[128];

   // Make sure the new worked. We only need to do this
   // if we haven't installed a memory exception handler.
   assert( pszColor );

   // Do something with pszColor...

   delete [] pszColor;
}
```

The last example shows an important point to remember when you're using `new` and `delete`. When an array of objects (or intrinsic types) is allocated, it's important to append [] to the `delete` operator so that it knows whether one or multiple objects are being deleted.

There are other benefits of using `new` and `delete` instead of `malloc` and `free`. Instead of checking the return from `new` every time you allocate a new object, you can ignore it. Most compilers (including Visual C++) provide an exception mechanism to handle situations when memory is low. If `new` cannot allocate the

needed memory, an exception is thrown. This technique allows developers to handle memory failures in one specific function instead of sprinkling their code with tests for a NULL return from malloc. However, the compiler's default behavior is to return 0 if new cannot allocate the memory. You must provide a simple exception handler to enable an exception thrown when an "out of memory" occurs. That's why I used the assert after the new in the preceding example.

Also, you can call delete on a NULL pointer. This is defined behavior, and delete will do nothing. There is no need to write code such as the following.

```
// During initialization (construction)
char *pszTemp = 0;
...
...// pszTemp may be used during execution
...
// Sometime later, during cleanup (destruction)
if ( pszTemp )
   delete pszTemp;
```

Because calling delete on a zero pointer is defined, the if is not needed, although some developers still do it just to be safe.

The MFC libraries override the default new and delete operators when compiling and linking in debug mode. This enables MFC to keep track of all allocations and deallocations during the lifetime of your application. If your application terminates without deallocating all the memory allocated, MFC will dump (to the debug window) the memory address and allocating source line of each memory block that was not deallocated properly. This is a very useful debugging tool.

Destructors

Just as C++ provides a method to ensure the orderly creation of objects, it also provides for the orderly destruction of objects via a *destructor*. The syntax for a destructor is similar to that of a constructor except that destructors cannot take parameters. Like constructors, destructors also cannot return a value. Destructors are identified using the class name preceded by a tilde (~). Following is a destructor for our Fruit object:

```
class Fruit {
private:
   CString m_strName;
   long    m_lWeight;
   CString m_strColor;

public:
```

```
   // Provide a default constructor that initializes
   // the class members.
   Fruit() {
     m_strName = m_strColor = "";
     m_lWeight = 0;
   };

   // Overloaded constructor. This constructor takes parameters
   // that initialize the members of the new fruit object.
   Fruit( CString strName, long lWeight, CString strColor ) {
     m_strName = strName;
     m_lWeight = lWeight;
     m_strColor = strColor;
   }
   // Here's the declaration of the destructor
   ~Fruit() {
     cout << "Destructing Fruit object" << endl;
   }
};
```

Destructors don't normally do very much, and the compiler will provide one for you if you do not declare one within your class. User-declared destructors should be used, though, when your class dynamically allocates memory (that lives across method calls) from the heap during its lifetime. Here's an example:

```
// Declaration
// CobList is an MFC linked-list class, here we are
// inheriting the functionality of the linked-list.
// We will discuss inheritance in a moment. For now,
// just treat the FruitBasket class as a linked list.
class FruitBasket : public CObList {
public:
   FruitBasket();
   // Declaration of the destructor
   ~FruitBasket();
   ...
   void AddItem( Fruit* pItem );
};

// Implementation of the destructor
FruitBasket::~FruitBasket()
{
   // Get the position of the first element
```

```
    POSITION pos = GetHeadPosition();
    // Spin through and delete each element in the list
    while( pos )
    {
        // get the element, delete it, and increment to the next element
        delete GetNext( pos );
    }
    RemoveAll();
}
```

The destructor for the `FruitBasket` class ensures that the linked list of `Fruit` items is deallocated just before the object's final breath. If the compiler had provided a default destructor, any items in the list would not be properly destroyed. Here's another use for constructor and destructor implementations:

```
class Fruit {
    ...
    static int m_nCount;
    Fruit();
    virtual ~Fruit();
    ...
};

int Fruit::m_nCount = 0;

Fruit::Fruit()
{
    m_nCount++;
}
Fruit::~Fruit()
{
    m_nCount--;
}
```

We haven't talked about `static` items yet, but the preceding code enables us to keep a count of the number of `Fruit` objects that exist at any time during execution. As each `Fruit` object is constructed, a counter is incremented. During destruction, the counter is decremented.

Inheritance

In the last few sections we discussed the first element of object-oriented development: encapsulation. Now let's take a look at the object-oriented language characteristic that provides for our primary goal of software

reusability. As discussed in Chapter 1, the object-oriented language characteristic of inheritance provides for the reusability of software modules. Inheritance is the primary mechanism for reuse and is one of the major strengths of C++. The MFC libraries also use inheritance extensively, so it's important that you understand what it's all about. Let's start with an example.

Figure 2.1 illustrates a hierarchical relationship between a set of collection classes. Class hierarchies are essential for organizing classes of similar capabilities. Hierarchies are usually depicted as in the illustration in a top-down manner. The top object is the most general and is usually abstract (we'll talk more about this in a minute). As you move down the hierarchy, the classes become more specific in their function. This element is important in the design of object-oriented software, where the goal is to program the exception. In other words, when you're developing new software, (new classes), it is desirable to derive from an existing, tested class.

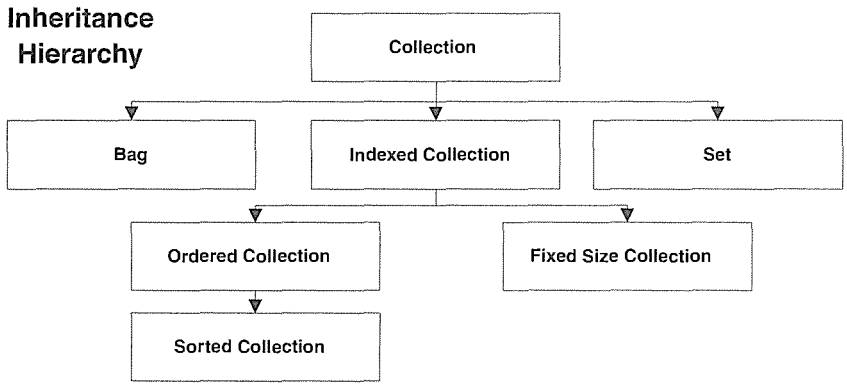


Figure 2.1 Example collection inheritance hierarchy.

The examples we discuss initially in this section are based on the Smalltalk implementation of collections. Smalltalk is a pure object-oriented programming environment and is useful for instructive purposes. The examples in this section use C++ syntax, but we won't concern ourselves much with how the functionality is provided. Our focus is on what inheritance is and what it can provide.

A *collection* is a group of disparate object instances. At the highest level in the hierarchy, there is no requirement for ordering or uniqueness of the collected elements. It's a loosely associated collection of objects. The top-level `Collection` class implements those functions (or methods) that are necessary for *all* collections. It strives to be general and thus provides only those functions that all collections need. At the next level are the `Bag`, `Set`, and `IndexedCollection` classes. These classes build on, or augment, that functionality provided by the base class, or superclass, `Collection`. Again, the purpose of aligning these classes in this way is to allow the reuse of base class code in the classes that derive from it. The `Collection` class might be declared this way in C++ (ignore the `virtual` keyword for now):

```
class Collection {

public:
```

```
   virtual void Add( CObject );
   virtual BOOL IsEmpty();
   virtual void Remove( CObject );
   virtual BOOL Has( CObject );
   virtual int  HowMany( CObject );
};
```

The `Collection` class encapsulates the basic behavior of a collection and implements the things that all collection classes should have. `Add`, `Remove`, `IsEmpty`, `Has`, and `HowMany` provide this basic functionality. For now, let's not concern ourselves with how the methods are implemented. Here's the definition of the `Bag` class.

```
class Bag : public Collection {

};
```

That's it—a `Bag`'s implementation is completely inherited from the `Collection` class. The functionality provided by `Collection` is all that is required to implement a `Bag` object. What is a `Bag`? It provides an unordered collection that allows duplicates and provides an easy way to count occurrences of objects. Pretty easy, isn't it? Here's how we might use the `Bag` class:

```
void SomeClass::SomeMethod()
{
   Bag aBag;

   for ( int i = 0; i <= 100; i++ )
   {
      // Add an integer that is the remainder
      // of the modulo division. This adds
      // integers in the range [0..9]
      // The Add member is from the base Collection class
      aBag.Add( i % 10 );
   }
   // How many 5s are in the bag?
   aBag.HowMany( 5 );
   // Are there any 14s in the bag?
   aBag.Has( 14 );
}
```

Now let's discuss the C++ syntax for declaring inheritance. In the `Bag` example, we are declaring a new class that is publicly derived from the `Collection` class. The colon following the class declaration (e.g., `Bag : public Collection`) signifies derivation, and the `public` keyword indicates public inheritance of the `Collection` class. Public inheritance is almost always used, and soon I'll explain why. With the syntax out of the way, let's look at the implementation of the `Set` class.

```
class Set : public Collection {

public:

   void Add( CObject);
};

void Set::Add( CObject obj )
{
   if ( Has( obj ) )
    return;
   // Call our parent's Add method
   Collection::Add( obj );
}
```

A `Set` is similar to a `Bag`, but it cannot contain duplicate elements. All we do is derive from the `Collection` class and override the `Add` member function. (We'll talk more about overriding functions in a minute. For now, overriding a member function means to re-implement, or hide, the implementation of the parent, or base, class.) As you can see, in the new `Add` method, we check to see whether the object being added already exists. If it exists, we simply return; otherwise, we call the base `Collection`'s `Add` method to add the item to the collection. This is what object-oriented development is all about. To add this new functionality, we wrote five lines of code.

The `IndexedCollection` class is a little more complicated. We need an additional member variable to help with the indexing, and it must be coordinated with the `Collection` class. We're talking in the abstract here, so don't worry about the details. Following is the definition for `IndexedCollection`. Indexed collections allow for iteration over the collection as well as direct access to specific elements via its index, but you can't specify the index when adding new elements. (It's basically a simple linked list.) Here's the definition for the `IndexedCollection` class.

```
class IndexedCollection : public Collection

{
public:

   // New Methods
   CObject  First();
   CObject  Last();
   CObject  Next( Position );
   long     IndexOf( CObject );
};
```

With this definition, the `FixedSizeCollection` is easy to implement:

```
class FixedSizeCollection : public IndexedCollection
```

```
{
private:
   long   m_lMaxSize;
   long   m_lCurrentSize;
public:
   long   Add( CObject );
};

long FixedSizeCollection::Add( CObject obj )
{
   if ( ++m_lCurrentSize > m_lMaxSize )
   {
     m_lCurrentSize--;
     return ERROR;
   }

   // Call the parent implementation to add the object.
   IndexedCollection::Add( obj );

   return NO_ERROR;
}
```

By now, you should see the effectiveness of reusing class code through inheritance. The goal is to build generic classes first and then proceed with inheritance to produce specific solutions. Inheritance won't solve all our problems, and we need to be careful not to abuse it. Let's look at what makes for good inheritance.

When should inheritance be used? This is a difficult question to answer. Typically, when we're deciding whether an object should be derived (or inherited) from an existing class, the new object should pass the "is-a" relationship test. In our simple example, a `Bag` "is-a" type of `Collection`. There is little ambiguity here, but in real development situations distinct delineations between objects seldom exist.

The MFC libraries use inheritance extensively. Almost all classes derive from the `CObject` class. Figure 2.2 shows a small part of the MFC hierarchy; in particular, this is a section of the visible objects hierarchy. The `CWnd` class encapsulates most of the Windows API functionality for generic window manipulation: `SendMessage`, `ShowWindow`, `InvalidateRect`, and all the others. And, thanks to inheritance, as you move down the hierarchy additional functionality is added. `CListBox` contains all the functionality of a standard window as well as the methods useful for listbox manipulation: `AddString`, `GetCurSel`, `GetCount`, `SetItemHeight`, and many others. Because `CListBox` is derived from `CWnd`, all the member functions implemented in `CWnd` are also available to the `CListBox` user.

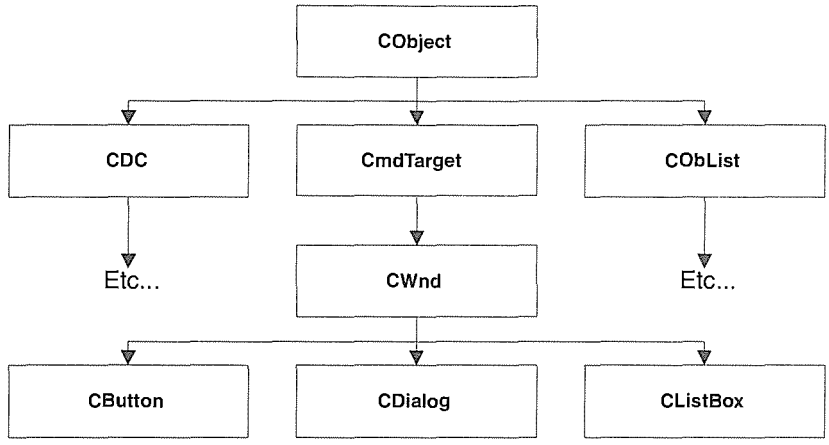


Figure 2.2 MFC visual object hierarchy.

Inheritance is only one of the methods that can be used to reuse code by building new classes. Other methods are equally effective. In a later section we will discuss class composition, another powerful method of reusing classes that have "has-a" type relationships.

The C++ keywords `public`, `protected`, and `private` allow class designers to tailor the inheritance capabilities of their classes. We'll discuss them next.

Public, Protected, and Private

As stated earlier, the concept of *data hiding*, or *encapsulation*, is an important characteristic of object-oriented development. C++ provides the `public`, `protected`, and `private` keywords to allow various degrees of visibility for class members.

`Public` is the easiest to understand. Anything declared public within a class, whether it is a member variable or a function, is visible to any user of the class. This is exactly the behavior if a structure is used instead of a class. The following declarations are equivalent:

```
struct Address {
   CString  m_strName;
   CString  m_strAddress;
   CString  m_strCityState;
};

class Address {
public:
   CString  m_strName;
   CString  m_strAddress;
```

```
   CString  m_strCityState;
};
```

`Protected` indicates that the members are private, or hidden, from class users, but are public to derived classes and their members. `Private` members of a class can be accessed only by other members of the same class. There is no visibility outside the class, even through inheritance. Determining when to use each of the three keywords is a design decision. Typically, class interfaces are declared `public`, and implementation members are declared `protected` or `private`. Here are some examples:

```
class Fruit {
// Private members, only Fruit members can access
private:
   CString  m_strName;
   void     DoSomethingPrivate() {}

// Protected members, only Fruit and other derived class members
//                    can access
protected:
   CString  m_strColor;
   long     m_lWeight;
   void     DoSomethingProtected() {}

// Public members - Everybody can access these
public:
   // Constructors
   Fruit() { m_strName.Empty(); }
   Fruit( CString str ) { m_strName = str; }

   void     DoSomethingPublic() {}
   void     SetWeight( long lWeight ) { m_lWeight = lWeight; }
   long     GetWeight() { return m_lWeight; }
   void     SetName( CString str ) { m_strName = str; }
   CString  GetName() { return m_strName; }
   void     SetColor( CString str ) { m_strColor = str; }
   CString  GetColor() { return m_strColor; }
};

//
// Publicly derive from Fruit
//
class Pear : public Fruit {
public:
   // Constructors
```

```
   Pear();
   Pear( CString strName );

   void  PearFunction();
}

Pear::Pear()
{
   // Can't do this because name is private

  m_strName.Empty(); // error

   // Must do this instead
   SetName( "" );
   // Protected member, OK
   m_strColor = "Green";
}

Pear::Pear( CString strName )
{
   // As above
   SetName( strName );
   m_strColor = "Green";
}

// Can access all of the public and protected members
// But can not access private members
void Pear::PearFunction()
{
   // We can directly access protected members
   m_strColor = "Green";

   // We can't to this, it's private
   DoSomethingPrivate();

   // We're derived so we can do this
   DoSomethingProtected();
}

main() {
   Fruit* pApple = new Fruit( "Apple" );

   // I'm a class user so this is illegal
   pApple->m_strColor = "Red";
```

```
    // Even more illegal
    pApple->m_strName = "Apple";
    // Illegal
    pApple->DoSomethingProtected();
    // Illegal
    pApple->DoSomethingPrivate();

    pApple->SetColor( "Red" );      // OK
    pApple->DoSomethingPublic();    // OK
    delete pApple;
}
```

All these examples use public inheritance. The keyword preceding the base class indicates the specific type of inheritance. `Protected` and `private` inheritance are seldom used, primarily because they defeat or severely hamper the notion of inheritance.

When `private` inheritance is declared, the members inherited from the parent all become `private` in the newly derived class. Also, any pointers to this new class cannot be substituted for pointers of the base class. Nothing inherited from the base class is exposed to the user of the derived class. In other words, the purpose of inheritance, extending and augmenting a base class, is forfeited. `Protected` inheritance is similar to `private` inheritance. All `public` and `protected` members of the derived class become `protected`, and any `private` members remain `private`. Most often, derivation of new classes should use the `public` keyword. Only in rare cases should `private` and `protected` inheritance be used. Private and protected inheritance disable or hide the interface of the base class, but leave its implementation.

Function Overriding

Function overriding allows the hiding or re-implementation of a particular member function that is inherited from a parent class. Continuing our fruit example:

```
class Fruit {
    ...
    // Ignore the virtual keyword for now.
    // We'll cover it momentarily
    virtual void SayName() { cout << "My name is " << GetName() << endl; }
};

class FrenchFruit : public Fruit {
    ...
    virtual void SayName() { cout << "Je m'apelle " << GetName() << endl; }
     };
```

Although this is a contrived example, it illustrates the point. By declaring a function, `SayName`, in a derived class with the same name as one in the parent class, you hide, or override, its implementation. This technique provides a method to override the default behavior of the parent class. Even though you are overriding the behavior of the parent class, you need not completely hide the functionality. You can call the parent implementation at any time during your new implementation. Using the MFC libraries, you'll do this quite often. Here's a simple example using `CObject`, the base class for most of MFC.

```
// Abbreviated from afx.h
class CObject {
...
public:
   virtual void Dump( CDumpContext& dc );
...
}

class Fruit : public CObject {
protected:
   CString   m_strName;
   CString   m_strColor;

   virtual void Dump( CDumpContext& dc )
   {
      // Dump the parent's members
      CObject::Dump( dc );
      // Now dump our members
      dc << "Fruit contains\n";
      dc << "\tName is " << m_strName << "\n";
      dc << "\tColor is " << m_strColor << "\n";
   }
}

class Apple : public Fruit {
protected:
   long    m_lNumberOfSeeds;

public:
   virtual void Dump( CDumpContext& dc )
   {
      // Dump the parent's members
      Fruit::Dump( dc );
      // Now dump our members
      dc << "Apple contains\n";
      dc << "\tNumber of seeds is " << m_lNumberOfSeeds << "\n";
```

```
   }
}

// Do this to get a run-time dump of the objects
afxDump << pApple;

// produces this output
>
> a CObject at $2411
> Fruit contains
>      Name is Apple
>      Color is Red
> Apple contains
>      Number of seeds is 35
>
```

Dump is a member function of CObject, the top-level object of the MFC hierarchy, and provides debugging information for classes that derive from it. Dump provides a hierarchical dump of member variables of each object in the inheritance chain. Derived classes don't need to know and sometimes cannot know about all the members of the parent classes. The parent's Dump function is called in the implementation of the derived member's Dump function. This arrangement provides a useful, encapsulated way of dumping all of an object for debugging purposes.

It isn't a good idea to override functions of the parent class unless they've been declared virtual. We will discuss virtual functions in a moment. For now, remember that declaring functions in derived classes hides all functions with the same name in the base class.

Function Overloading

C++ allows functions, methods, and operators to be overloaded. *Overloading* is an object-oriented term that describes a type of polymorphism. C++ overloading allows you to implement a function (or method) with the same name multiple times, but each implementation must differ by the number or type of parameters, and it cannot differ by return type only. Here is an example from the MFC libraries:

```
class CRect {
   ...
public:
   void InflateRect( int x, int y );

   void InflateRect( SIZE size );

   ...
}
```

The MFC `CRect` class provides two methods for inflating a rectangle. To make it easy for the class user, `InflateRect` can be called with an x and y coordinate or a `SIZE` structure. This arrangement allows the developer to pass the most convenient parameter. The compiler determines which function to call by matching the type to the appropriate function. How does the linker resolve the ambiguity of two public methods having the same name?

In Chapter 1, I pointed out that C++ objects don't natively support sharing across languages or even across C++ implementations, because there is no standard way of mangling or decorating function and method names. Borland, Watcom, and Microsoft mangle C++ function names in proprietary ways. Mangling provides unique public names for each C++ function, solving the problem of link-time resolution. The mangling algorithm guarantees a unique name for all possible combinations of function names and parameters. Both the compiler and the linker must agree on the particular technique of mangling. Here are the mangled names generated by the Visual C++ compiler for the two `InflateRect` methods. To see them, compile and link with the proper command-line switches to produce a **.MAP** file.

```
// Mangled names for InflateRect
?InflateRect@CRect@@RECXHH@Z
?InflateRect@CRect@@RECXUtagSIZE@@@Z
```

One obvious use of overloading is in constructors. Class constructors are often overloaded to take various parameters during construction. This technique enables an object to be instantiated in various states depending on what is known about the object at the time of instantiation.

```
class CRect {
   ...
public:
   // Overloaded constructors for CRect class
   CRect();
   CRect( int l, int t, int r, int b );
   CRect( const RECT& srcRect );
   CRect( LPCRECT lpSrcRect );
   CRect( POINT point, SIZE size );
   ...
}

// Mangled names for the above CRect constructors
??0CRect@@REC@AFUtagRECT@@@Z
??0CRect@@REC@HHHH@Z
??0CRect@@REC@PFUtagRECT@@@Z
??0CRect@@REC@UtagPOINT@@UtagSIZE@@@Z
??0CRect@@REC@XZ
```

The MFC `CRect` class has five public constructors. The constructors are overloaded so that the class user has multiple methods to initialize a `CRect` object. This technique provides flexibility for class users who can use the constructor that best fits their needs.


NOTE

When you're overriding an inherited function that is overloaded, all the parent member functions, and not just the particular one you have overridden, are hidden. Here's an example:

```
class Base {
public:
   virtual long  Sum( int i) { return i; }
   virtual long  Sum( int i, int i2 ) { return i + i2; }
   virtual long  Sum( int i, int i2, int i3 ) { return i + i2 + i3; }
};

class Derived : public Base {
public:
   // Overriding this function hides all implementations
   // in the base class.
   virtual long Sum( int i , int i2 ) { return SpecialSum( i, i2 ); }
};

main()
{
   Derived derived;
   // Error can't do this, the inherited overloaded members are gone
   derived.Sum( 1, 2, 3 );
}
```

Virtual Functions

Virtual functions allow C++ programs to resolve function calls dynamically (at run time) instead of statically at compile time. This powerful feature of C++ is what provides many of its object-oriented features. In *early-binding* languages such as C, the specific function to call in any particular instance is determined at compile time. C supports run-time binding, but the developer must do much of the work (using pointers to functions). C++ makes it much easier. *Virtual functions* allow the implementation of *polymorphism,* which is the ability of an object to respond differently to the same method or member function at run time depending on the object's type.

Virtual functions don't do much if you declare only one class. The strength of virtual functions is visible only as you augment existing base classes by creating subclasses of the original base class. The following

example demonstrates the power of using virtual functions, which allow the dynamic determination of member function calls at run time instead of compile time.

```
class Fruit {
protected:
   CString   m_strName;
   CString   m_strColor;
public:
   virtual void      Draw() {};
   CString           GetColor();
};
```

The declaration for our `Fruit` class now contains a virtual function, `Draw`, that returns `void` and does nothing. The base fruit class does not have an implementation for `Draw`, because each particular type of fruit should implement its own `Draw` function. Let's derive some fruit from this base class.

```
class Apple : public Fruit {
   virtual void  Draw() { cout << "I'm an Apple" << endl; }
};
class GrannySmith : public Apple {
   virtual void  Draw() { cout << "I'm a Granny Smith Apple" << endl; }
};
class Orange : public Fruit {
   virtual void  Draw() { cout << "I'm an Orange" << endl; }
};
class Grape : public Fruit {
   virtual void  Draw() { cout << "I'm a Grape" << endl; }
};
```

Each class that is derived directly or indirectly from `Fruit` implements its own `Draw` function, which prints that particular fruit's type. In itself, this isn't anything spectacular but look at the following code:

```
int main()
{
   Fruit* pFruitList[4];
   pFruitList[0] = new Apple;
   pFruitList[1] = new Orange;
   pFruitList[2] = new Grape;
   pFruitList[3] = new GrannySmith;

   for( int i = 0; i < 4; i++ )
   {
      pFruitList[i]->Draw();
```

```
      delete pFruitList[i];
   }
}

// Produces this output
>
> I'm an Apple
> I'm an Orange
> I'm a Grape
> I'm a Granny Smith Apple
>
```

This code illustrates the power of virtual functions. We've declared an array of pointers of type `Fruit`, the base class, and have assigned to each element the address of an instance of a particular derived fruit type. As the line `pFruitList[i]->Draw()` is executed, the program dynamically determines which member function to invoke. This dynamic binding is implemented with a virtual function table, which we'll discuss in a moment.

Pure Virtual Functions and Abstract Classes

Abstract classes provide a model or template for all classes that derive from them. In our `Fruit` example, the base class `Fruit` is a very good abstract-class candidate. A "fruit" is itself an abstract thing. In real life, we cannot instantiate a "general" fruit object, something that has the broad characteristics of a fruit but isn't a specific kind of fruit. Abstract classes are used this way to categorize and classify things that have similar characteristics. Our base `Fruit` class contains the essence of all fruits but nothing specific. This makes it a perfect example of an abstract class.

```
class Fruit {
protected:
   CString   m_strName;
   CString   m_strColor;
public:
   void      SetColor( CString str ) { m_strColor = str; }
   CString   GetColor() { return m_strColor; }
   void      SetName( CString str ) { m_strName = str; }
   CString   GetName()  { return m_strName; }
   virtual   void Draw() = 0;
   virtual   long GetAvgWeight() = 0;
    };
```

Abstract classes provide those properties and actions that all fruits share. Abstract classes can choose not to implement specific member functions and require deriving classes to implement those functions. In the pre-

ceding example, the `Draw` function is declared as pure virtual by using the notation = `0`. This notation indicates that all deriving classes must implement some form of the `Draw` function. By declaring a pure virtual function within a class, the class designer also makes the class abstract, meaning that the class cannot be instantiated. Although the class itself cannot be instantiated, pointers to the class can be used, and this proves to be an important characteristic. Recall the preceding example with the array of `Fruit` pointers.

```
int main()
{
   // Now we can't do this
   Fruit fruit;      // It doesn't make sense anyway, does it?

   // But we can still do this, and it produces the same output
   // as the example above.
   Fruit* pFruitList[4];
   pFruitList[0] = new Apple;
   pFruitList[1] = new Orange;
   pFruitList[2] = new Grape;
   pFruitList[3] = new GrannySmith;

   for( int i = 0; i < 4; i++ )
   {
     pFruitList[i]->Draw();
     delete pFruitList[i];
   }
}
```

The ability to determine object behavior at run time instead of only at compile time is a major improvement over C and provides the polymorphic behavior required by object-oriented development.

Understanding Vtables

Virtual functions allow C++ programs to invoke functions dynamically instead of statically. Other terms for dynamic and static function invocation are *late* binding and *early* binding, referring to binding the function address either at run time or at compile time. Whenever you declare a function as virtual, the compiler adds a pointer to your class structure called the *vptr*, or virtual pointer. The vptr points to a *Vtable* structure that contains the addresses of any virtual functions in your class including its base classes. Figure 2.3 depicts the vptr and the Vtable entries for the following class definition:

```
class Fruit {
protected:
   CString   m_strColor;
public:
```

```
   void      SetColor( CString str ) { m_strColor = str; }
   CString   GetColor() { return m_strColor; }
   virtual   void Draw() = 0;
   virtual   long AvgWeight() = 0;
}

class Apple : public Fruit {
protected:
   long    lAvgWeight;
public:
   // Constructor
   Apple() { lAvgWeight = 10; }
   virtual void Draw() { cout << "I'm an Apple" << endl; }
   virtual long AvgWeight() { return m_lAvgWeight; }
}

class GrannySmith : public Apple {
public:
   virtual void Draw() { cout << "I'm a Granny Smith Apple" << endl; }
}
```

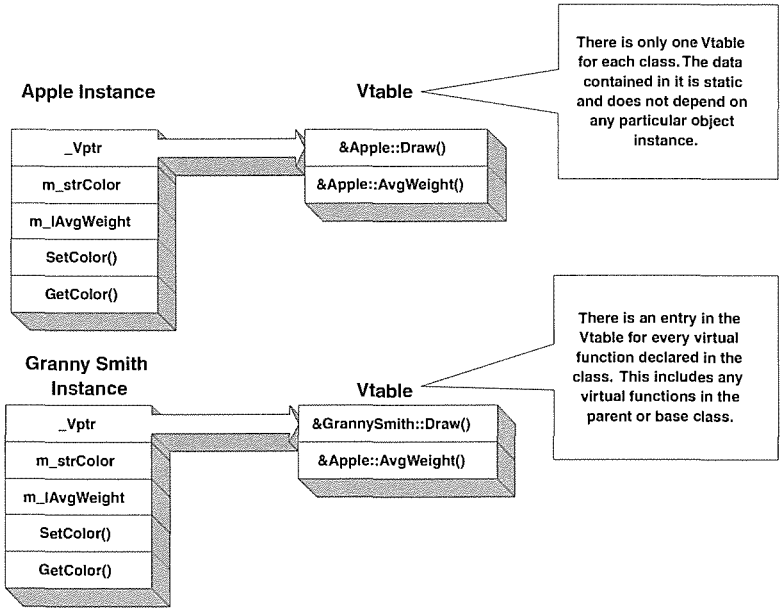


Figure 2.3 Vtable for fruit classes.

The Vtable provides the dynamic binding capability for C++. Examine the following example code:

```
main()
{
   Fruit* pFruit;
   Apple  apple;
   GrannySmith gsApple;

   pFruit = &apple;

   pFruit->Draw();
   cout << "Average weight for an Apple is " << pFruit->AvgWeight() << endl;

   pFruit = &gsApple;
   pFruit->Draw();
   cout << "Average weight for a Granny Smith is "
        << pFruit->AvgWeight()
        << endl;
}
```

This example produces the following output:

```
>I'm an Apple
>Average weight for an Apple is 10
>I'm a Granny Smith
>Average weight for a Granny Smith is 10
```

The example creates `pFruit`, a pointer to the `Fruit` abstract class, and then assigns to it the address of an `Apple` instance. Because `Fruit` is an abstract class, the compiler knows that `pFruit` will point to a dynamic (or polymorphic) type. When the statement `pFruit->Draw()` is encountered, the compiler generates instructions to access the vptr, look up the virtual function by position in the Vtable, and transfer execution to the address contained in the Vtable.

This late binding of function addresses at run time is important to object-oriented languages. Some object-oriented languages—Smalltalk and Java in particular—bind all functions late. Other languages, including C++, leave it up to the developer to decide which functions should bind late. C++ does this for performance reasons. There is overhead in providing the late binding necessary for polymorphic behavior. For every class that has at least one virtual function, a Vtable is needed for the class and a vptr is needed for each instance. The vptr must be initialized for each instance, and there is the run-time overhead of function lookup every time a virtual function is called.

That finishes our review of virtual function capabilities in C++. I've included it here because we will need this background as we begin to discuss COM and Automation in subsequent chapters.

Multiple Inheritance

Multiple inheritance (MI) is one of those philosophical topics that are best not discussed. There are those who say MI is an important part of object-oriented development and that certain problems can be solved (elegantly) only with MI. Others argue that anything you can do with MI can also be done using single inheritance. Indeed, many object-oriented languages, such as Smalltalk and Objective-C, support only single inheritance, but C++ supports multiple inheritance. We won't use multiple inheritance for our purposes, but in the Chapter 4 we will encounter it as we discuss the Component Object Model's concept of component interfaces.

We've discussed the wonderful things that inheritance can do. If inheritance is a wonderful thing, then multiple inheritance must be really impressive. You tell me. In the next example we have two classes: `Speedometer` and `Tachometer`.

```
class Speedometer {
protected:
   int    m_Speed;
   RECT   m_Position;
public:
   int   Mph();
   int   Kph();
   void  Display();
};

class Tachometer {
protected:
   int    m_Rpm;
   RECT   m_Position;
public:
   int    Rpm();
   void   Display();
};
```

Now, we create a new class, `DashBoard`, that combines the features of both classes.

```
class DashBoard : public Speedometer, public Tachometer {
     void  SpeedPos();
     void  TachPos();
   void  Display();
     void  SpeedDisplay();
     void  TachDisplay();
};
```

DashBoard now has "name ambiguity" within itself: both Tachometer and Speedometer have member variables named m_Position. In addition, there is a collision with the member function Display. This ambiguity is one of the primary problems with multiple inheritance. It can be overcome either by not using the ambiguous members or by directly addressing them using the class name:

```
void DashBoard::Display() {
  Speedometer::Display();
  Tachometer::Display();
    }
```

The MFC libraries do not use multiple inheritance at all, primarily because MFC is structured as a hierarchy in which almost every class derives from CObject. Multiple inheritance of MFC objects would imply the inclusion of multiple CObject objects, and this would cause name collisions, or ambiguity, with the MFC Dump function, among others. All this is explained in detail in MFC *TechNote 16*. If Microsoft developed MFC without resorting to multiple inheritance, I think we can infer that most C++ projects can do without it. Later, when we discuss COM interfaces, you'll be tempted to use multiple inheritance. It's a problem begging to be solved with MI, but MFC uses class nesting and class composition instead.

Class Composition

Instead of using single or multiple inheritance, a problem can also be solved by using *composition*, also called *containment* or *embedding*. Class composition involves including instances of other classes within the new class. This approach works best when there is a "has-a" relationship between the various classes. For example, an apple "is-a" type of fruit, so inheritance is appropriate. A fruit tree, on the other hand, "has-a" fruit, so composition is the better object-oriented approach. Instead of using multiple inheritance to combine the needed features, we create a new class by using a combination of inheritance and composition or by using composition alone.

Using the previous dashboard example, let's try to build a dashboard class by using class composition. A dashboard doesn't fit the "is-a" relationship required for inheritance. A dashboard is definitely not a speedometer, but it does fit the "has-a" relationship of class composition. A dashboard "has-a" speedometer and possibly a tachometer. So we implement a DashBoard class as follows:

```
class Speedometer {
public:
     int     Mph();
     int     Kph();
};
class Tachometer {
public:
     int     Rpm();
};
```

```
class DashBoard {
private:
   Speedometer   m_Speedometer;
   Tachometer    m_Tachometer;
public:
   void    SpeedPos();
   void    TachPos();
   void    SpeedDisplay();
   void    TachDisplay();
};
```

When using composition, we can obtain additional flexibility by storing only a pointer to the included class. This class can be an abstract base class and will allow us to "plug in" various derived classes. If we had two types of speedometers—say an analog and a digital type—we could implement the class as follows:

```
class Speedometer {
public:
   virtual void DisplayData() = 0;
};

class AnalogSpeedo : public Speedometer {
protected:
   void GetAnalogData();
public:
   virtual void DisplayData():
};

class DigitalSpeedo : public Speedometer {
protected:
   void GetDigitalData();
public:
   virtual void DisplayData():
};

class DashBoard {
   Speedometer* m_pSpeedometer;
   Tachometer   m_Tachometer;
   // Constructor that takes a speedometer as a parameter
   DashBoard( Speedometer* pSpeedo )
   {
      m_pSpeedometer = pSpeedo;
   }
};
```

```
...
// At run time, create a dashboard with a digital or analog speedometer
// Dynamic determination of created object
DigitalSpeedo* pDigitalSpeedo = new DigitalSpeedo;
AnalogSpeedo*  pAnalogSpeedo = new AnalogSpeedo;

// A digital dashboard

DashBoard* pDigitalDashBoard = new DashBoard( pDigitalSpeedo );
// A analog dashboard
DashBoard* pAnalogDashBoard = new DashBoard( pAnalogSpeedo );
...
```

At run time we can determine what type of speedometer we need for the particular dashboard that we are instantiating. Composition is an effective method of reusing existing classes and providing the flexibility needed to reuse code effectively.

The const Keyword

The C++ `const` keyword is useful for adding rigor to your class implementations. We don't have the space to go into much detail, so I'll hit the high spots with the next example:

```
BOOL Expression::IsNumber( const CString& strToken ) const;
```

Here are two examples of using `const`. `Const` prior to `CString&` indicates, that the `strToken` parameter is constant and cannot (or will not) be modified by the `IsNumber` function. Because we're passing by reference (we'll discuss this next) to increase efficiency, we want to ensure the class user that we will not modify the token that is being passed. The `const` following the function declaration ensures that the `IsNumber` function cannot modify any member variables of the `Expression` class. In other words, the `const` applies to the implicit `this` parameter passed to all member functions. Additionally, it ensures that `IsNumber` will not call any other member functions that can or might modify a member variable—that is, other member functions that aren't declared `const`.

References

The ampersand after `CString` in the `IsNumber` method above is called a reference. This new C++ feature causes much confusion, especially for old C programmers like me. Let's take a quick look at why references are useful when you're developing C++ applications. First, references clean up the syntax when you're working with pointers, and second, their use can greatly increase efficiency when you're passing objects to functions.

A reference behaves just like a constant pointer, but the compiler always provides the dereference operator for you.

```
CString strColor = "Blue Green";   // declare a CString
CString& rStrColor = strColor;     // declare a reference to strColor
CString* const pStrColor = &strColor // declare a const pointer to strColor

cout << "Reference " << rStrColor << endl;
cout << "Dereferenced Pointer " << *pStrColor << endl;
```

Both `cout` lines produce "Blue Green." References are like `const` pointers, because once a reference is initialized, it cannot be reassigned to point to a different object. In fact, a reference must be initialized when it is declared. The next example makes this clear:

```
CString strColor = "Blue Green";   // declare a CString
CString strAnotherColor = "Yellow";  // declare another CString
CString& rStrColor = strColor;     // declare a reference to strColor
CString* const pStrColor = &strColor // declare a const pointer to strColor

// You can't do this
pStrColor = &strAnotherColor

// But you can do this, What do you think this does?
rStrColor = strAnotherColor;

cout << "rStrColor value is " << rStrColor << endl;
cout << "strColor value is " << strColor << endl;
cout << "strAnotherColor value is " << strAnotherColor << endl;

// Produces this output
> rStrColor value is Yellow
> strColor value is Yellow
> strAnotherColor value is Yellow
```

If `rStrColor` behaves like a `const` pointer, why can we assign `strAnotherColor` to it? We can't. The example produces an output of `"Yellow"` not because we were able to reassign the reference, but because the contents of `strAnotherColor` were assigned to `strColor`. In other words, the assignment statement behaves as if it were this:

```
*pStrColor = strAnotherColor;
// or
strColor = strAnotherColor;
```

The assignment operator for `CString` is called, and it copies the contents of `strAnotherColor` to `strColor`. Remember, wherever `rStrColor` occurs (outside of initialization) it really means `strColor`.

References are also useful for increasing the performance of object passing when you're calling functions. The following code must call the `CString` copy constructor and build a copy of the `CString` instance to pass to the `IsNumber` function:

```
// return TRUE if the string is a number
BOOL  Expression::IsNumber( CString strToken )
{
   int nLen = strToken.GetLength();
   for ( int i = nLen - 1; i >= 0 ; i- )
   {
      if (! isdigit( strToken[i] ))
         return FALSE;
   }
   return TRUE;
}
```

This may not seem too costly for `CString` objects, but it can get expensive on large objects. References allow you to use the same syntax as in the preceding example, but instead of a copy, a reference to the object is passed to the function. This technique can greatly increase the performance of an application that passes around large objects. Here's the new function:

```
// return TRUE if the token is a number
BOOL  Expression::IsNumber( const CString& strToken )
{
   int nLen = strToken.GetLength();
   for ( int i = nLen - 1; i >= 0; i- )
   {
      if (! isdigit( strToken[i] ))
         return FALSE;
   }
   return TRUE;
}
```

The syntax is exactly the same when you're accessing an object through a reference, but you must be careful. Because we're passing a reference instead of a copy of the string, the function can modify its contents. However, we're passing by reference for performance reasons. We don't want the function to be able to modify the contents of the parameter. That's why we've added the `const` keyword.

The this Keyword

C++ defines a keyword, `this`, that is a pointer to the instantiated object. This keyword is similar to the `self` keyword in Smalltalk and the `me` keyword in Visual Basic. The `this` pointer is in scope only within nonstatic member functions and is normally used when you're dealing with the copy constructor and when you're overloading operators. As stated earlier, `this` is available only within member functions and operators. In these cases, it is implicitly passed, and thus the following code is equivalent:

```
void SomeClass::SetValue( const short sNewValue )
{
   m_sValue = sNewValue;
}

void SomeClass::SetValue( const short sNewValue )
{
   this->m_sValue = sNewValue;
}
```

You'll use the `this` keyword when building copy constructors and when overriding the assignment (=) operator. We haven't discussed these two topics, so let's do that now.

Copy Constructors

We've discussed the role of class constructors in C++, but a particular type of constructor, called the *copy constructor*, is important in C++ class development. Whenever you develop a new class in C++, you should include a declaration for both a copy constructor and the assignment operator. The compiler's default implementations of the copy constructor and assignment operator are rarely what you want. The compiler performs a bit-wise copy of the instance members. If your class contains any pointers, this default will cause problems. We'll examine the assignment operator in the next section.

The copy constructor is used whenever a copy of an existing C++ object is needed. The class it constructs takes a parameter that is a reference to another instance of the class. Here is an example:

```
// Copy constructor for Apple class
Apple::Apple( Apple& x )
{
   // do the assignment
   m_strName = x.m_strName;
   m_strColor = x.m_strColor;
   ...
}

Apple* pApple = new Apple;
Apple Apple2( *pApple );     // Calls copy constructor
void PrintFruitName( Fruit fruit )
{
  cout << "fruit name is " << fruit.GetName() << endl;
}
PrintFruitName( Apple2 );   // Copy constructor called
```

Overloading Operators

Like member functions, C++ operators can be overloaded on either a class or global level. The assignment operator is similar to the copy constructor and should be declared in every class that you design and implement. The assignment operator is used whenever a instance assignment is performed.

```
Apple* pApple = new Apple;
Apple Apple2 = *pApple;
Apple Apple3, Apple4;

Apple3 = Apple2;     // Assignment operator
// If you return a reference to the left hand side (lhs) you can do this
Apple4 = Apple3 = Apple2;
```

The assignment operator implementation is similar to that of the copy constructor described in the last section. The primary difference is that you typically return a reference to the object being assigned. Here's its implementation.

```
Apple& Apple::operator=( const Apple& rhs )
{
   // Ensure we're not assigning to ourselves
   if ( &rhs == this )
      return *this;

   // do the assignment
   m_strName = rhs.m_strName;
   m_strColor = rhs.m_strColor;
   ...
   return *this;
}
```

I told you that we weren't going to get very deep in C++ details, but I'll give you a couple of sentences on deep versus shallow copies (because it's easy at this point). When making a copy of an object that contains pointers to other objects, what should you do? If you just copy the pointers to the new object, it's a shallow copy, but if you allocate additional memory and replicate any contained objects or data (for both) you've performed a deep copy. The choice is, of course, implementation-dependent. (Don't you hate that phrase?)

Static Class Members

In the constructors example, we encountered the use of `static` within a class declaration. The next example describes a method of keeping count of all instantiated objects of a particular class by using a static member variable. Static member variables are sometimes called *class variables* because they pertain to the class as a whole and not to any specific class instance (and because that's what Smalltalk calls them). The class vari-

able is a single instance that is available to any and all instantiated objects of that class. Because class variables reside outside class instances, there isn't a constructor to initialize them. Like global variables, they must be initialized by a definition:

```
class Fruit {
   ...
private:
   static int m_nCount;
   Fruit();
public:
   virtual ~Fruit();
   ...
   int GetNumFruits() { return m_nCount; }
};

// Initialization (definition) of class variable (static member variable)
int Fruit::m_nCount = 0;

Fruit::Fruit()
{
   m_nCount++;
}
Fruit::~Fruit()
{
   m_nCount--;
    }
```

This example also has a member function, `GetNumFruits`, that retrieves the number of outstanding instances of the `Fruit` class. We would use the function like this:

```
Fruit* pF1 = new Fruit;
cout << "There are "
     << pF1->GetNumFruits()
     << " Fruit objects"
     << endl;

Fruit* pF2 = new Fruit;
cout << "There are "
     << pF2->GetNumFruits()
     << " Fruit objects"
     << endl;
delete pF1;

cout << "There are "
```

```
    << pF2->GetNumFruits()
    << " Fruit objects"
    << endl;
delete pF2;

//  How do we get the number of objects?
```

As you can see, we have a problem determining when there are zero objects. How can we retrieve the number of objects when none is instantiated? Static member functions, or *class functions* or *class methods*, let us declare functions that operate on the class as a whole outside the scope of any particular instance, just like class variables. So if we declare the `GetNumFruits` function as follows, we can always determine the number of fruit objects.

```
class Fruit {
  ...
private:
  static int m_nCount;
  ...
public:
  static int GetNumFruits() { return m_nCount; }
};
```

Here's the syntax for invoking a class function. You can also append the call to a class instance, but it doesn't make sense.

```
Fruit* pF1 = new Fruit;
cout << Fruit::GetNumFruits(); // The preferred method, no ambiguity
cout << pF1->GetNumFruits();   // Also works, but seems a little silly
```

There are many cases when you need variables and functions that operate on the class and not on an instance of a class. The MFC library provides some good examples. Here's an abbreviated look at the MFC `CFile` class:

```
class CFile : public CObject
{
  ...
public:
  static void  Rename( const char* pszFileName, const char* pszNewName );
  static void  Remove( const char* pszFileName );
  BOOL         GetStatus( CFileStatus& rStatus ) const;
  static BOOL  GetStatus( const char* pszFileName, CFileStatus& rStatus );
  ...
};

main() {
```

```
  // Let's rename a file
  CFile::Rename( "c:\\oldname.txt", "c:\\newname.txt" );
  // Now let's delete it
  CFile::Remove( "c:\\newname.txt" );
};
```

The static member functions (class methods) can be used without an instance of the `CFile` class, and this approach is appropriate for functions such as `Rename`, `Remove`, and `GetStatus`. But as you can see, a member function is also provided that is specific to an instance of the `CFile` object. Remember, static member variables and functions should operate on the class as a whole and not on any particular class instance.

That ends our overview of the C++ language. Next, we'll design and develop a small class that we will use in later chapters.

The Problem

Our hypothetical expression evaluator application needs to allow its users to enter an expression in a standard Windows entry field in which they are currently allowed to enter only integer values. In short, we need to provide a simple integer calculator where they may need it, within the entry field itself. This is standard fare in a spreadsheet, but our users want something simple. To start, let's develop a C++ class that provides validation and evaluation of simple algebraic expressions such as these:

```
((1 + 2) * 100 / 5) - 12
5 * 17 - 3 + 3200
(1 + 22) * 7 / 10
```

Only integers and the binary operators +, -, *, /, and parentheses will be supported, but support for additional operators, user variables, and rational numbers shouldn't require much effort. We will use this example throughout the book as we move from a C++ class to an Automation component and eventually to an ActiveX control based on this expression functionality.

Solving the Problem with a Reusable Class

We have a problem statement, so let's design a C++ class that will provide the needed functionality. Designing reusable classes in C++ is a difficult task. In typical development environments, because of project time constraints, developers tend to solve the specific problem first. Then, if there's time, they go back and adapt the software (classes) to be more general so that it can be reused. There are valid reasons for doing it this way. If the problem to be solved is unclear or complex, only development of the specific solution will yield enough understanding to ultimately produce a general solution. Technology changes so rapidly that we really understand the problem only when we've finished solving it. But if we approach the problems as general problems, we will eventually gain the ability to solve them generally first and specifically second. In the long run, this is by far the best approach.

Designing reusable classes in C++ for use by C++ is best done by use of the inheritance mechanisms. The goal is to implement general behavior in a base (possibly abstract) class and then to override certain functionality through the use of virtual functions. Other approaches include designing the base class to allow other derived classes with their own specific behavior to be plugged in. The MFC libraries provide many good examples of these approaches, as you will see when we build components using MFC in later chapters.

As we discussed in Chapter 1, there are many ways to achieve reusability in software development. One approach is by using inheritance. The other technique—reuse by using discrete specialized components—is our primary focus, so we won't spend much time discussing how to design reusable classes for C++ developers. Instead, we will focus on designing C++ classes that can be easily converted into COM-based software components.

There are many books on object-oriented design, but this book is about component development. My purpose here is to help in the design of C++ classes that will eventually be used outside the C++ environment. In many ways, this minimalist approach makes the design process easier because binary standards provide only a subset of the object-oriented facilities provided by C++. We have less to work with, but it simplifies the design process. For example, a major aspect of C++ class design is to decide how, when, and why to overload operators. Binary standards are a long way from defining overloading capabilities at the function level, let alone the overloading of operators. Binary standards allow the standardizing only of the interface to a component and provide only limited (if any) inheritance capabilities. These restrictions require us to be specific in the way we design classes for use as components, and it is quite different from designing classes that will be reused within C++ development. With that said, let's design a class to solve the problem I've outlined.

When we're designing C++ classes that will be used as components, our focus should be on the interface of the class. It's important to focus on the member functions the user will use. These functions define the behavior of our class (and component) and will eventually be exposed through a binary standard interface. For the problem described, I've designed a simple C++ class as follows:

```
class Expression {
public:
   // Constructors and Destructor
   Expression();
   Expression( CString strExp );
   Expression( CString strExp, BOOL bInfix );
   ~Expression();

// Here's the interface
public:
   CString  GetExpression();
   void     SetExpression(CString strExp, BOOL bInfix );
   BOOL     Validate();
   long     Evaluate();
};
```

We're not worried about how to implement this yet. For now, we're interested in how we think the user should interact with our class. The user needs the ability to provide an expression string; `SetExpression` provides this capability. The user also needs to validate and at some point evaluate the expression, so we've provided `Validate` and `Evaluate` functions. We've also provided `GetExpression` in case the user expects us to maintain the storage for the expression (so that he or she doesn't have to). Here's how the class might be used:

```
main()
{
   Expression* pExp = new Expression;
   cout << "Enter an expression: ";
   cin >> strExp;
   pExp->SetExpression( strExp, TRUE );
   if ( pExp->Validate() )
      cout << "The result is " << pExp->Evaluate() << endl;
   else
      cout << "Invalid expression" << endl;

   delete pExp;
}
```

We could have designed the interface in other ways. One option is to have only one function:

```
class Expression {
public:
   // Returns False if Validate fails
   BOOL Evaluate( const CString strExp, BOOL bInfix, long& lResult );
}
```

This would work, but there is at least one problem. Binary standard wrappers, such as COM, don't necessarily support the use of C++ references or pointers. This design would preclude the use of this class as a component because of its use of references.

Next, we'll design the implementation, but first let's review interfaces and implementations.

Interface versus Implementation

It is important to distinguish between an interface and its implementation. We touched on this briefly earlier. The ability to insulate the class user from implementation details is a key advantage of C++. This ability is the essence of object-oriented encapsulation. The details of how an object implements its functionality are hidden or encapsulated from the class user. Thus, the implementation can change without affecting the user at all, although that isn't exactly true when C++ is used. When a C++ class library changes, it must be recompiled. If any users of the class also want these new features, they must at least relink or recompile their applications, depending on the nature of the changes. Binary standards overcome this limitation of C++ and

allow dynamic changing of the method implementation without affecting the component user; there is no need to recompile or relink. Figure 2.4 illustrates the difference between the implementation and interface in the context of a C++ class.

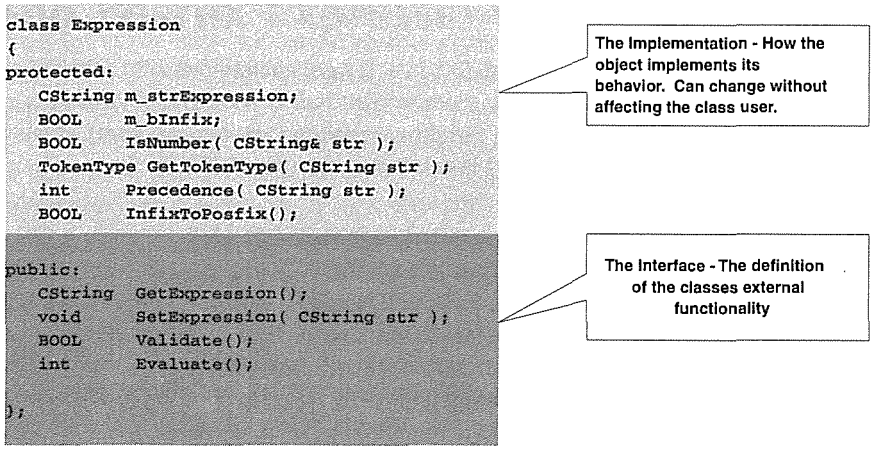


Figure 2.4 Class implementation and interface.

The public methods of a C++ class are what describe, to the external user, the capabilities and behaviors of the class. This public interface is also described as an interface *contract*. It's a contract in the sense that once it is defined and exposed for use by clients of the class, it shouldn't change. If the interface must change, it should be changed by augmentation; only new methods should be added. The old methods should not change. This arrangement ensures that users of the original interface will not be adversely affected by additions (upgrades) to the class interface. The idea of an interface contract is also important to component development. Additions and changes to a component interface should be handled in the same manner as they are in C++ implementations.

The Expression Class Implementation

We've defined how the class user will interact with our class, and now it's time to develop the implementation. Here's the complete declaration of the `Expression` class:

```
class Expression
{
protected:
   // Similar to a static, enum values are for the class as a whole
   enum TokenType
   {
```

```
      BogusToken,
      OperatorToken,
      OpenParenToken,
      CloseParenToken,
      NumberToken
   };

// Implementation variables
protected:
   CString        m_strExpression;
   BOOL           m_bInfix;

// Implementation functions
protected:
   BOOL             IsNumber( const CString& strToken );
   TokenType        GetTokenType( const CString& strToken );
   int              Precedence( const CString& strToken );
   BOOL             InfixToPostfix();

public:
   // Constructors
   Expression();
   Expression( CString str, BOOL bInfix );
   // Destructor
   ~Expression();

   // Copy constructor
   Expression( Expression& x );

   // Assignment operator
   Expression& operator=( Expression& rhs );

public:
   CString  GetExpression();
   void     SetExpression( CString strExp, BOOL bInfix );
   long     Evaluate();
   BOOL     Validate();
};
```

The `Expression` class also needs the services of a `Tokenizer` and a `Stack` class, which we will cover in detail later in the chapter. We're using the top-down approach to solve the problem. Our implementation contains four member functions. `InfixToPostfix` converts a standard infix expression to postfix for easier evaluation. `IsNumber` determines whether an expression token is a number. `GetTokenType` returns the

type of the expression token, and `GetPrecedence` returns the evaluation precedence of the passed token. In the next section we'll describe each of these functions.

Infix and Postfix Expressions

We're all familiar with infix expressions (e.g., (1 + 2) / 3), but there are some difficulties in evaluating them. The problem with infix expressions is that they're more difficult programmatically to evaluate than their postfix equivalents. (Infix expressions require parentheses to remove ambiguity in the expression, but postfix expressions do not.) Table 2.1 shows some examples of infix and equivalent postfix expressions.

Table 2.1 Examples of Infix and Postfix Expressions

Infix	Postfix
(1 + 3) * 5 - 10	1 3 + 5 * 10 -
100 / 3 * 5 + 15 * 10	100 3 / 5 * 15 10 * +
900 - 45 + 10 / 5 * 10	900 45 - 10 5 / 10 * +

Postfix expressions place the binary operators after the two operands, removing any ambiguity in infix expressions. There is no need for parentheses. This technique also makes programmatic evaluation of the expression easy. This book isn't about expression evaluation, so we won't go into the details. If you're interested, complete coverage of this topic is available in *Intermediate Problem Solving and Data Structures*, by Helman and Veroff. (See the Bibliography.)

Here's our implementation of `InfixToPostfix`:

```
// Convert the expression from infix to postfix form.
// Use a local (on the stack) instance of the Tokenizer class
BOOL Expression::InfixToPostfix()
{
   CStringStack stack;
   Tokenizer tokenizer;
   CString strToken;
   CString strPostfix;
   CString strTop;
   CString strPop;

   tokenizer.SetString( m_strExpression );
   // Tokenize the expression
   tokenizer.Tokenize();
```

```
// while we have more tokens
while( tokenizer.GetToken( strToken ) )
{
   switch( GetTokenType( strToken ) )
   {
      // If we have a number, append it to the new postfix string
      case NumberToken:
         strPostfix += strToken;
         // Delimit the number tokens by appending an extra space
         strPostfix += " ";
         break;

      // If we encounter an open paren '(', push it on the stack
      case OpenParenToken:
         stack.Push( strToken );
         break;

      // If we encounter a close paren ')'
      case CloseParenToken:
         if ( stack.Peek( strTop ) )
         {
            // While we haven't found an open paren and
            // the stack is not empty
            while( strTop.Compare( "(" ) )
            {
               stack.Pop( strPop );
                // Pop the next element and append it to the postfix string
               strPostfix += strPop;
               if (! stack.Peek( strTop ) )
                  break;
            }
         }
         // Pop the paren off the stack, we don't need it
         stack.Pop( strPop );
         break;

      case OperatorToken:
         // While there is something in the stack
         while( ! stack.IsEmpty() )
         {
            // Peek at the next element on the stack
```

```
            stack.Peek( strTop );
            // If the top element on the stack is NOT an open paren, and its
            // precedence is greater than or equal to our current token, then
            // pop it off the stack and append it to our postfix string.
            // Exit the loop when the stack is empty, or we encounter an
            // element that is the open paren or whose precedence is lower
            // than our current token.
            if ( strTop.Compare( "(" ) &&
                 Precedence( strTop ) >= Precedence( strToken ))
            {
               stack.Pop( strPop );
               strPostfix += strPop;
               stack.Peek( strTop );
            }
            else
               break;
         }
         // Push the token that caused us to exit back
         // on the stack
         stack.Push( strToken );
         break;

      }
   }
   // Empty the stack and append the elements
   // to our postfix string.
   while(! stack.IsEmpty() )
   {
      stack.Pop( strPop );
      strPostfix += strPop;
   }

   // Everything is Ok
   m_strExpression = strPostfix;
   TRACE1( "New Postfix expression is %s\n", strPostfix );
   return FALSE;
}
```

Three other support functions are needed for the Expression class: `IsNumber`, `GetToken`, and `Precedence`. Here they are:

```
// Determine if the token is a number.
```

```
// Return TRUE if it is, else FALSE.
BOOL Expression::IsNumber( const CString& strToken )
{
   int nLen = strToken.GetLength();
   for ( int i = nLen - 1; i >= 0; i– )
   {
      if (! isdigit( strToken.GetAt(i) ))
         return FALSE;
   }

   return TRUE;
}

// Get the token type of the passed token
Expression::TokenType Expression::GetTokenType( const CString& strToken )
{
   if ( strToken.Compare( "(" ) == 0)
      return( OpenParenToken );
   else if ( strToken.Compare( ")" ) == 0 )
      return( CloseParenToken );
   else if ( strToken.FindOneOf( "+-*/" ) != -1 )
      return( OperatorToken );
   else if ( IsNumber( strToken ) )
      return( NumberToken );
   else
      return( BogusToken );
}

// Return the precedence of the operator
// 2 is the highest precedence
int Expression::Precedence( const CString& strToken )
{
   if ( strToken.FindOneOf( "*/" ) != -1 )
      return( 2 );
   else if ( strToken.FindOneOf( "+-" ) != -1 )
      return( 1 );
   else
      return( 0 );
}
```

To finish, here are the member functions that implement the interface for the `Expression` class, the public constructors, and the assignment operator:

```
// Constructors
Expression::Expression()
{
   // Default to infix
   m_bInfix = TRUE;
}

Expression::Expression( CString str, BOOL bInfix )
{
   m_strExpression = str;
   m_bInfix = bInfix;
}

// Copy constructor
Expression::Expression( Expression& x )
{
   m_bInfix = x.m_bInfix;
   m_strExpression = x.m_strExpression;
}

// The assignment operator
Expression& Expression::operator=( Expression& rhs )
{
    // If we're assigning to ourselves just return
    if ( this == &rhs )
       return *this;

    m_bInfix = rhs.m_bInfix;
    m_strExpression = rhs.m_strExpression;

    return *this;
}

// The destructor
Expression::~Expression()
{
}

// Get the current expression.
CString Expression::GetExpression()
{
   return m_strExpression;
}
```

```
// Set the expression to evaluate, set the bInfix flag to
// TRUE if it is an infix expression, FALSE for postfix
void Expression::SetExpression( CString strExp, BOOL bInfix )
{
   m_strExpression = strExp;
   m_bInfix = bInfix;
}

//
// Validate an infix expression by balancing the parentheses
// and checking for invalid tokens. Return TRUE if the expression
// is valid, else FALSE
BOOL Expression::Validate()
{
   CStringStack stack;
   Tokenizer tokenizer;
   CString strToken;
   CString strTop;

   tokenizer.SetString( m_strExpression );

   // Tokenize our expression
   tokenizer.Tokenize();

   // Check for validity
   while( tokenizer.GetToken( strToken ) )
   {
      switch( GetTokenType( strToken ) )
      {
         case NumberToken:
         case OperatorToken:
             break;

         case OpenParenToken:
             stack.Push( strToken );
             break;

         case CloseParenToken:
             if ( stack.IsEmpty() )
             {
               TRACE0( "Too many closing parens\n" );
               return FALSE;
             }
```

```
            else
              stack.Pop( strTop );
            break;

         default:
            // Invalid operator type
            TRACE1( "Invalid Operator Type %s\n", strToken );
            stack.Clear();
            return FALSE;
      }
   }

   // If there's something on the stack, there's a paren mismatch
   if (! stack.IsEmpty() )
   {
      TRACE0( "Too many open parens\n" );
      return FALSE;
   }

   return TRUE;
}

//
// Actually evaluate the expression. If the expression is infix,
// convert it to postfix first. Return the result of the expression.
// You should call validate before calling this function.
long Expression::Evaluate()
{
   CStringStack stack;
   Tokenizer tokenizer;
   CString strToken;
   CString strOperand1;
   CString strOperand2;
   CString strResult;
   long lResult;
   char szTemp[32];

   if ( m_bInfix )
   {
      TRACE0( "Converting to postfix\n" );
      InfixToPostfix();
   }
```

```
tokenizer.SetString( m_strExpression );
tokenizer.Tokenize();

// While there are tokens to process

while( tokenizer.GetToken( strToken ) )
{
   switch( GetTokenType( strToken ) )
   {
      case NumberToken:
         stack.Push( strToken );
         break;

      // If we have an operator, pop the next two elements as they
      // will be the operands. Use the binary operands and evaluate
      // them. Push the result onto the stack for the next operator.
      case OperatorToken:
         stack.Pop( strOperand2 );
         stack.Pop( strOperand1 );
         if ( strToken.Compare( "+" ) == 0)
             lResult = atol( strOperand1 ) + atol( strOperand2 );
         else if ( strToken.Compare( "-" ) == 0)
             lResult = atol( strOperand1 ) - atol( strOperand2 );
         else if ( strToken.Compare( "*" ) == 0)
             lResult = atol( strOperand1 ) * atol( strOperand2 );
         else if ( strToken.Compare( "/" ) == 0)
         {
             // If division by zero is attempted
             // clear the tokens and continue. This
             // will exit the while
             if ( atol( strOperand2 ) == 0 )
             {
                 tokenizer.ClearTokens();
                 TRACE0( "Division by Zero\n" );
                 continue;
             }
             lResult = atol( strOperand1 ) / atol( strOperand2 );
         }
         sprintf( szTemp, "%ld", lResult );
         // Push the result onto the stack
         stack.Push( szTemp );
```

```
            break;

         default:
            TRACE1( "Invalid operator type %s\n", strToken );
      }
   }

   // When we're all finished, there is one element
   // on the stack and it is the final result
   stack.Pop( strResult );

   // Convert the result to a long and return it
   return atol( strResult );
}
```

A Tokenizer Class

To convert and evaluate algebraic expressions, we need a way to break the expression string into discrete expression elements, or *tokenize* it. Let's design a tokenizer class to do this. Remember, one way to achieve reusable classes is to break the problem into small pieces and implement each piece as an object. Small generic objects make reusability more attainable when you're using C++.

What follows is our `Tokenizer` class. It takes a string either during construction or later using `SetString`. When the user calls `Tokenize`, it parses the string into its tokens and stores them in a linked list (`CStringList` again). Then the user can iteratively retrieve (or peek at) the tokens.

```
/////
// Tokenizes an algebraic expression string
/////
class Tokenizer
{
protected:
   char         m_szBuffer[256];
   CStringList  m_TokenList;

public:
   Tokenizer();
   Tokenizer( const CString& strString );
   ~Tokenizer();

public:
   void     SetString( const CString& str );
   short    Tokenize();
   BOOL     GetToken( CString& str );
```

```
   BOOL     PeekToken( CString& str );
   void     ClearTokens();

};

//////
// Tokenizer class members
//////
// Constructors
Tokenizer::Tokenizer()
{
}

// Construct a tokenizer with a default string
Tokenizer::Tokenizer( const CString& strString )
{
   strcpy( m_szBuffer, strString );
}

// When the tokenizer is destroyed, make sure
// that all of the strings are deallocated
Tokenizer::~Tokenizer()
{
    ClearTokens();
}

// Set the string to tokenize
void Tokenizer::SetString( const CString& str )
{
   strcpy( m_szBuffer, str );
}

// Tokenize the string into its discrete elements
short Tokenizer::Tokenize()
{
   char *pChar;
   short sCount = 0;
   char szTemp[128];
   char *pTemp;

   // Make sure the token list is empty
   ClearTokens();

   // Get a pointer into the string to tokenize
   pChar = m_szBuffer;
```

```
// While we haven't encountered the
// NULL termination character
while( *pChar )
{
   // skip any spaces
   if ( *pChar == ' ' )
   {
      pChar++;
      continue;
   }

   // Get a pointer into a temp storage
   // for each token
   pTemp = szTemp;
   switch( *pChar )
   {
      // digit
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
      case '0':
         // While the char is a digit and
         // it's not the terminating NULL
         while( pChar && isdigit( *pChar ) )
         {
            *pTemp = *pChar;
            pTemp++; pChar++;
         }
         // Back up one char
         pChar--;
         // Terminate the number string
         *pTemp = '\0';
         break;

      default:
         // All other tokens are one character
```

```
            // assign it to szTemp and NULL terminate it
            *pTemp = *pChar;
            pTemp++;
            *pTemp = '\0';
            break;

      }
      // Count the number of tokens
      sCount++;

      TRACE1( "Adding token %s\n", szTemp );

      // Add the token to the CStringList
      m_TokenList.AddHead( szTemp );

      // Move to the next character in the Token string
      pChar++;
   }

   // return the number of tokens found
   return sCount;
}

// Get the next token in the list. Return
// FALSE if the list is empty, TRUE otherwise.
BOOL Tokenizer::GetToken( CString& str )
{
   if ( m_TokenList.IsEmpty() )
      return FALSE;

   str = m_TokenList.RemoveTail();
   return TRUE;

}

// Clear all of the tokens in the list
void Tokenizer::ClearTokens()
{
    m_TokenList.RemoveAll();
}

// Peek at the top token
// Return FALSE if there are no tokens
BOOL Tokenizer::PeekToken( CString& str )
{
   if ( m_TokenList.IsEmpty() )
```

```
      return FALSE;

   str = m_TokenList.GetTail();
   return TRUE;
}
```

A Stack Class

The `Expression` class needs the services of a stack, so let's build a stack class. Stacks are normally implemented using arrays, but we want to build a stack that allows pushing of an arbitrary number of elements so that we have no intrinsic limit on the length of an expression. Using a static array would limit the number of elements that could be stored; in addition, the MFC libraries provide a linked-list class that we can reuse by inheriting its capabilities. Should we inherit this functionality from the `CObList` class? Let's take a look.

The following code shows a first attempt at building a `CString` stack. It's implemented by inheriting the methods from the MFC `CStringList` class. `CStringList` is like the `CObList` class, but it uses `CStrings` as the elements of the list. The "is-a" relationship seems to fit. A stack "is-a" kind of linked list, isn't it? Building a stack class that would support multiple data types would also be nice. The best way to do this would be to use C++ templates, but our purpose here is to illustrate the pros and cons of using inheritance or composition when building new classes.

```
class CStringStack : public CStringList
{
public:
   void   Push( CString );
   BOOL   Pop( CString& );
   BOOL   IsEmpty();
   BOOL Peek( CString& );
};

void CStringStack::Push( CString str )
{
   AddHead( str );
}

// Pop an element
BOOL CStringStack::Pop( CString &str )
{
   if (! IsEmpty() )
   {
      str = RemoveHead();
      return FALSE;
   }
```

```
   return TRUE;
}

BOOL CStringStack::IsEmpty()
{
   return CStringList::IsEmpty();
}
BOOL CString Stack::Peek( CString& Str )
{
   if(! IsEmpty() )
   {
      str = GetHead();
      return TRUE;
   }
   return FALSE;
}
```

This implementation works fine. In fewer than 40 lines of code, we've implemented a useful stack that supports an arbitrary number of `CString` elements. Inheritance is great! We've just reused hundreds of lines of MFC library code. But there's a problem. Here's how the class user might implement the code:

```
main()
{
   CStringStack stack;    // Get a stack

   stack.Push("One");     // OK
   stack.Push("Two");     // OK
   stack.Push("Three");   // OK

   POSITION pos = stack.FindIndex( 1 ); // Oh-oh, what's going on here?
   // Here's a way to get at any element in the stack!
   cout << "The middle element on the stack is " << stack.GetAt( pos ) <<
          endl;

   // Mess up the stack
   stack.RemoveAt( pos );
}
```

As you can see the user of the stack class can violate and corrupt our stack implementation. If we're going to provide a solid class implementation, we can't allow this. The encapsulation of the stack object is not complete. What happened? When we decided to use inheritance to build the stack class, we neglected to remember that all protected and public member functions of the base class are inherited. This means that all the linked-list functions—such as `Find`, `FindIndex`, `InsertBefore`, `GetAt`, and so on—are available to our `CStringStack` user. Somehow we must hide the linked-list–specific functions from `CStringStack` users

so that they don't hurt themselves (or us). We could do something like the following to keep users from calling the linked-list functions:

```
class CStringStack : public CStringList
{
protected:                  // Override inherited member functions
   void   FindIndex() {}   // Not implemented
   void   GetAt()   {}     // Not implemented
   ...                     // For all linked-list function we want to hide
public:
   void   Push( CString& );
   BOOL Pop( CString& );
   BOOL IsEmpty();
   BOOL Peek ( CString& );
}
```

Now when stack users try to use the linked-list routines, they will get a parameter mismatch error from the compiler, because the linked-list functions expect and return specific parameters. But we've hidden them by effectively overriding them with do-nothing functions. This isn't the best solution; it requires additional code and is basically a kludge. Another approach is to use private inheritance, but this would also hide those functions that we want to expose publicly (such as `IsEmpty`). Let's try another approach.

Maybe a stack does not really have an "is-a" relationship with a linked list. It seems that it doesn't, because a linked list has functions that can violate the integrity of a stack. But a stack could have a "has-a" relationship with a linked list. We can implement a stack that "has-a" linked list. This means that we will use class composition instead of inheritance to implement our stack:

```
//
// A Stack class that supports CStrings
//
class CStringStack
{
protected:
   CStringList m_StringList;

public:
   // Default constructor
   CStringStack();
   // Copy constructor
   CStringStack( CStringStack& stack );
   // Destructor
   ~CStringStack();

   // Assignment operator
```

```
   CStringStack& operator=( const CStringStack& lhs );

// The interface
public:
   void     Push( Cstring& );
   BOOL     Peek( CString& );
   BOOL     Pop( CString& );
   BOOL     IsEmpty();
   void     Clear();
};

// Default constructor
CStringStack::CStringStack()
{
}

// Copy constructor
CStringStack::CStringStack( CStringStack& stack )
{
   // Copy the stack elements
   POSITION pos = stack.m_StringList.GetHeadPosition();
   while( pos )
   {
      CString strElement = stack.m_StringList.GetNext( pos );
      // Add them in the reverse order by using AddTail
      m_StringList.AddTail( strElement );
   }
}

// When we destroy the CStringStack, ensure
// that the linked list of strings is deallocated
CStringStack::~CStringStack()
{
   Clear();
}

// assignment operator

CStringStack& CStringStack::operator=( const CStringStack& rhs )
{
   // If we're assigning to ourselves just return
   if ( this == &rhs )
      return *this;
```

```
   // remove the elements of the target stack
   m_StringList.RemoveAll();
   // Now move all of the elements
   POSITION pos = rhs.m_StringList.GetHeadPosition();
   while( pos )
   {
      CString strElement = rhs.m_StringList.GetNext( pos );
      // Add them in the reverse order by using AddTail
      m_StringList.AddTail( strElement );
   }
   return *this;
}

// Add an element to the top of the stack
void CStringStack::Push( Cstring& str )
{
   m_StringList.AddHead( str );
}

// Remove an element from the stack. If the
// stack is empty return TRUE, else FALSE
BOOL CStringStack::Pop( CString &str )
{
   if (! m_StringList.IsEmpty() )
   {
      str = m_StringList.RemoveHead();
      return FALSE;
   }
   else
      return TRUE;
}

// Return TRUE if the stack is empty
// This is easy, we defer to the CStringList
// IsEmpty method.
BOOL CStringStack::IsEmpty()
{
   return m_StringList.IsEmpty();
}

// Peek at the top element. If the stack
// is empty return TRUE, else FALSE
```

```
BOOL CStringStack::Peek( CString& str )
{
   if (! m_StringList.IsEmpty() )
   {
      str = m_StringList.GetHead();
      return TRUE;
   }
   else
      return FALSE;
}

// Remove any elements in the stack by
// clearing the strings in the CStringList
void CStringStack::Clear()
{
   m_StringList.RemoveAll();
}
```

There are only two differences between the implementations. The new `CStringStack` class has a member variable `m_StringList` of type `CStringList`, and all the member functions must explicitly access this variable when performing linked-list functions. Now that we have encapsulated the linked-list functionality within our stack class and expose only those functions that we deem appropriate, users can no longer access functions that violate our concept of a stack. Composition is just as important as inheritance to our goal of reusability.


NOTE

On the accompanying CD-ROM, there is an example for Chapter 2. It is a Win32 console application that uses the MFC libraries that we'll discuss in Chapter 3. The example is in the \EXAMPLES\CHAP2\EXPRESS subdirectory and contains the **EXPRESS.H** and **EXPRESS.CPP** files that we have developed in this chapter. The directory also contains the appropriate Visual C++ make file, so it is easy to modify. We will use this example in the next few chapters as we convert it to use the COM binary standard. It would be beneficial at this point to become familiar with the Expression example.

Summary

That wraps up our review of the C++ language and the various object-oriented methods used for designing reusable C++ classes. We also discussed how we might design C++ classes so that their functionality can be exposed, using a binary standard, to other languages and processes. In the next chapter, we'll review Visual C++ and the Microsoft Foundation Class libraries.

Chapter 3

Visual C++ and the MFC Libraries

Throughout this book we will use Visual C++ and its application framework, the Microsoft Foundation Class (MFC) libraries, for the sample code and application examples. In this chapter we'll take a quick look at the Visual C++ environment, what the MFC libraries are, and how they can help with the building of COM-based components. We won't actually start using COM until the next chapter. This chapter will introduce the tools we will use as we explore Microsoft's COM, OLE, and ActiveX using MFC. We will get this experience by developing a simple MFC application to test the `Expression` class we developed in Chapter 2. For a more exhaustive treatment of Visual C++ and the MFC libraries, I recommend *The Revolutionary Guide to MFC 4 Programming with Visual C++* by Mike Blaszczak and *Inside Visual C++*, third edition, by David J. Kruglinski. Once you have a handle on MFC and you want to dig into the details, pick up a copy of *MFC Internals*, by Scot Wingo and George Shepherd.

Win16 versus Win32 Development

The decision whether to develop Win16 (Windows 3.x) or Win32 (Windows 95 and Windows NT) applications gets easier every day. By the time you read this, Microsoft's Windows 95 will have been generally available for more than a year. According to the latest sales figures I've heard, it is outselling Windows 3.x by a factor of five. I'm a big advocate for doing all new development in Win32. Windows 3.x applications will be replaced by more robust Win32-based ones, and the limitations of 16-bit development will eventually fade away. The future lies with the Win32 API, so the applications developed in this book use the 32-bit version of Visual C++. In the previous edition, we also provided 16-bit implementations of the controls, but Microsoft is no longer keeping the 16-bit and 32-bit versions of MFC in sync. Many of the new features of COM and ActiveX that we will explore in this book do not have an implementation in the 16-bit Windows environment, so we will focus purely on 32-bit development. There is a 16-bit version (1.52c) of Visual C++ and MFC that supports the development of COM and ActiveX-based applications, but it is lacking in several areas.

Visual C++

We will use the integrated development environment (IDE) of Visual C++, called Developer Studio, throughout this book. If you've been developing Windows software for a few years, as I have, you're probably skeptical of integrated development environments. If you're like most SDK developers, all you need is a good text-based editor, hand-coded make files, and the various command-line utilities to build solid Windows applications. I agree, but there are features of the Visual C++ IDE that can be a great help, particularly if you are using the MFC libraries. You don't have to use the IDE to use the MFC libraries, but by using it initially, you'll learn more quickly. Later, after you understand MFC inside and out, you can go back to using your favorite editor and the command-line utilities. But you'll soon miss the powerful **F1** key, the AppWizard, and the ClassWizard.

One of the reasons Visual C++ is such a powerful Windows development tool is its included application framework: the MFC libraries. Before we get started, let's look at what application frameworks can provide.

Application Frameworks

The MFC libraries can be described as an *application framework*. An application framework provides an abstracted, high-level view of the underlying operating system, or application environment (e.g., Windows). The primary purpose of application frameworks is to make the developer more productive. An application framework's goals are similar to those of C++: to hide the mundane details of programming within class libraries so that developers need not continually deal with trivial details. A second goal of application frameworks is to provide platform independence. An application framework can separate the details of a particular platform (Windows, OS/2) at an abstracted level within the framework. At the framework API or C++ class level, the details of the underlying target platform can be hidden. This arrangement allows the developer to *target* the application framework. If the developer adheres to the rules of the framework, the resulting source is portable among the various platforms supported by the framework. It also serves the first purpose of increasing developer productivity.

As with most tools, application frameworks have drawbacks. First, there is a significant learning curve involved in becoming familiar with an application framework. It takes time, and the knowledge is not directly transferable to other frameworks. Spending two years developing applications with Borland's OWL framework will not help you very much when your boss tells you to switch to Visual C++ and MFC.

Application frameworks also have *feature lag*. The implementation of new platform-specific features does not occur at the same time that it does for the underlying operating system. For example, MFC's ActiveX classes do not currently support the development of ActiveX controls that completely support the OLE Control 96 specification, but such development is supported via direct COM/ActiveX API calls. So when you're using a framework, be prepared either to wait for new features to be implemented in the framework or to go around the framework and implement needed features using the explicit API calls, possibly providing your own subclasses until the framework is updated.

The Microsoft Foundation Class Libraries

MFC's implementation provides only a thin layer of abstraction above the Windows API. This arrangement has caused some criticism within the industry, because it requires the programmer to understand many of the esoteric Windows constructs to use the libraries effectively. Depending on your perspective, this requirement can be a benefit or a stumbling block. Experienced Windows developers already understand the underlying Windows API, so the MFC libraries quickly enhance their productivity. Others who have used other application frameworks complain that MFC does not *hide*, or abstract, the details of Windows sufficiently. If you don't already have Windows development experience, the MFC libraries can be difficult to learn. Many developers moving from other platforms, such as OS/2, may get a triple whammy. To use MFC, they must also learn Windows, C++, and the libraries, and that is enough to cause many to consider a new career.

I think that Microsoft did it right when it developed the MFC libraries. The abstraction is at just the right level. It increases productivity but doesn't compromise application performance in the process. If you don't agree, there are other options.

Borland's application framework, Object Windows Library (OWL), which comes with Borland C++, provides a higher level of abstraction. This allows implementation of the framework on disparate GUI environments. Programs written using OWL can be ported among the various Windows environments and OS/2, but not the Macintosh platform, which MFC supports. IBM has its Open Class libraries, which are part of its Visual Age series of products. The Open Class framework is abstracted at a level that allows fairly direct movement between OS/2 and various Windows platforms. However, at the time of this writing, IBM's Open Class libraries provide very little support in the area of COM, OLE, and ActiveX.

Choosing a specific framework is difficult. The MFC libraries provide a great deal of functionality and support target platform features that other frameworks do not support. Specifically, other frameworks lack support for our primary goal: ActiveX controls. This is a very important difference, at least for our purposes.

In the scope of platform, or target, portability, the MFC libraries support easy movement of MFC source code among the various Windows platforms: Windows 3.x and Windows 95 on Intel hardware, and Windows NT on Intel, MIPS, Alpha, and PowerPC hardware. However, as I mentioned earlier, because of their differences, it is getting more difficult to move between the 16-bit and 32-bit Windows environments with MFC. Versions of MFC are also available for various flavors of UNIX as well as for Apple's System 7 operating environment. OS/2 is a target that the MFC libraries currently do not support.

The Internet—and its heterogeneous environment of Windows, UNIX, Solaris, OS/2, and Macintosh machines—makes multiplatform support very important. Microsoft has stated publicly that it is committed to providing the COM and ActiveX technologies on most of these platforms. It has also announced that the COM and ActiveX technologies will be handed over to an open standards body so that it will be available to all vendors that want to implement the technology on their platforms. Microsoft, however, is committed to providing the best implementation of the standard.

An MFC Application that Evaluates Expressions

Before we jump into the Component Object Model and ActiveX, let's build a quick MFC-based application to test the `Expression` class we developed in the last chapter. This application will give you a chance to get familiar with the Visual C++ IDE and introduce you to the use of the MFC libraries. We will augment this project as we move through the various chapters. For starters, we'll build a simple Windows single-document interface (SDI) application to try out our expression evaluator class. So fire up Visual C++ and let's get to work.

Using AppWizard

The Visual C++ AppWizard allows a developer to quickly build an MFC application from scratch. It provides a minimal application based on the options picked during the dialog with AppWizard. The resulting application will compile, link, and run, but not much else. Its purpose is to create the various source, resource, and project files necessary to build a Windows application using MFC. Once you run AppWizard to create a particular application, you cannot run AppWizard again to modify an existing application. The purpose of AppWizard is to quickly generate a template application on which to build. This aim is different from that of the usual application generators, which generate a complete application and allow subsequent modifications through the tool.

To start building the Chapter 3 example, start Visual C++ and invoke **New** from the File menu. The New dialog box will display; pick **Project Workspace** and click **OK**, and you will see the dialog box shown in Figure 3.1.

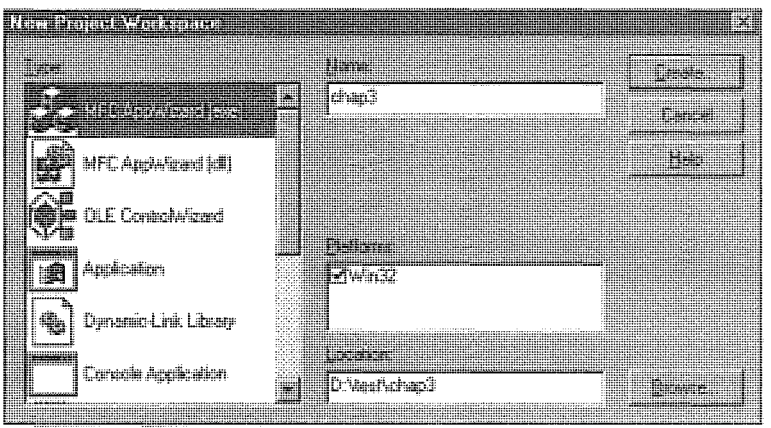


Figure 3.1 New Project Workspace dialog box.

Type **chap3** in the **Name** entry field as shown. Make sure that the **Type** is **MFC AppWizard (exe)** and then click **Create**. This action brings up the AppWizard dialog box shown in Figure 3.2.

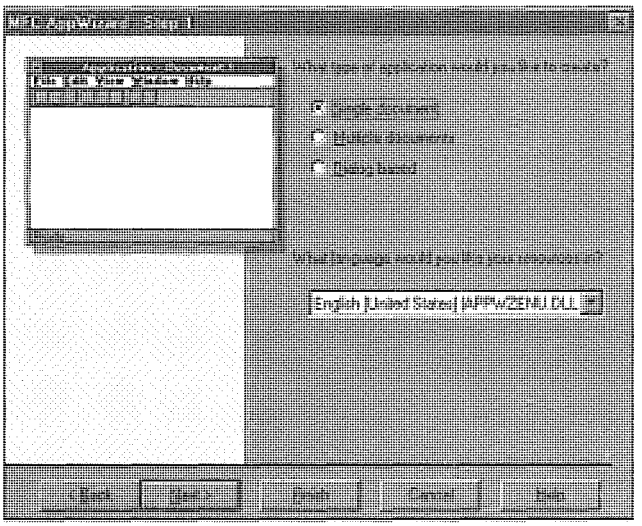


Figure 3.2 AppWizard dialog box, step 1.

For our initial projects we will use Windows single-document interface applications because they aren't as complex as their multiple-document interface (MDI) counterparts (remember, our focus is on COM-based components). The dialog box–based option produces an application that does not use the document/view architecture (we'll discuss this shortly) but instead uses a simple dialog box as the main window. We could have used this option for our simple example, but we will use the document/view architecture in several examples, and now is a good time to discuss it. Later, we will develop an example with MFC's dialog box–based support. Click the **Next** button to continue.

The next dialog box concerns MFC database support. We're not interested in that for now, so click **Next** again. The OLE Support dialog box is next. We're not going to include OLE support just yet, so choose **None** for OLE Compound Document support and **No, Thank you** for OLE (ÁctiveX) Automation support. Click **Next** again to move to the Application Features dialog box shown in Figure 3.3.

To reduce the complexity of the code that AppWizard will generate for us, we've turned off all the features except **Initial status bar** and **3D controls**. Once you've done this, click the **Advanced** button to get the dialog box shown in Figure 3.4.

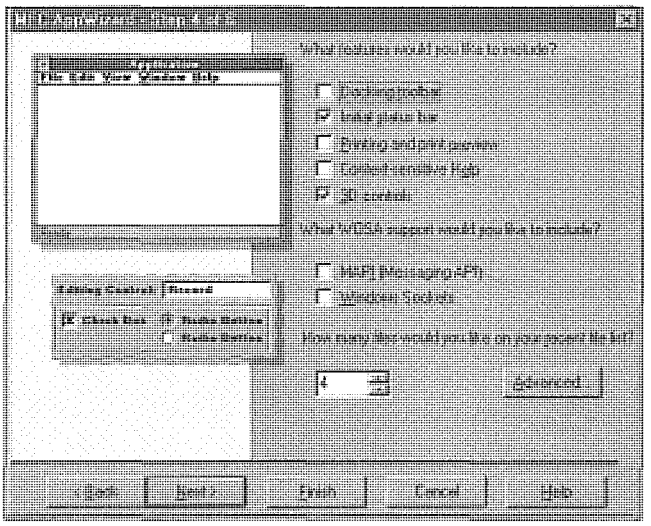


Figure 3.3 AppWizard dialog box, step 4.

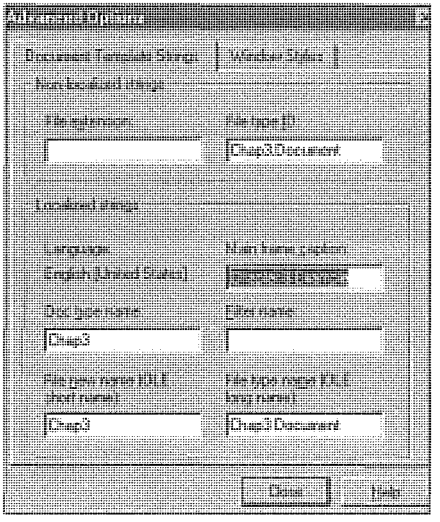


Figure 3.4 Advanced Options dialog box.

Change the caption and check the other features as shown in Figure 3.4, close the dialog box, and click **Next** to move to the Source Options dialog box shown in Figure 3.5.

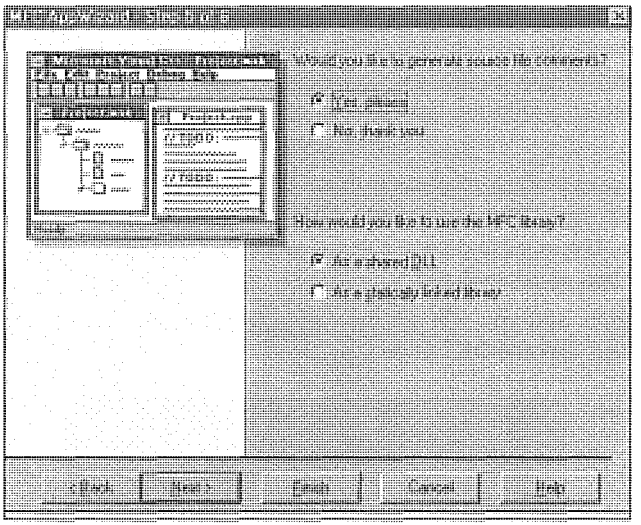


Figure 3.5 AppWizard dialog box, step 5.

I like comments in my code so I answer **Yes, please** to the first option. The next option, **How would you like to use the MFC library?**, allows you to either statically link with the MFC libraries or use the shared DLL version. The DLL option greatly reduces the size of your executable file, but you must distribute **MFC40.DLL** and any supporting files. This isn't a big deal, because most systems will already have these files present. We'll use the **shared DLL** option.

The final AppWizard step is shown in Figure 3.6.

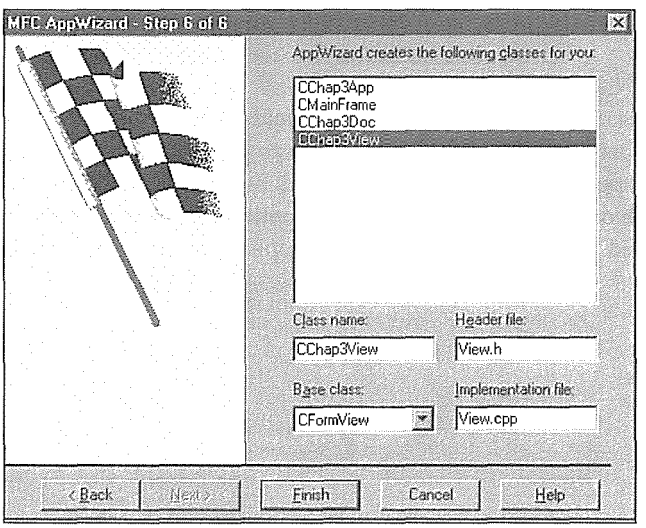


Figure 3.6 AppWizard dialog box, step 6.

There are a few things that must be done here. First, for consistency throughout the examples, select the **CChap3Doc** class and change the header file name to **DOCUMENT.H** and the implementation file to **DOCUMENT.CPP**. Then change the `CChap3View` class header file to **VIEW.H** and the implementation file to **VIEW.CPP**. Finally, make sure that the base classes of the `CChap3View` class is `CFormView` instead of `CView`.

`CFormView` allows easy placement of controls on the client area of a view. This is similar to a dialog box–based application, because `CFormView` uses a dialog resource for the client area. Earlier versions of AppWizard required the developer to do some of the work to derive from `CFormView`, but the latest versions make this task painless, as you will see.

Click the **Finish** button, and you will see the dialog window shown in Figure 3.7. Make sure all the options are correct and then click the **OK** button.

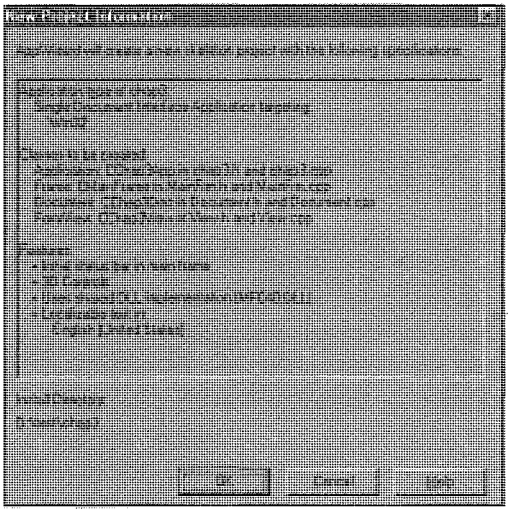


Figure 3.7 New Project Information dialog window.

As it says, the "skeleton project" has been generated, so let's go ahead and see whether it will compile and link before we add a line of code. Then we run it. The application doesn't do much, but in a few easy steps AppWizard has produced a fairly complete Windows application that can be enhanced with our intended functionality.

MFC Application Class Hierarchy

Our simple MFC SDI application contains the five highlighted classes in Figure 3.8. The interaction among these classes defines the behavior of the application. In a moment we will go through each class in detail. First let's examine how the classes interact.

`CWinApp` controls the application at its highest level. It is responsible for managing the relationships among the other classes. `CDocument`, `CFormView`, and `CFrameWnd` are responsible for managing and displaying the application user's data. You will add most of your application's functionality by augmenting the `CView`- and `CDocument`-derived classes. `CFrameWnd` is very functional as provided.

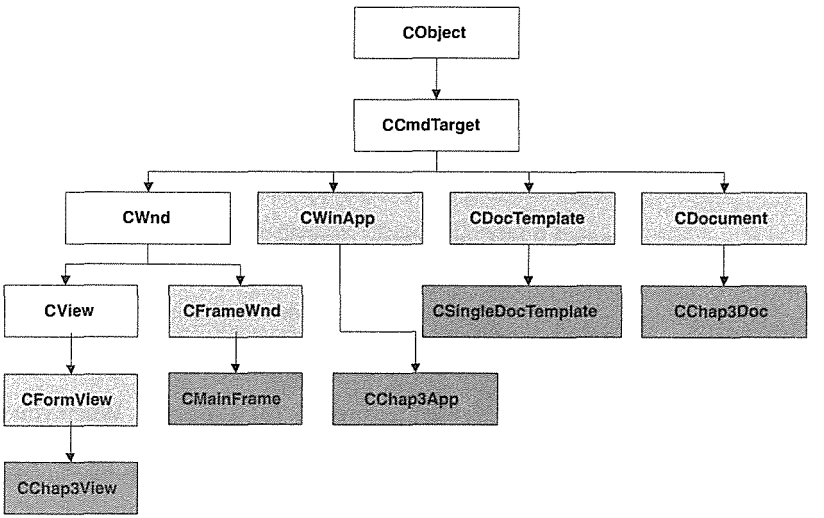


Figure 3.8 Abbreviated MFC application class hierarchy.

`CDocument`, `CFormView`, and `CFrameWnd` are collectively described as a *document template*. The document interacts with one or more views, and all views are contained inside a frame window. When the user selects **File/New**, `CWinApp` calls `CDocTempate` to create a new document/view/frame set. Applications can have more than one document type and thus would contain a list of document templates. This structure is illustrated in Figure 3.9.

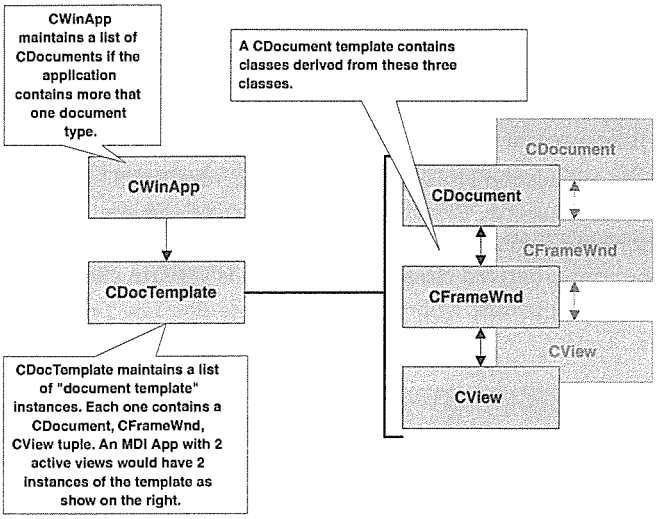


Figure 3.9 MFC application class relationships.

The Document/View Architecture

Most MFC applications use the document/view architecture that MFC supplies. When you use AppWizard to generate an application, the SDI and MDI options automatically create both a document and a view class. The document class (in our case, `CChap3Doc`) is derived from `CDocument`. Classes derived from `CDocument` provide functionality for the handling of "document" data. Whenever **File/New** is chosen, a new instance of `CChap3Doc` is created. `CChap3Doc` is responsible for storing data that our application should maintain when the user saves the document using **File/Save**. This data is loaded into the `CChap3Doc` class when the user reads a document from disk using **File/Open**. Of course, the concept of a *document* is different for every application. Microsoft Word's document is the internal representation of the characters, words, sentences, paragraphs, fonts, diagrams, and so on that the Word user creates. For Excel, the document is the internal representation of the spreadsheet and its figures and formulas.

The *view* part of the document/view architecture is responsible for rendering the document in the client area of the frame window or to a printer. Continuing the Word example, the view class will contain the code necessary to render the words, paragraphs, fonts, and so on of the document for presentation to the user. The view class provided by AppWizard for our application is `CChap3View`; it is derived from the `CFormView` class. The view is responsible for graphically displaying the data maintained in the `CDocument`-derived class. For our application, this view is actually a dialog resource that contains Windows controls. Our application has no associated data, so the document class is of little use. We will modify the `CChap3View` class to add the functions needed to test the `Expression` class.

The document/view architecture is important to most applications developed using MFC. For our purposes, though, it provides more functionality than we need. In Chapter 6, we will describe it in more detail

as we develop an application that needs the architecture. ActiveX controls typically do not use the MFC document/view architecture, but the architecture can be used in the development of MFC-based ActiveX control containers, and, as we'll see in a later chapter, a view can contain instances of ActiveX controls.

AppWizard-Generated Files

Now that we've generated our first MFC application, let's look at the files that were produced by AppWizard. Table 3.1 briefly describes each file that AppWizard created. Next, we'll go through the classes and code that were generated for our application.

Table 3.1 AppWizard-Generated Files

Files	Purpose
chap3.h, chap3.cpp	`CWinApp`-derived class: `CChap3App`
mainfrm.h, mainfrm.cpp	Class for the main application window, `CMainFrame`
document.h, document.cpp	`CDocument`-derived class: `CChap3Doc`
view.h, view.cpp	`CFormView`-derived class: `CChap3View`
stdafx.h, stdafx.cpp	Application framework includes
chap3.def	Windows definition file
chap3.mak	Visual C++–maintained make file; you should not modify this file directly. Modifications should be made through the Visual C++ IDE
chap3.rc	Windows resource file; contains dialog definitions, string tables, and menu resources
resource.h	Include file with application and framework-defined IDs
res\chap3.ico	Icon for the application; used when minimized and shown in the default About box
res\document.ico	Icon representing the specific document type; this is useful when there are multiple document types within the application
res\chap3.rc2	User-editable resource file; Visual C++ will not touch this file. Maintains version information and so on

CDocument

The `CDocument` class and its derivatives encapsulate and maintain the data of a user document. This document is quite different from the one you are now reading. `CDocument` represents the way typical Windows applications handle the creating, saving, and retrieving of files from the Windows File menu. When the application user selects **File/New**, MFC creates a new `CDocument` object and, using the services of `CView` and `CFrameWnd`, displays the data contained in the document. The document class contains a pointer to the associated `CView` class. When data within the document is modified, `CDocument` is responsible for notify-

ing the view that this modification has occurred. The view then determines whether the modified data is visible, and, if necessary, it updates the display.

The `CDocument` class stores data that needs to persist after the user closes the document. Temporary data that may be needed by the view when rendering should be maintained within the `CView` class. Think of an MFC document as a file (e.g., **TEST.DOC**) where you store your application data before, during, and after execution. How the data is stored is up to the developer, but MFC provides data serialization capabilities that simplify this process.

Our application does not require saving or retrieving data, so we don't actually need the `CDocument`-derived class, `CChap3Doc`. Table 3.2 contains useful members of the `CDocument` class. The functions that start with *On* are typically overridden in derived classes. Following are excerpts, with comments, from the **DOCUMENT.H** and **DOCUMENT.CPP** files. Our application will use the code as is.

```
//
// document.h
//
class CChap3Doc : public CDocument
{
...
   // Overrides
   // ClassWizard-generated virtual function overrides
   //{{AFX_VIRTUAL(CChap3Doc)
     public:
   // Override of the OnNewDocument function
   // Called when the user selects File/New or when the
   // framework initially displays the application
   virtual BOOL OnNewDocument();
   //}}AFX_VIRTUAL

// Implementation
public:
   virtual void Serialize(CArchive& ar);  // overridden for document i/o
...
};

//
// document.cpp
//
...
BOOL CChap3Doc::OnNewDocument()
{
   if (!CDocument::OnNewDocument())
      return FALSE;

   // TODO: add re-initialization code here
```

```
    // (SDI documents will reuse this document)

    return TRUE;
}

// CChap3Doc serialization
// You would add code here to save and restore
// any data maintained in CDocument
// This data defines the structure of the
// "document" when written to disk
void CChap3Doc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

Table 3.2 Useful `CDocument` Functions

Member	Purpose
`IsModified`	Has the document been modified since last save?
`SetModifiedFlag`	Indicates that the document has changed since last save.
`UpdateAllViews`	Notifies each view that the document has changed.
`DeleteContents`	Called to clear a document (e.g., when the user opens a new document in an SDI application it must remove the previous document data).
`OnCloseDocument`	Called when the user selects **File/Close** or its equivalent.
`OnNewDocument`	Called when the user selects **File/New** or its equivalent.
`OnSaveDocument`	Called when the user selects **File/Save**.
`OnOpenDocument`	Called when the user selects **File/Open**.

CView

The MFC `CView` classes are responsible for the visual display of the document's data. The view class typically handles Windows messages that relate to the rendering of information in the client area of a window and responds to command messages that may alter the display of the view. In our example application, the view class is defined in **VIEW.H** and **VIEW.CPP**.

We derived our view class from `CFormView`. `CFormView` associates a dialog resource with the client area of the frame window in an SDI application. Later, we will modify the dialog's resource file to add an entry field and two command buttons. Because `CFormView` uses a dialog resource as the view, we make sure that the client of the frame window is sized appropriately. The framework initially creates the frame window (and thus the client area) size using a default value stored in a static variable, `rectDefault`. For our purposes, we will override the `OnInitialUpdate` method and resize the frame window so that it is equal to our dialog resource. We'll do this in a moment, in our discussion of ClassWizard. Table 3.3 contains useful members of the `CView` class.

Table 3.3 Useful `CView` Members

Member	Purpose
`GetDocument`	Gets the `CDocument` instance associated with this view.
`OnInitialUpdate`	Called right after the view is attached to its associated document.
`OnDraw`	Called to render the document data within the view window.
`OnPrint`	Called to print or print preview the document.
`OnUpdate`	Called when the document data has changed, possibly requiring an update of the view.

CFrameWnd

The `CFrameWnd` class provides the functionality of a typical frame window in a Windows application. It contains and is responsible for the title bar, menu bar, status bar, and any control bars that are needed outside the client area. The frame window houses the application view, which in our application is an instance of `CFormView`. The frame creates the view and is responsible for many aspects of it. Our SDI application has only one frame window. It is the frame for both the application and our `CDocument`-derived class.

The `CFrameWnd` class provides much useful behavior that is standard for Windows applications. The menu bar contains the usual **File/New-**, **File/Open**-type commands and provides member functions that can be overridden to easily allow their implementation. The status bar and tool bars are easily maintained using methods provided within the `CFrameWnd` classes. I'm not going to show you the code generated by AppWizard for the `CMainFrame` class, because it is simple; however, it provides significant default functionality. There is one thing we should do for our `CMainFrame`: change its border so that it cannot be resized. We'll do this in a moment. Table 3.4 contains useful members of the `CFrameWnd` class.

Table 3.4 Useful `CFrameWnd` Members

Member	Purpose
`Create`	Creates a frame window.
`LoadFrame`	Creates a frame window based on a resource ID.
`GetActiveDocument`	Gets the active document.
`GetActiveView`	Gets the active view.

CDocTemplate

The CDocTemplate class manages the documents of an application. *Document* in this context includes the CFrameWnd-, CView-, and CDocument-derived classes. Information needed to instantiate the classes that form an application document is maintained in a CDocTemplate object. CWinApp contains a member variable, m_templateList, that maintains a list of all the valid document templates for the application. CDocTemplate is an abstract class with two standard MFC-derived classes: CSingleDocTemplate for SDI applications and CMultiDocTemplate for MDI applications. Following is the code that creates the document/view/frame template and adds it to the application's template list.

```
//
// chap3.cpp
//
...
BOOL CChap3App::InitInstance()
{
  ...
  // 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(CChap3Doc),
    RUNTIME_CLASS(CMainFrame),     // main SDI frame window
    RUNTIME_CLASS(CChap3View));
  // Update the list of valid "documents" for the application
  AddDocTemplate(pDocTemplate);
  ...
}
```

Table 3.5 Useful CDocTemplate Members

Member	Purpose
GetDocString	Retrieves document information from the resource file; examples include windowTitle, docName, filterName, etc.

CWinApp

Our main application class, CChap3App, is derived from CWinApp. CWinApp provides basic support for a Windows application. Except for one function, InitInstance, that has been overridden, CChap3App inherits all its functionality from CWinApp. The application instance is declared as a global object in

CHAP3.CPP. Global C++ objects are constructed *before* any application code is executed. This practice ensures that the application object is available before MFC enters `WinMain`. The standard Windows function `WinMain` is not provided within the `CWinApp` class but is provided by the framework. After MFC has set up its environment by registering window classes and initializing variables, including our already-constructed application instance, it calls the `InitInstance` member function, which is responsible for creating and displaying the main application window. `InitInstance`, as generated by AppWizard, is shown next:

```
// chap3.h
class CChap3App : public CWinApp {
  ...
  // Override of CWinApp InitInstance
  virtual BOOL InitInstance();
  ...
}

//
// chap3.cpp
//
...
// Global instance of CWinApp, constructed before WinMain() is called
CChap3App theApp;
...
BOOL CChap3App::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
   // Call this when using MFC in a shared DLL
   Enable3dControls();
#else
   // Call this when linking to MFC statically
   Enable3dControlsStatic();
#endif

   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(CChap3Doc),
        RUNTIME_CLASS(CMainFrame),       // main SDI frame window
        RUNTIME_CLASS(CChap3View));
    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;

    return TRUE;
}
```

`InitInstance` is called only once; its purpose is to set up the application environment and ultimately display the main window. Let's follow the preceding code.

Depending on the version of Windows that the application is running on, `Enable3dControls` either does nothing (Windows 95, Windows NT 4.0) or loads **CTL3D32.DLL** so that the standard Windows controls will appear three dimensional. `LoadStdProfileSettings` loads the MRU list of recently accessed files and attaches them to the File menu of the frame window if this feature has been enabled (in our case it has not). Next, a document template is created and our view/document/frame classes are associated with it. The application resource ID, `IDR_MAINFRAME`, is also included in the document template. This ID is used to load menus, toolbars, etc. for the application.

Every document type has certain resources. Typically, there are menu, string, accelerator, and icon resources associated with a document template. They are used in the creation of the frame window and child window areas. Pertinent sections of the **.RC** file are detailed next:

```
// chap3.rc - sections of the application resource file
...
// Icon
IDR_MAINFRAME      ICON  DISCARDABLE   "res\\chap3.ico"

IDR_CHAP3TYPE      ICON  DISCARDABLE   "res\\document.ico"

// Menu
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "&New\tCtrl+N",        ID_FILE_NEW
```

```
    MENUITEM "&Open...\tCtrl+O",    ID_FILE_OPEN
    MENUITEM "&Save\tCtrl+S",       ID_FILE_SAVE
  ...
END

// Accelerator
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
  ...
  "N",      ID_FILE_NEW,      VIRTKEY, CONTROL
  VK_F6,    ID_NEXT_PANE,     VIRTKEY
  ...
END

// This stringtable contains 7 substrings delimited by
// new lines (\n) that are used by the framework to define
// certain things.
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME "Chapter 3 Example\n\nChap3\n\n\nChap3.Document\nChap3 Document"
END
```

The string table contains seven substrings that are used by the framework. The first item is the caption of the main window. The others contain the default document extension, document type name, and OLE type name (which we'll cover in the next chapter).

The `RUNTIME_CLASS` macro returns the run-time class structure that allows the document template to dynamically create instances of the document, view, and frame classes. Once the instance of `CSingleDocTemplate` is created, it is added to the list of application-supported document types by calling `AddDocTemplate`.

Next, an instance of the `CommandLineInfo` class is passed to `ParseCommandLine`, which parses the command line and updates the `CommandLineInfo` instance. By default, MFC provides seven command-line options, each of which is indicated by an `enum` and two Booleans maintained in the `CommandLineInfo` class shown next. Table 3.6 shows the syntax and actions provided by the default MFC implementation. The last two are OLE-specific; we'll have more to say about them in later chapters.

```
class CCommandLineInfo : public CObject
{
public:
   // Sets default values
   CCommandLineInfo();
   ~CCommandLineInfo();

   ...
```

```
    enum{
        FileNew,
        FileOpen,
        FilePrint,
        FilePrintTo,
        FileDDE,
        FileNothing = -1
    } m_nShellCommand;

    BOOL m_bShowSplash;
    BOOL m_bRunEmbedded;
    BOOL m_bRunAutomated;
    ...
};
```

Table 3.6 MFC Default Command-Line Support

Command Line	Action
`appname`	Start with a new file.
`Appname` *filename*	Start and open the specified file.
`Appnamne` **/p** *filename*	Print the specified file to the default printer.
`Appname` **/pt** *filename* `printer_port`	Print the specified file to the specified printer.
`Appname` **/dde**	Start the application and await DDE command.
`Appname` **/Automation**	Start the application as an Automation server.
`appname` **/Embedding**	Start the application to edit an embedded OLE item.

Finally, `ProcessShellCommand` is called with the updated instance of `CommandLineInfo`. `ProcessShellCommand` then performs whatever action was specified on the command line. In most situations, this action will be a call to `OnFileNew`. In other cases, such as when the command line specifies the printing of a document, MFC will hide the application, print the document, and return `FALSE` from `ProcessShellCommand`. A return of `FALSE` forces the application to shut down.

In our case, `OnFileNew` starts everything. It calls `OpenDocumentFile`, which causes the creation of the document, which causes the creation of the frame window, which.... You get the idea. Figure 3.10 shows the process in fine detail. If everything works correctly, `ProcessShellCommand` returns `TRUE` to indicate that the main window has been created and processing should continue. Once this occurs, the framework invokes `CWinApp::Run` and starts processing the application's message loop.

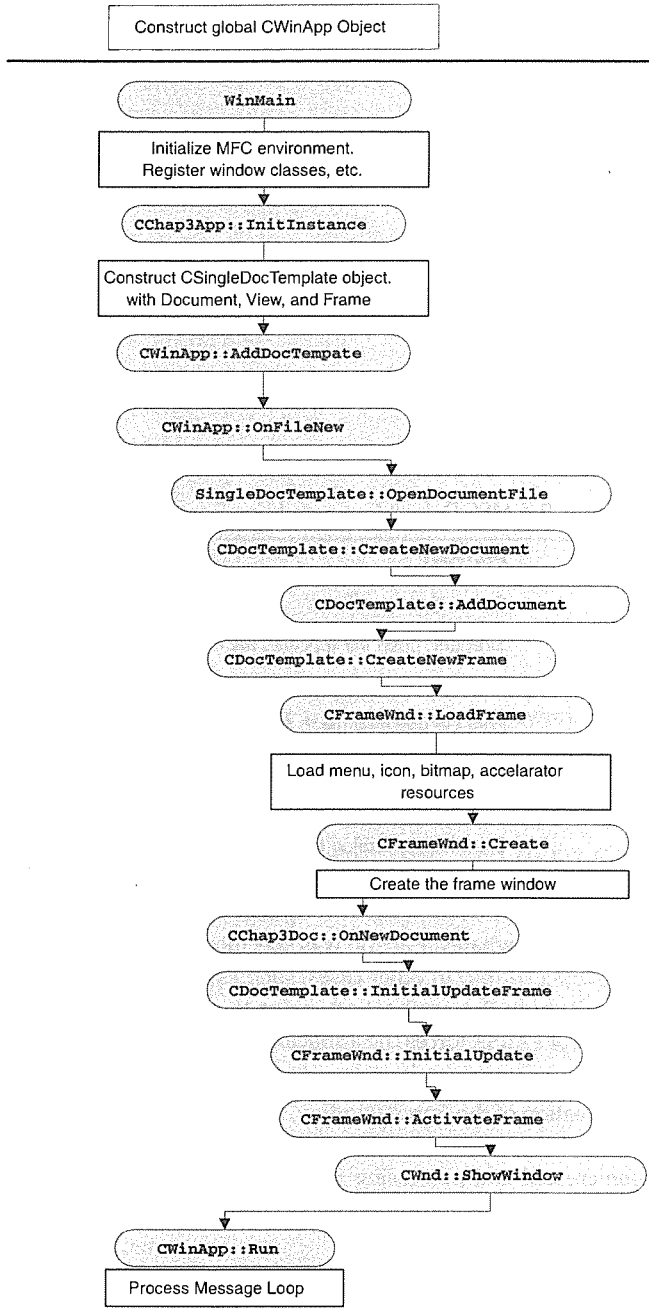


Figure 3.10 MFC application startup flow.

Table 3.7 describes some of the useful CWinApp members.

Table 3.7 Useful CWinApp Members

Member	Purpose
m_hInstance	Handle to the application instance.
m_lpCmdLine	Command line used to invoke the application.
m_pMainWnd	Pointer to application's main window.
m_pszProfileName	The application's INI filename.
m_pszRegistryKey	The complete registry key for the application; see the SetRegistryKey, WriteProfileString, and GetProfileString methods for details.
AddDocTemplate	Adds a document template, CDocTemplate, instance to the list of valid application documents.
WriteProfileString	Writes data to either the application's INI file or the system registry; the SetRegistryKey method determines which technique is used.
ParseCommandLine	Parses the command line that started the application; MFC provides default support for seven basic command-line options.
ProcessShellCommand	Opens a new file, prints a file provided on the command line, etc., depending on the state of the provided CommandLineInfo instance.
InitInstance	Initializes the application instance, used to define documents, views, and frames; sets up document templates and displays, initially, the main application window.
Run	Starts message loop, dispatches messages, etc.
ExitInstance	Called prior to exiting application.
OnFileOpen	Called when user selects **File/New**.

Editing and Adding Resources

We now have a simple MFC application that we can run, but it doesn't do much. Let's continue our quest to build an application that will use the Expression class from Chapter 2. We indicated to AppWizard that our view class should be derived from CFormView. This selection allows the client area to behave like a dialog box, which makes it easy to add controls. It isn't exactly a dialog box, but it does use a dialog resource to identify and position the controls.

To invoke the dialog editor, click the **Resource View** tab in the Project View window and then click the **chap3 resources** folder. This action brings up a list of the existing resource types in the application. Click the **Dialog** folder, and you should see two dialog resources: IDD_ABOUTBOX and IDD_CHAP3_FORM. AppWizard has created a default dialog resource for our CChap3View class. Double-click **IDD_CHAP3_FORM** and the dialog editor will look like Figure 3.11.

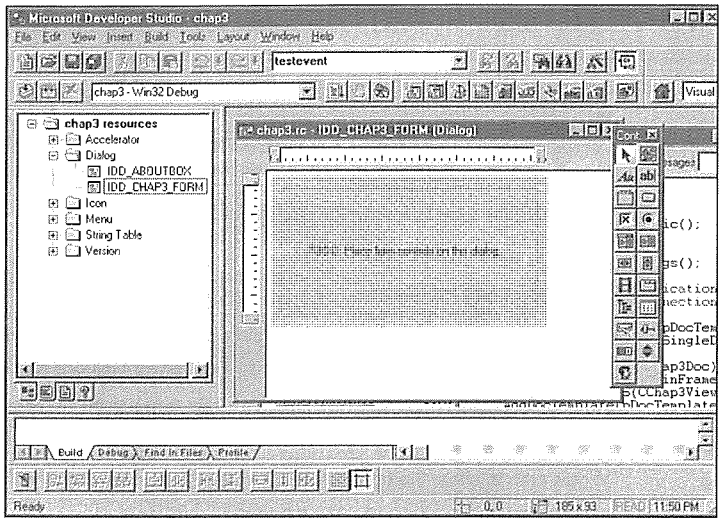


Figure 3.11 Editing the `CChap3View` dialog resource.

Resize the **To do** static field and change the text to **Enter an expression:**. Increase the size of the dialog box and add an entry field with an ID of `IDC_EXPRESSION`. Also, add two buttons: one with a caption of **Validate** and the other with a caption of **Evaluate**. Set the IDs for these buttons to `IDC_VALIDATE` and `IDC_EVALUATE`. When you're finished, it should look something like Figure 3.12.

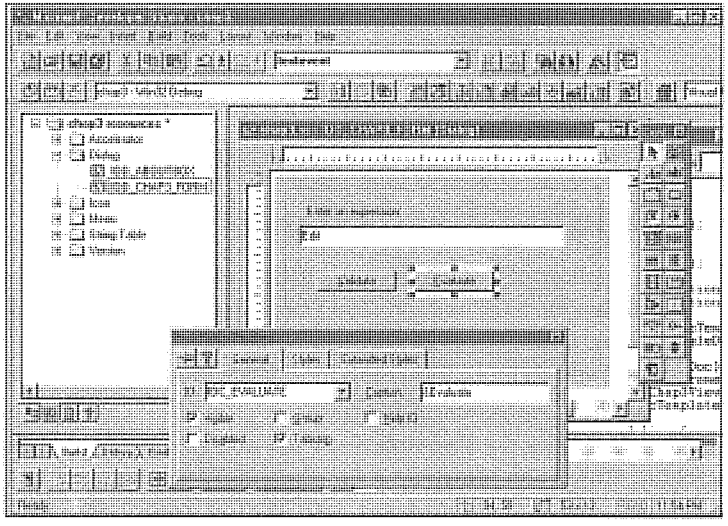


Figure 3.12 Finished `CChap3View` dialog resource.

Once we've modified the dialog resource and we recompile and relink, the application should have a working entry field and two push buttons. The dialog controls don't actually do anything yet. For that, we must tie a function to the specific control event using the Visual C++ ClassWizard.

ClassWizard

Even though you can't use AppWizard to change your application source once it has been initially generated, Visual C++ provides other tools to help with the management of various aspects of the application. ClassWizard performs many of these functions.

Visual C++ 2.0 added a useful feature to ClassWizard: ClassWizard now shows all the overridable (virtual) functions for the selected class in your application. This arrangement makes it easy to override the functions necessary to add functionality. In previous versions, you had to look up the function name, add the declaration to the header file, and implement the code in the **.CPP** file. ClassWizard now does everything except write the implementation code.

As I discussed earlier, we want the initial size of our frame window to be the same size as our `CFormView` dialog resource. The easiest way to do this is to override the `OnInitialUpdate` function of our view class, `CChap3View`. `OnInitialUpdate` is called right before the frame (and client) windows are displayed. This gives us time to resize the frame to match our dialog box. Start ClassWizard by pressing **Ctrl-W**. Click on the **Message Map** tab and select **CChap3View** from the drop-down list. Choose **CChap3View** in the **Object IDs** listbox. You should see a screen similar to Figure 3.13.

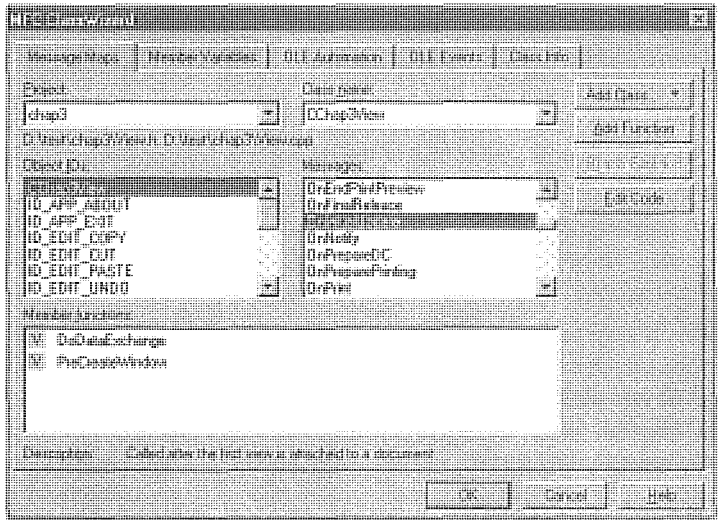


Figure 3.13 Overriding parent methods with ClassWizard.

Next, select the **OnInitialUpdate** method from the **Messages** listbox, click the **Add Function** button, and then click **Edit Code**. Add the following code:

```
void CChap3View::OnInitialUpdate()
{
  // TODO: Add your specialized code here and/or call the base class
  CFormView::OnInitialUpdate();

  // Set the frame window to the size of the Form (Dialog)
  GetParentFrame()->RecalcLayout();
  ResizeParentToFit( FALSE );
  ResizeParentToFit( TRUE );
}
```

While we're at it, we should ensure that our newly sized frame cannot be resized by the application user (now that it's the right size). We do this by removing the `WS_BORDER` window style just before the frame is created. Start ClassWizard again, choose the **CMainFrame** class, and override the `PreCreateWindow` method to do the following:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  // TODO: Add your specialized code here and/or call the base class

  // Don't include the WS_THICKFRAME style
  cs.style &= ~WS_THICKFRAME;

  return CFrameWnd::PreCreateWindow(cs);
}
```

Build the application to verify that its size is that of the dialog box we created for `CFormView`. You should be unable to resize the window, because it does not have a border.

Message Maps

ClassWizard also makes it easy to manage MFC message maps (message maps will be discussed in more detail in the next section), which provide a convenient way to map Windows messages to specific C++ functions. Let's do that for the two push buttons that we added to our application. Invoke ClassWizard (**Ctrl-W**) and click the **Message Map** tab. Select **CChap3View** from the **Class Name** drop-down list and you'll see a dialog box similar to Figure 3.14.

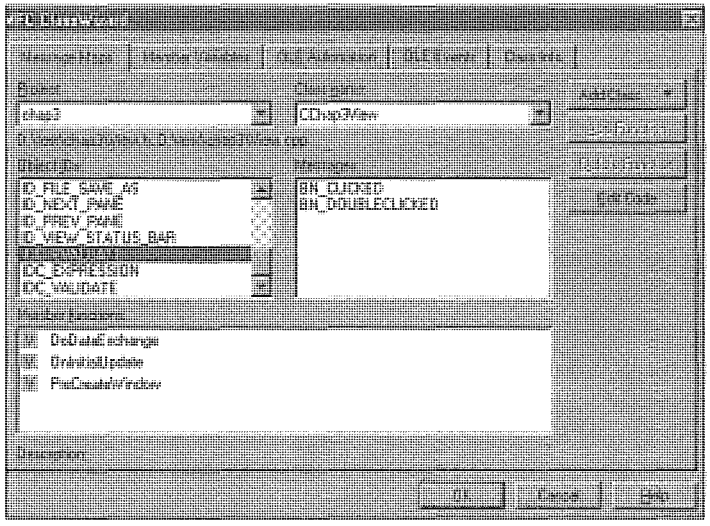


Figure 3.14 Message Maps dialog box.

Each of the custom IDs that we added through the dialog editor (e.g., `IDC_EVALUATE`) shows up under the **Object IDs**. When an ID is selected, the messages generated by the particular ID are displayed in the other listbox. We're interested in the `BN_CLICKED` message that is generated whenever a user clicks the push button. Click the **Add Function** button to add a member function for each button. This action should add two methods—`OnValidate` and `OnEvaluate`—to the `CChap3View` class. Now when the `BN_CLICK` message is generated from the `IDC_VALIDATE` push button, the `OnValidate` member function will be called. All this occurs through the magic of MFC message maps. Let's add one more message handler for instructional purposes. Select **CChap3View** in the **Object IDs** listbox and double-click on the **WM_LBUTTONDOWN** message to add a handler called `OnLButtonDown`.

Windows is a message-based operating environment. Everything that occurs does so because of a Windows or application *event* that is communicated through a Windows message. Anyone experienced in Windows programming understands this concept. The MFC libraries provide a mechanism called *message mapping* that hides somewhat the handling of these messages. Gone is the standard Windows `GetMessage/DispatchMessage` loop. Gone also are the enormous switch statements that SDK developers are used to. In their place are MFC message maps.

Microsoft implemented message mapping primarily because of the inherent inefficiencies of using the C++ virtual function mechanism. A good object-oriented solution would have been to provide a virtual function for each message that a particular object could handle. Here's an example of the how the `CWnd` class might have been implemented:

```
class CWnd : public CObject
{
   virtual int OnCreate( LPCREATESTRUCT& );
```

```
    virtual void OnDestroy();
    virtual void OnMove( int, int );
    virtual void OnSize( UINT, int, int );
    ...
};
```

As you can imagine, the virtual table (vtable) for this class would be large and inefficient. MFC uses message maps to overcome this problem without using proprietary C++ constructs. The message map solution uses standard C++ notation and conventions (and a few macros to make things more readable).

Message Types

Before we take a look at MFC message maps, we need to categorize the different types of Windows messages. The ones we're most used to dealing with are those generated by Windows for a particular window. Examples include `WM_PAINT`, `WM_LBUTTONDOWN`, and `WM_MOVE` (basically, all the `WM_` messages except `WM_COMMAND`). We'll classify these messages as *window messages*; they are, by their nature, destined to be handled by a window.

The `WM_COMMAND` message is used for two purposes. The first is for Windows control notification messages. These messages typically are sent from a child window to its parent. For example, in our application, the `CFormView` class contains an entry field and two command buttons. Each of these elements is a child window to the parent (the client window). When a command button is pressed, a `WM_COMMAND` message containing a `BN_CLICKED` notification code is passed to the parent.

The second use of WM_COMMAND messages is in the communication of menu selections and accelerator key presses. These messages aren't meant for a specific window, but they can be sent to various objects within the application as a whole. In our example application, the **File/New** selection generates a `WM_COMMAND` containing `ID_FILE_NEW`. This message could be handled by the frame object, the document object, or the view. Because `WM_COMMAND` messages need to be routed to various application objects, they are treated differently from standard Windows messages and control notification messages. Next, we'll detail how each message type is handled.

Window Messages and Control Notifications

Standard window messages and control window notifications are handled by a window. In MFC, all window objects derive from the `CWnd` class. `CWnd` has default message handlers for many of the standard window messages. If you were to look at the declaration for `CWnd`, you would see the following function declaration:

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

`OnLButtonDown` isn't virtual, but it provides a default implementation for the message mapping architecture. `CWnd::OnLButtonDown` calls the default window procedure if no window overrides or contains a message map entry for `WM_LBUTTONDOWN`.

We want to implement a message handler for the WM_LBUTTONDOWN message in our view class, so let's see how that works. Here's what ClassWizard added to the CChap3View class:

```
//
// view.h
//
...
class CChap3View : public CFormView
{
protected: // create from serialization only
   CChap3View();
   DECLARE_DYNCREATE(CChap3View)
...
// Generated message map functions
protected:
   //{{AFX_MSG(CChap3View)
   afx_msg void OnEvaluate();
   afx_msg void OnValidate();
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   //}}AFX_MSG
 DECLARE_MESSAGE_MAP()
};

// in view.cpp

BEGIN_MESSAGE_MAP(CChap3View, CFormView)
   //{{AFX_MSG_MAP(CChap3View)
   ON_BN_CLICKED(IDC_EVALUATE, OnEvaluate)
   ON_BN_CLICKED(IDC_VALIDATE, OnValidate)
   ON_WM_LBUTTONDOWN()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

This code in **VIEW.H** and **VIEW.CPP** sets up a static array of message-map entries for the CChap3View class. The DECLARE_MESSAGE_MAP macro does most of the work. It creates a message map with three entries, each of which has its own unique signature (maintained by the framework). We're discussing the OnLButtonDown handler for now; we'll discuss the ON_BN_CLICKED entries next, but they are in the message map. The code in **VIEW.CPP** initializes the message map with the function addresses for our handler, CChap3View::OnLButtonDown, and initializes a pointer in the message map that points to our base class (CFormView) message map. The ON_WM_LBUTTONDOWN entry is a little tricky, because it doesn't contain the name of the function handler. When you added the WM_LBUTTONDOWN handler using ClassWizard, it didn't

give you an opportunity to change the handler's name. The name of the handler must match that declared in the CWnd class. It's just like overriding a parent class member.

We know that CWnd has a default handler for OnLButtonDown, but we want *our* handler to be called instead. How does MFC do it? When the WM_LBUTTONDOWN message is generated by pressing the left button within the client area of our application, CWnd::WindowProc is called. If the message is not a command message, the message map for the "most derived" class (CChap3View) is checked for an entry that matches the message. If an entry in the map is not found in that class, the next class in the chain is checked, and so on. This continues until the inheritance hierarchy is exhausted. If no one handles the message, it is passed to DefWindowProc for default processing. If a handler is found, it is called and the search stops. Then it is important to call the parent's implementation of the handler if additional processing is required. In our example, once we process the OnLButtonDown event, we pass the message to our parent CFormView by directly calling its implementation.

Handling window messages is wonderfully easy using MFC, once you get the hang of it.

Command Messages

In the previous example, once we reached CWnd in the search for a handler and none was found, the framework called DefWindowProc. Command messages are treated differently. First, if command messages never find a handler, instead of calling a default hander, they're ignored. Second, command messages can be routed to various framework objects and not just to windows. A special class, CCmdTarget, provides the routing of command messages between the framework objects. Any class derived from CCmdTarget (including, of course, a CWnd-derived class; CWnd is derived from CCmdTarget) can participate in the receiving and routing of command messages.

What about our BN_CLICKED messages that are generated when a user clicks one of our command buttons? Well, technically, BN_CLICKED is a control notification message and it should be handled like the other standard Windows messages. But because command buttons can be used as tool bar buttons, the framework treats them as if they were command messages. They can be routed like all command messages.


NOTE

In reality, all control notification messages, not just BN_CLICKED, can be routed like command messages. This capability is explained in detail in MFC Tech Note 21, but its use outside of BN_CLICKED, where it has a specific purpose, is not recommended because ClassWizard does not support this feature.

Command messages from menus or tool bars are typically first received by the application's frame window. Without the ability to *route* these messages using `CCmdTarget` and the message-mapping architecture, all commands would have to be handled in the `CMainFrame` class, which is not where you would typically want an MFC application. Command routing follows the course illustrated in Figure 3.15. When a frame window command message is received, it is routed first to the active view. If no handler is found, the routing continues all the way to the application object. If it is not processed by any object, it is discarded.

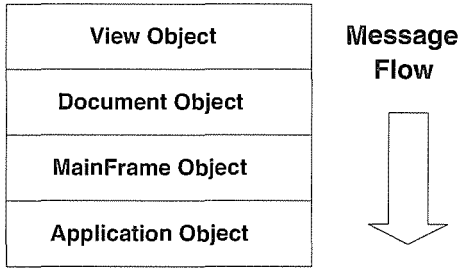


Figure 3.15 Command routing in an SDI application.

Command messages that contain control notifications (e.g., `BN_CLICKED`, `EN_CHANGED`) are received by their parent window, which is usually a dialog window. In our example, the button controls send their messages to the parent window—which is the view—and not to the application's frame window. This is the primary difference between command messages that are not actual control notifications and those command messages that are.

We have taken a quick look at the MFC library message maps for processing and routing Windows messages. Within the MFC libraries, this powerful concept of message routing is not confined to the handling of Windows messages. This concept is used in various MFC areas, including COM and OLE. So we've briefly stopped to ponder what message maps are and how they work. Later, understanding various concepts of COM, OLE, and ActiveX in the context of MFC will be easier.

Adding the Expression Class

We now have all the pieces in place to add the functionality from the `Expression` class that we developed in the last chapter. To do this, first copy the **EXPRESS.CPP** and **EXPRESS.H** files from the CD-ROM (**EXAMPLES\CHAP3**) to the working directory for your CHAP3 project. Then, from the Insert menu select **Insert Files into Project...** and add the file **EXPRESS.CPP** to the project (see Figure 3.16).

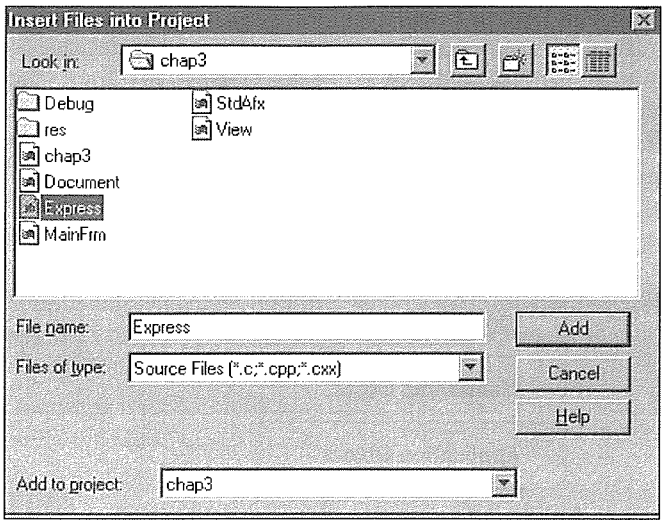


Figure 3.16 Adding the file **EXPRESS.CPP** to the project.

Add a forward declaration for the `Expression` class and add an `Expression` class pointer member variable to the `CChap3View` class as follows:

```
//
// view.h : interface of the CChap3View class
//
/////////////////////////////////////////////////////////////////////////////

// Forward declaration of Expression class
class Expression;
class CChap3View : public CFormView
{
...
protected:
  Expression*  m_pExpression;
...
};
```

Now we need to do something when the user clicks the `Validate` and `Evaluate` keys on the form view. Add the following code to **VIEW.CPP** to use the `Expression` class:

```
//
// view.cpp : implementation of the CChap3View class
//

#include "stdafx.h"
```

```
#include "chap3.h"

#include "document.h"
#include "view.h"

// Include our Expression class declarations
#include "express.h"

...
////////////////////////////////////////////////////////////////////////////
// CChap3View construction/destruction
CChap3View::CChap3View()
     : CFormView(CChap3View::IDD)
{
   //{{AFX_DATA_INIT(CChap3View)
   // NOTE: the ClassWizard will add member initialization here
   //}}AFX_DATA_INIT

   // TODO: add construction code here
   m_pExpression = new Expression;
}

...

void CChap3View::OnEvaluate()
{
   // TODO: Add your message handler code here
   CString strExpression;
   char szTemp[128];

   // Get the expression from the entry field
   CWnd* pWnd = GetDlgItem(IDC_EXPRESSION);
   ASSERT( pWnd );
   pWnd->GetWindowText( strExpression );

   TRACE1( "OnEvaluate: Expression is %s\n", strExpression );

   // Set the expression to evaluate
   m_pExpression->SetExpression( strExpression, TRUE );
   long lResult = m_pExpression->Evaluate();
   sprintf( szTemp, "%ld", lResult );

   // Update the entry field with the result
   pWnd->SetWindowText( szTemp );

   // Set focus back to the entry field
   GetDlgItem( IDC_EXPRESSION )->SetFocus();
```

```
}
void CChap3View::OnValidate()
{
   // TODO: Add your message handler code here

   CString strExpression;

   // Get the expression from the entry field
   CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
   ASSERT( pWnd );
   pWnd->GetWindowText( strExpression );

   TRACE1( "OnValidate: Expression is %s\n", strExpression );

   m_pExpression->SetExpression( strExpression, TRUE );
   if (! m_pExpression->Validate() )
      AfxMessageBox( "Invalid Expression, try again" );

   // Set focus back to the entry field
   GetDlgItem(IDC_EXPRESSION)->SetFocus();
}
```

Once you've entered the code, compile, link, and run the application in debug. Everything should work. Figure 3.17 shows the finished application with a complex expression about to be evaluated. After you terminate the application, examine the **Debug** tab and see whether there are any problems with the application. There should be, so let's review some MFC debugging techniques.

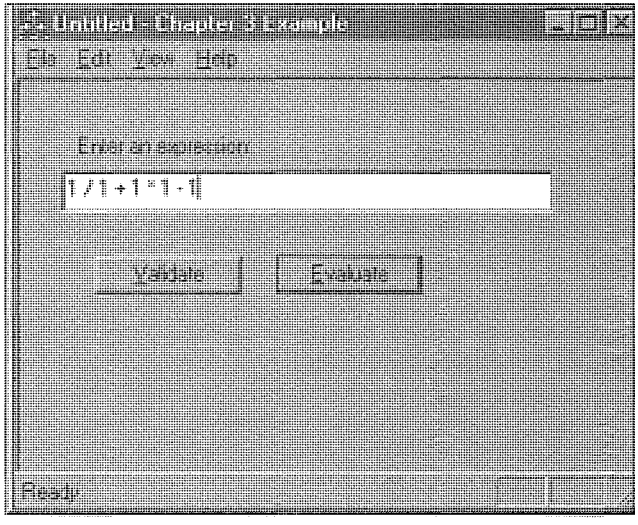


Figure 3.17 Finished application.

MFC Debugging Techniques

The MFC libraries provide a rich set of functions and classes for debugging applications. We'll look at a couple of the debugging features available to MFC developers. In later chapters, we'll use more-sophisticated techniques as we develop complex applications. Windows application debugging is a difficult task, but when you introduce COM into the equation—with multiple, interacting processes—it gets hairy. Luckily, MFC provides facilities to help with this process.

First, there are the `TRACE` macros which allow you to display messages as your code is executing. In the previous example, we used the `TRACE1` macro to update the actual expression in the `OnEvaluate` and `OnValidate` methods. The `TRACE` macros work just like the old C-style `printf` functions, so you can format the output however you like. To see the trace output, however, you must compile and link your project in debug. This basically means defining the `_DEBUG` flag.

Second, there's the `ASSERT` macro, which is provided by MFC. It is a simple enhancement of the C-style `assert` function. The `ASSERT` macro is a convenient way of checking the validity of your program's variables; it provides a basic sanity check. The `ASSERT` macro is used like this:

```
// Get the expression from the entry field
CWnd* pWnd = GetDlgItem(IDC_EXPRESSION);
ASSERT( pWnd );
pWnd->GetWindowText( strExpression );
```

We're about 100 percent sure that we will get a valid, non-null pointer back from the `GetDlgItem` method, but just to be sure, we assert that it is nonzero. If this assertion fails, the `ASSERT` macro will pop up a dialog box informing us that it failed, and it will provide the file and line number where it failed. It is a good idea to sprinkle your code with assertions like this; it helps tremendously during the debugging phase of a project. You can't have too many, because they don't affect the size or performance of the release version of your program. If the `_DEBUG` flag is not set, the `ASSERT` macro is defined as nothing (i.e., the code is removed).

The MFC source code is a great example of this technique. There are thousands of `ASSERT`s throughout the MFC source code. There are also specialized versions. `ASSERT_POINTER` asserts that the parameter is a valid MFC-based pointer. `ASSERT_KINDOF` tests whether a class instance is a specific type of class. There are several other `ASSERT`s as well. Check out the MFC source code and the help files for more information.

Another nice feature provided by MFC is its tracking of every memory allocation (`new`) and deallocation (`delete`) that your application performs. If you somehow forget to delete something that you have allocated, MFC will dump (to the debug window) the source file and line number of the specific allocation that was not deallocated, assuming that you adhere to the following rules:

1. Derive your class directly or indirectly from `CObject`. Almost all the classes in MFC derive from `CObject`, so this shouldn't be a problem.
2. Within your class declaration, include the `DECLARE_DYNCREATE(`*`classname`*` )` macro and within the **.CPP** file for your class, include the `IMPLEMENT_DYNCREATE(`*`classname, parentclass`*` )` macro.

3. In your **.CPP** file, after any IMPLEMENT_DYNCREATE() macros, change the line: `#define new` to be `DEBUG_NEW`. This is added by default when you compile with `_DEBUG` defined.

In the example, we forgot to delete the `Expression` instance that we created in the constructor of the view class. When you terminate the application, you should see something similar to the following in the debug window:

```
Loaded symbols for 'C:\WINNT\system32\MFC42D.DLL'
Loaded symbols for 'C:\WINNT\system32\MSVCRTD.DLL'
  ...
Detected memory leaks!
Dumping objects ->
strcore.cpp(76) : {159} normal block at 0x004127A0, 20 bytes long.
 Data: <            2343> 01 00 00 00 07 00 00 00 07 00 00 00 32 33 34 33
D:\test\chap3\View.cpp(41) : {65} normal block at 0x004118B0, 8 bytes long.
 Data: < 'A     > AC 27 41 00 01 00 00 00
Object dump complete.
The program 'D:\test\chap3\Debug\chap3.exe' has exited with code 0 (0x0).
```

Because the AppWizard provided the `DEBUG_NEW` definition for us, we can immediately determine which line of code allocated memory but did not delete it. Line 41 of **VIEW.CPP** looks like this:

```
////////////////////////////////////////////////////////////////////////////
// CChap3View construction/destruction
CChap3View::CChap3View()
     : CFormView(CChap3View::IDD)
{
   m_pExpression = new Expression;
}
```

Just as we thought, we forgot to deallocate our `Expression` instance. MFC can do even more. If we derive our class from `CObject` and add the `DECLARE_DYNCREATE` and `IMPLEMENT_DYNCREATE` macros, MFC will display the class name of the object that wasn't deallocated. Classes derived from `CObject` inherit run-time type identification (RTTI) abilities. (This isn't the C++ standard RTTI, but it provides similar capabilities.) This arrangement allows the debugger to determine the type and name of the object that wasn't deallocated.

This behavior is provided by the `DECLARE_DYNAMIC` and `IMPLEMENT_DYNAMIC` macros as well as the `DYNCREATE` macros we are using. The `DYNCREATE` macros also allow MFC to dynamically create instances of classes at run time. We've used `DYNCREATE` instead of the more basic `DYNAMIC` macros here, because in the next chapter MFC will need the `DYNCREATE` feature to create instances of COM-enabled classes. Here's an example of adding the new macros to the `Expression` class:

```
// express.h
...
class Expression : public CObject
```

```
{
protected:
  DECLARE_DYNCREATE( Expression )
...
};

// express.cpp
...
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "express.h"
IMPLEMENT_DYNCREATE( Expression, CObject )
```

Now we get even more information about our memory leak. In addition to the line number, MFC informs us of the class name that caused the problem:

```
Loaded symbols for 'C:\WINNT\system32\MFC42D.DLL'
Loaded symbols for 'C:\WINNT\system32\MSVCRTD.DLL'
...
Detected memory leaks!
Dumping objects ->
strcore.cpp(76) : {401} normal block at 0x00411D00, 17 bytes long.
 Data: <            235 > 01 00 00 00 04 00 00 00 04 00 00 00 32 33 35 20
D:\test\chap3\View.cpp(41) :
        {65} client block at 0x004118B0, subtype 0, 12 bytes long.
a Expression object at $004118B0, 12 bytes long
Object dump complete.
The program 'D:\test\chap3\Debug\chap3.exe' has exited with code 0 (0x0).
```

To fix the leak, we need to ensure that we delete the `Expression` instance in the destructor of the view class:

```
CChap3View::~CChap3View()
{
  delete m_pExpression;
}
```

When you're developing MFC applications, I recommend that you use all the available features. You're paying for them with the additional overhead, so you should take advantage of what they offer in the way of application debugging. The use of `DYNCREATE` is a simple example. There are several classes designed just for debugging. When you get a chance, check out the features of the `CObject`, `CMemoryState`, and `CDumpContext` classes.

Summary

In this chapter, we reviewed the benefits of developing software with Visual C++ and Microsoft's Foundation Class libraries. Using Visual C++ and MFC allows us to be more productive as Windows developers, because we can focus on the problems being solved rather than on continually solving old ones. We reviewed the main design features of MFC, and in the process we completed a simple MFC application that tested the `Expression` class from Chapter 2.

The only way to really understand MFC is to dive into the source code. It is included with the professional edition in the default location **\MSDEV\MFC\SRC**. Next time you're debugging your code, go ahead and step into those MFC functions. It may be confusing at first, but after some time you'll begin to understand what's going on. Then watch your productivity skyrocket (or plummet if you get too enthralled with what's happening).

This book is about developing COM-based components and ActiveX controls, so let's get started.

Chapter 4

Microsoft's Component Object Model

In Chapter 3 we looked at Visual C++ and the MFC libraries and how they can help in the development of Windows software. In this chapter we will take a detailed look at Microsoft's Component Object Model (COM). Microsoft's COM provides the binary standard that we need to make our software more reusable, and we'll provide much low-level detail about how COM provides a binary standard wrapper for software components. You don't need to know this level of detail to use the technology, but if you understand what's going on at this level, you'll have a much easier time. It will also help when something goes wrong and you must do some low-level debugging. To paraphrase the great physicist Richard P. Feynman, once you understand the universe at the atomic level, everything else is easy. And so it is with COM: once you comprehend it at its lowest level, the rest (OLE, ActiveX) is simple.

The sample code in this chapter, except for the two example applications at the end, is used strictly for illustration and is not expected to be compiled and run. Instead, it's intended to demonstrate various C++ techniques used in building COM-based, binary standard components. After this chapter, we will be using the MFC implementation of COM, OLE, and ActiveX. Chapter 5 will provide a transition from the low-level COM to the higher-level ActiveX and MFC. So hang tight, we'll soon be doing some cool things.


NOTE

There is much confusion about what COM, OLE, and ActiveX are and how they are different. COM is a *system*-level standard that provides basic object model services on which to build higher-level services. OLE and ActiveX are examples of a higher-level service. OLE and ActiveX provide application-level features but are built using COM's services. So the terms *COM*, *OLE*, and *ActiveX* are somewhat interchangeable in that their capabilities and features are closely related. However, each term describes a separate set of technologies. OLE and ActiveX are built using COM, as are other new Microsoft technologies, such as the DirectX game SDK. All its services are accessed through COM-based interfaces. So don't let these terms get in the way of your understanding. We'll discuss COM in this chapter and then move to OLE and ActiveX in the remaining chapters. Just remember that OLE and ActiveX are primarily COM-based interface definitions with a few APIs thrown in to help.

A Binary Standard

In previous chapters, we discussed the importance of binary standards to the development of software components. Binary standards allow easy interoperation and reuse of components across language implementations (such as Microsoft and Borland C++), between disparate languages (such as Visual Basic and C++), and across process boundaries. COM is such a standard. We've discussed the benefits of using binary standards and the benefits of developing software applications using them. Now we're ready to dig into the details of how it all works.

In Chapter 2, we developed a C++ class, `Expression`, that we will continue to use in this chapter. Our design of the `Expression` class allows for easy reuse within the C++ language environment. Our goal now is to use COM to augment the `Expression` class so that it can be shared across various languages and processes.

When used within C++, the `Expression` class is a compile-time construct. We can create instances of `Expression` at run time, but the definition of `Expression` and how it behaves cannot change at run time. To use the `Expression` class in C++, you must include its definition (for example, **EXPRESS.H**). If the implementation for `Expression` changes, the program using it may need to be recompiled and relinked. COM eliminates this dependency on compile-time definitions by providing run-time mechanisms to query a component's functionality.

We will use the C++ language to implement our COM objects. COM does not require that you use C++ for its implementation, but, as you will see, because of the internal structure of COM objects C++ is the preferred method.

Component Interfaces

One important aspect of a software component is its interface. The component's interface is its contract with the user, and it describes exactly what can be done. As we've discussed, the other important feature of a software component is complete encapsulation of its implementation details. COM provides this capability by defining a standard way of implementing the interfaces of a COM object. Recall the public interface of the `Expression` class:

```
class Expression : public CObject
{
...
// Here's the interface
public:
   CString  GetExpression();
   void     SetExpression( CString strExp, BOOL bInfix );

   BOOL     Validate();
   long     Evaluate();
};
```

Because the interface is the essence of our component, we need a binary standard way of making it available to non–C++ users. The COM, OLE, and ActiveX specifications are composed mostly of interfaces. Once you understand the concept of a COM interface, the rest is easy. The following example code shows how a simple COM interface is implemented in C++:

```
// public interface definition of our Expression component
// An abstract class
class IExpression
{
public:
   virtual CString GetExpression() = 0;
   virtual void SetExpression( CString str, BOOL bInfix ) = 0;
   virtual BOOL Validate() = 0;
   virtual long Evaluate() = 0;
};

// Inherit from the abstract Interface definition class
class Expression : public IExpression
{
// Class Implementation here, just as before
...
// Here's the interface implementation
public:
   virtual CString  GetExpression();
   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
};
```

First, we declare a new class that is abstract and contains only the `public` functions of our class. All interface functions are declared pure virtual. This new class is called `IExpression`; the *I* indicates that it is an interface declaration. COM uses this nomenclature throughout its implementation. The `IExpression` class provides an external interface declaration for our `Expression` object. It also forces the creation of a virtual function table (Vtable) in any derived classes.

This polymorphic use of virtual functions in a base class is central to the design of COM. It provides a Vtable that contains only the public methods (the interface) of the class. The `IExpression` class contains no data members and no implementation functions. Its only purpose is to force the derived class, `Expression`, to implement, virtually, the methods of the component interface.

Because we allow access only through a Vtable pointer, access to the members of the component implementation is impossible. Only those functions declared in the interface class are available to the component user. Study the preceding example. It is fairly small and simple, and it contains the core concept of COM: the

use of Vtables to hide the implementation of component interfaces. A COM interface is just a pointer to a pointer to a C++-style Vtable. This relationship is illustrated in Figure 4.1.

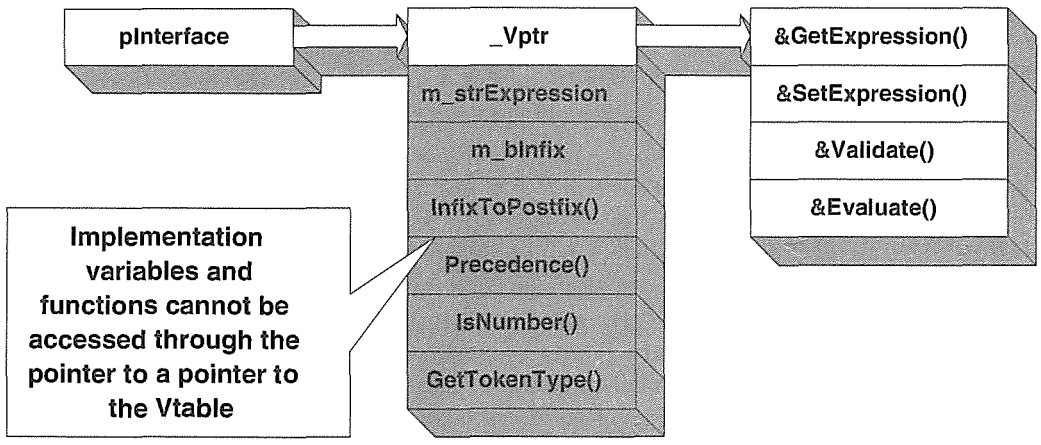


Figure 4.1 COM Vtable usage.

Following is an example of how a client application might use the `IExpression` interface provided by the `Expression` class. This example isn't a true implementation of COM but rather is simplified to show, conceptually, how COM works.

```
// An identifier for our Expression class
#define CLSID_Expression 1000

// This provides a standard method of obtaining
// an instance pointer for the specified object
RESULT GetObject( OBJID objId, void** ppv )
{
   if (  objId == CLSID_Expression )
   {
      *ppv = new Expression;
      return ( S_OK );
   }
   else
     return( E_INVALID_OBJECT_ID );
}

void RemoveObject( void* pv )
{
   delete pv;
}
```

```
main()
{
   IExpression* pIExpression;
   if ( GetObject( CLSID_Expression, (void**) &pIExpression ) == S_OK )
   {
      pIExpression->SetExpression( "1 + 2", TRUE );
      cout << pIExpression->Evaluate() << endl;
      RemoveObject( pIExprBAD );
   }
}
```

Given our previous `Expression` declarations, we implement a function, `GetObject`, that takes as a parameter an object ID that identifies the specific component that we need and returns a pointer to that component object. As illustrated in the `main()` section, the user of the component requires knowledge only of the `IExpression` interface to use the object effectively. The `GetObject` function could possible reside outside the client's executable or even outside the client's process. The `RemoveObject` function is called when the client is finished with the component. `RemoveObject` is responsible for deleting the component instance when all clients are finished using it. But we're getting ahead of ourselves.

Standard COM Interfaces

The preceding example is not yet a COM object. COM requires that at least one standard interface be present in an object to qualify it as a COM object. `IUnknown` is the one interface that all COM objects must support. `IUnknown` serves two purposes. The first is to provide a standard way for the component user (or client) to ask for a specific interface within a given component. `QueryInterface`, a method of `IUnknown`, provides this capability. The second purpose is to help in the management of the component's lifetime. The `IUnknown` interface provides two methods—`AddRef` and `Release`—that provide lifetime management of a component instance. Here is the definition of `IUnknown`:

```
class IUnknown
{
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv ) = 0;
   virtual ULONG    AddRef() = 0;
   virtual ULONG    Release() = 0;
};
```

As you can see, `IUnknown` is an abstract class that provides the requirements for all classes that derive from it. It mandates that the three methods be implemented in the deriving class. It also ensures that the deriving class will have a Vtable, just as in the `IExpression` interface we examined earlier.

`QueryInterface` returns a pointer to a specific interface (such as `IUnknown` or `IExpression`) contained within a COM object. The first parameter, `REFIID`, is a reference to the specific interface ID, which is a unique identifier for the interface we are querying for. The second parameter, `void**`, is the location

where the interface pointer will be returned. The return value, an HRESULT, is a handle to a COM-specific error structure that contains any error information. We'll get to the details later; right now we're trying to grasp the big picture.

In our example, if a user requires the services of the Expression class, he or she would request the IExpression interface. This request can be made through an existing IUnknown interface (on an Expression component) or can be requested during the component's instantiation. COM requires that the IUnknown interface be present in any COM object and that all COM interfaces also contain the IUnknown interface. This arrangement allows a component user to obtain an interface pointer to any interface within the component by querying any existing interface on that component. Here's what our example looks like with the IUnknown interface added:

```
// public interface definition
// An abstract class that derives from IUnknown
class IExpression : public IUnknown
{
public:
   virtual CString GetExpression() = 0;
   virtual void SetExpression( CString str, BOOL bInfix ) = 0;
   virtual BOOL Validate() = 0;
   virtual long Evaluate() = 0;
};

// Derive from the abstract Interface definition class
class Expression : public IExpression
{
// Class Implementation here, just as before
...
// Here's the interface implementation
public:
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();

   virtual ULONG    AddRef();

   virtual CString  GetExpression();
   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
};
```

The addition of deriving IExpression from IUnknown requires us to now implement seven methods: three from IUnknown and the four original Expression class methods. We now have a Vtable in our class that looks like Figure 4.2.

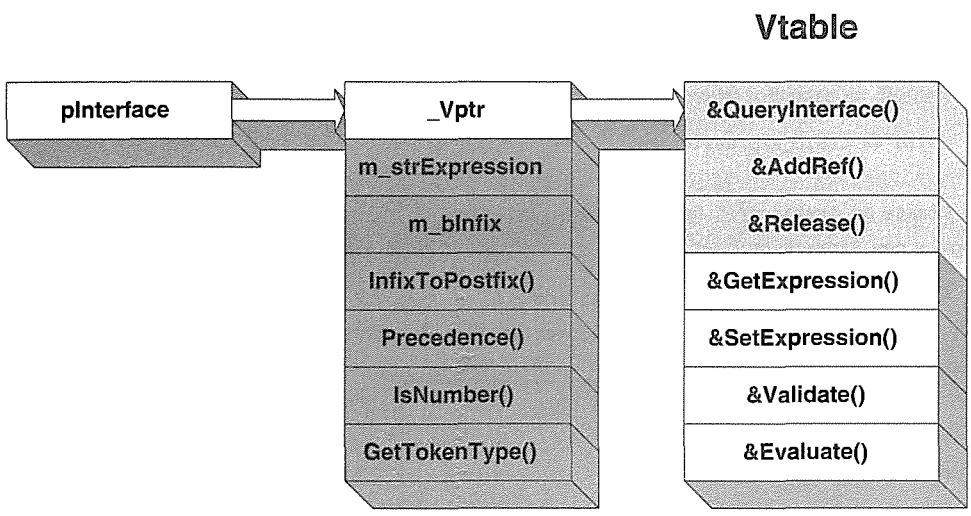


Figure 4.2 Expression class with `IUnknown` added.

Our example now looks like this. We've added the implementation of `IUnknown::QueryInterface`.

```
// The identifier for the COM IUnknown interface
#define IID_IUnknown 1
// An identifier for our Expression class
#define CLSID_Expression 1000
// An identifier for our IExpression interface
#define IID_IExpression 2000

HRESULT Expression::QueryInterface( REFIID riid, void** ppv )
{
   // Our Expression class contains both of these
   // interfaces so returning a pointer to this provides
   // both implementations. The client will cast the returned
   // pointer to the appropriate pointer type.
   switch( riid ) {
     case IID_IUnknown:
     case IID_IExpression:
        *ppv = this;
        return( S_OK );
     default:
        return( E_NOINTERFACE );
   }
}
```

```
// This provides a standard method of creating a component
// object and returning the requested interface on that object
RESULT GetObject( OBJID objId, REFIID riid, void** ppv )
{
   if (  objId == CLSID_Expression )
   {
      // Create an instance of Expression
      *ppv = new Expression;

      // Now query for the requested interface
      ( ( Expression* ) ( *ppv ) )->QueryInterface( riid, ppv );
      return ( S_OK );
   }
   else
     return( E_INVALID_OBJECT_ID );
}

void RemoveObject( void* pv )
{
   delete pv;
}

main()
{
   IUnknown*    pIUnknown;
   IExpression* pIExpression;

   // Create an Expression component and get its
   // IUnknown interface
   GetObject( CLSID_Expression, IID_IUnknown, (void**) &pIUnknown );
   // Now use the IUnknown interface to get the
   // IExpression interface
   pIUnknown->QueryInterface( IID_IExpression, (void**) &pIExpression );

   // Now use the IExpression interface pointer
   pIExpression->SetExpression( "1 + 2", TRUE );
   cout << pIExpression->Evaluate() << endl;

   RemoveObject( pIExpression );
};
```

The implementation of `QueryInterface` may be a little difficult to comprehend at this point. All it does is to return a pointer to the Vtable pointer for the `Expression` object. Because `IExpression` is derived from `IUnknown`, a pointer to the `Expression` object provides both interfaces. The preceding code won't actually compile and run—a few things are still missing—but I hope you're getting the idea.

There is a standard way of depicting COM objects and their interfaces. Figure 4.3 depicts our `Expression` class with its `IExpression` and `IUnknown` interfaces. `IUnknown` is shown on the upper right, because it is always required and so will be present in any COM object. Other interfaces are usually shown on the left-hand side of the component. Remember, though, that every interface on the left contains an `IUnknown` interface, too. Every COM interface implements `IUnknown`.

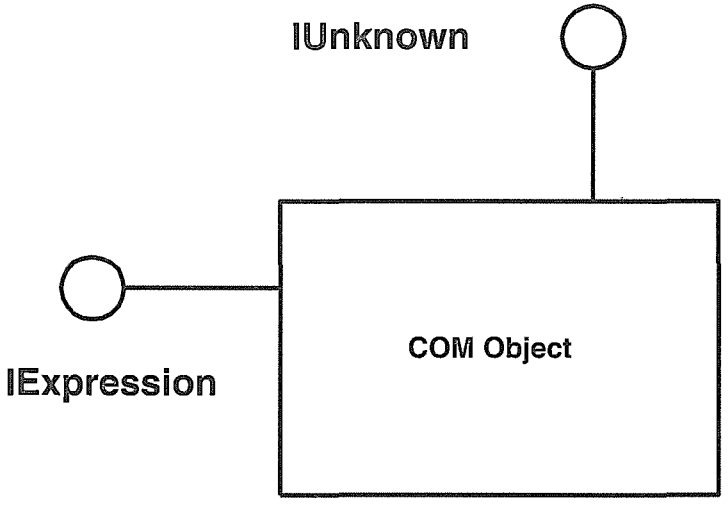


Figure 4.3 COM object representation.

Component Lifetimes

We've learned that access to a component interface is obtained through `IUnknown::QueryInterface`. The other two functions provided by `IUnknown` are used to control the lifetime of the component object. This process is termed *reference counting*.

Each component class contains a member variable, usually named `m_dwRef`, that maintains a count of the outstanding references to (or clients of) its COM interfaces. The interface user has only a pointer to a pointer to the object and so cannot directly delete the object when finished. In reality, the client shouldn't think of deleting the object anyway, because others may be using the same component object. Only the COM object can determine when it can or should be deleted. The object's reference count is controlled by the client using the `AddRef` and `Release` functions of `IUnknown`. This is one reason every COM interface must contain an implementation of `IUnknown`. Let's use our simple `Expression` example to describe the reference counting requirements.

```
class IUnknown
{
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv ) = 0;
   virtual ULONG    AddRef() = 0;
   virtual ULONG    Release() = 0;
};

class IExpression : public IUnknown
{
public:
   virtual CString GetExpression() = 0;
   virtual void SetExpression( CString str, BOOL bInfix ) = 0;
   virtual BOOL Validate() = 0;
   virtual long Evaluate() = 0;
};

class Expression : public IExpression
{
// Class Implementation here, just as before.
...
   // Add a new member variable to keep track of the
   // outstanding interface references to an instantiation
   // Initialise to zero during construction
   DWORD m_dwRef;

public:
   // Nothing has changed here
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

   virtual CString  GetExpression();
   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
};

HRESULT Expression::QueryInterface( REFIID riid, void** ppv )

{
   switch( riid ) {
     case IID_IUnknown:
```

```
    case IID_IExpression:
       *ppv = this;
       // We're returning a new interface pointer
       // so call AddRef on the new pointer
       ( (LPUNKNOWN) *ppv)->AddRef();
       return( S_OK );
    default:
       return( E_NOINTERFACE );
  }
}

Expression::Expression ()
{
    m_dwRef = 0;
    m_bInFix = TRUE;
}

// IUnknown::Release implementation
ULONG Expression::Release()
{
  m_dwRef--;
  // When the reference count reaches zero
  // delete ourselves
  if ( m_dwRef == 0 )
  {
     delete this;
     // Can't return m_dwRef, it's gone
     return 0;
  }
  else
     return m_dwRef;
}

// IUnknown::AddRef implementation
ULONG Expression::AddRef()
{
  m_dwRef++;
  return m_dwRef;
}
```

To support the AddRef and Release functions of IUnknown we added a member variable, m_dwRef, that keeps a count of the current references, or outstanding interface pointers, to our object. The AddRef and Release functions directly affect a COM interface, but an interface is not an instance of the object itself. This object can have any number of users of its interfaces at a given time and must maintain an internal count of its active interfaces. When this count reaches zero, the object is free to delete itself. It is important that object users diligently AddRef and Release the interfaces when appropriate. If they don't, the component will hang around forever (or until a reboot, whichever comes first). The AddRef, Release pair is similar to the new, delete pair used to manage memory in C++. Whenever a user obtains a new interface pointer or assigns its value to another variable, AddRef should be called through the new pointer. You must be careful, though, because many COM interface functions return pointers to interfaces; in these cases, the functions call AddRef through the returned pointer. The most obvious example is QueryInterface. It always calls AddRef after allocating a new interface, so it isn't necessary to call AddRef again. Here are some examples:

```
// The initial CoCreateInstance calls QueryInterface internally.
// We'll study this COM function in a moment. It instantiates a
// COM object of the requested type and returns an interface pointer
// to the created object.
HRESULT hr;
hr = CoCreateInstance( ID_Expression, &lpExpression );
if ( FAILED( hr ) )
   return;

// At this point m_dwRef = 1

// Get an IUnknown pointer (QI calls AddRef internally)
hr = lpExpression->QueryInterface( IID_IUnknown, &lpUnknown );
if ( FAILED( hr ))
{
   // The call failed, so decrement m_dwRef to 0 by calling
   // release through the initial pointer and return.
   lpExpression->Release();
   return;
}

// If everything worked, m_dwRef = 2

// Make a local copy of the interface
LPUNKNOWN lpU2 = lpUnknown;
lpU2->AddRef();

// Now m_dwRef = 3

// use lpU2
// free the interface
```

```
luU2->Release();
// The release decrements m_dwRef so it is now 2
// No longer valid, set to NULL
lpU2 = NULL;

{
   Expression* lpExp2 = lpExpression;
   lpExp2->AddRef();  // m_dwRef = 3
   ... Use lpExp2
}
// We screwed up here. lpExp2 was destroyed as
// it went out of scope, but we did not call
// Release() so the reference count is still 3.

lpUnknown->Release();     // m_dwRef = 2
lpExpression->Release(); // m_dwRef = 1
// The object lives forever
```

Like the C++ `new` and `delete` operators, `AddRef` and `Release` calls should "match up." The exception is `QueryInterface`. Treat a call to `QueryInterface` as a call to `AddRef`. `QueryInterface` allocates a new interface internally and so must call `AddRef`.

Multiple Interfaces

COM objects nearly always contain multiple interfaces. If a COM object had only one interface, it would have to be `IUnknown`, and that wouldn't provide very much functionality. Some of the interfaces implemented by an object (such as `IUnknown`) are there to support COM's purpose of providing the binary standard wrapper around the component's functionality. By exposing the component's distinct behavior, the other interfaces are what make the component useful. In many cases, these interfaces are already defined by the COM, OLE, or ActiveX standard. Your job is to provide a distinct implementation of the interface. In other cases, you will create your own custom interfaces to provide specific behavior. We will discuss both approaches in the next few chapters.

In C++, there are three ways to implement multiple interfaces in a COM object: multiple inheritance, interface implementation, and C++ class nesting. Of the three methods, we will focus on and use C++ class nesting, because that is how MFC implements COM/OLE interfaces. C++ class nesting is the hardest to understand (at first), but it is the most efficient. Let's briefly look at each method. In Chapter 5, we will go into more detail as we discuss MFC's COM/OLE implementation.

Declaring multiple interfaces within a single C++ class doesn't sound very difficult at first, but it can be tricky. Because COM interfaces are really pointers to a C++-style Vtable, a multiple interface class requires multiple Vtables. Another reason that component classes and their interface implementation classes must be closely coupled (via class nesting) is the need to maintain reference counting. When there are multiple inter-

faces on a COM object, all of them must cooperate in the reference counting scheme. There is only one reference count per object, and the interfaces must share it. This coupling is maintained through multiple `IUnknown` interfaces.

Multiple Inheritance

First, let's look at how multiple inheritance can solve the multiple Vtable problem. We'll continue the `Expression` class example by adding an interface, `IExpression2`. One of the useful features of COM is the ability it gives you to maintain multiple versions of an interface for a particular component class. This arrangement allows easier upgrading and augmenting of components. When developers enhance an existing component, they can provide a new interface that exposes the new functionality. At the same time, the original interface can be maintained. This approach ensures that any applications developed with the original interface will continue to work, and the component user can migrate to the more functional interface as time permits.

The new `IExpression2` interface will provide a new feature that `IExpression` did not. Its declaration follows.

```
class IExpression : public IUnknown
{
   virtual CString GetExpression() = 0;
   virtual void    SetExpression( CString strExp, BOOL bInfix ) = 0;
   virtual BOOL    Validate() = 0;
   virtual long    Evaluate() = 0;
};

// New interface derives from the old one
class IExpression2 : public IExpression
{
   virtual CString ErrorString() = 0;
};
```

The new method, `ErrorString`, returns a textual description (such as "Too many closing parentheses.") of any errors that occur during the call to `Validate`. If `Validate` returns `FALSE`, the `ErrorString` method is called to obtain a description of the problem, which is then presented to the user. We would like to provide both interfaces for our COM object so that the user can choose the more appropriate one. We can deploy the new component without breaking any existing applications that use it. Using multiple inheritance, we would declare something like this:

```
class Expression : public IExpression, public IExpression2
{
   // implementation of all three interfaces
   // IUnknown
   // IExpression
   // IExpression2
};
```

To illustrate one of the difficulties of using multiple inheritance to implement multiple interfaces, I've written the `QueryInterface` function for our declaration. One of the ambiguities of using multiple inheritance is that the multiple Vtable implementations are handled by typecasting to the appropriate interface declaration. The value of `this` (and therefore its Vtable pointer) varies depending on the required casting. Both `IExpression` and `IExpression2` contain an `IUnknown` interface, so either one could be used to return `IUnknown`.

```
HRESULT Expression::QueryInterface( REFIID riid, void** ppv )
{
   *ppv = NULL;

   if ( riid == IUnknown || riid == IExpression )
      *ppv = (IExpression *) this;
   else if ( riid == IExpression2 )
      *ppv = (IExpression2 *) this;
   else
       return( E_NOINTERFACE );

   ((LPUNKNOWN) *ppv)->AddRef();
   return( S_OK );
}
```

Multiple inheritance may work in some circumstances, but in our case we have a severe "name collision" problem. The two `Expression` interface classes—`IExpression` and `IExpression2`—contain method names that are the same. (For example, `GetExpression` is contained in both classes.) This ambiguity makes multiple inheritance difficult to use and complex to implement. The `IUnknown` interface is also ambiguous, making it difficult to implement a COM-based reuse technique called *aggregation*, which we will discuss in a moment. In other circumstances, where there are no name collisions and aggregation is not a requirement, multiple inheritance can be an effective technique.

Interface Implementations

Another option is to use interface implementations, which use multiple classes that contain pointers to the main class that implements the component's behavior. As you will see, the purpose of the interface implementation classes is to provide a Vtable for the interface they define. They delegate all their `IUnknown` functions to the main class. To promote clarity, error handling is not included.

```
// Classes that implement each COM interface but
// delegate their implementations of IUnknown
class ImpExpression : public IExpression
{
   Expression* m_pExpression;
public:
```

```
   // Constructor
   ImpExpression( Expression* pExp ) { m_pExpression = pExp; }

   // Interface
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

   virtual CString  GetExpression();
   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
};

// Classes that implement each COM interface but
// delegate their implementations of IUnknown
class ImpExpression2 : public IExpression2
{
   Expression* m_pExpression;
public:
   // Constructor
   ImpExpression2( Expression* pExp ) { m_pExpression = pExp; }

   // Interface
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

   virtual CString  GetExpression();
   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
   virtual CString  ErrorString();
};

class Expression : public IUnknown

{
...
// All basically the same as before except for the
// addition of pointers for the implementation classes
   ImpExpression*   m_pImpExpression;
   ImpExpression2*  m_pImpExpression2
```

```
...
// Only IUnknown functions are implemented in the base
// or parent class.
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

// Make the Interface Implementation classes
// friends so that they can access protected members
   friend class IExpression;
   friend class IExpression2;
};

// The constructor for Expression
// must create the Interface Implementation objects
Expression::Expression()
{
   // Instantiate the Implementations
   m_pImpExpression = new ImpExpression( this );
   m_pImpExpression2 = new ImpExpression2( this );
}
// Destructor must remove the Interface Implementation instances
Expression::~Expression()
{
   delete m_pImpExpression;
   delete m_pImpExpression2;
}

// QueryInterface is a little special
HRESULT Expression::QueryInterface( REFIID riid, void** ppv )
{
   if ( riid == IID_IUnknown )
      *ppv = this;
   else if ( riid == IID_IExpression )
      *ppv = m_pImpExpression;
   else if ( riid == IID_IExpression2 )
      *ppv = m_pImpExpression2;
   else
      return( E_NOINTERFACE );

   ((LPUNKOWN) *ppv)->AddRef();
   return( S_OK );
}
```


NOTE

By declaring another class as a `friend` class, you explicitly give that class access to your class's `private` and `protected` members. In the preceding example, we declare the implementation classes as `friend`s so that they can access the implementation members of the `Expression` class.

The `Expression` class now inherits only from the `IUnknown` interface and delegates to other classes the implementation of the behavior-specific interfaces. These classes are made `friend` classes so they can access the protected variables and methods of `Expression` for implementation of the exposed functionality. `Validate` is implemented in the other classes but still must access `m_strExpression`, `InfixToPostfix`, and so on. `AddRef` and `Release` are the same as before, but `QueryInterface` now returns a pointer to the Vtable of the specified interface implementation class. The lifetimes of the interface implementation objects are controlled by `Expression`. When an `Expression` instance is deleted, its destructor also removes the interface implementation class instances.

```
// Classes that implement each COM interface but
// delegate their implementations of IUnknown.
class ImpExpression : public IExpression
{
   Expression* m_pExpression;

public:
   // Constructor
   ImpExpression( Expression* pExp ) { m_pExpression = pExp; }

   // Interface
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

   virtual CString  GetExpression();

   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
};

HRESULT ImpExpression::QueryInterface( REFIID riid, void** ppv )
{
   // delegate to the main class
   return m_pExpression->QueryInterface( riid, ppv );
}
DWORD ImpExpression::AddRef()
{
   return m_pExpression->AddRef();
}
```

```
DWORD ImpExpression::Release()
{
   return m_pExpression->Release();
}

BOOL ImpExpression::Validate()
{
   ...
   // Same as all previous examples, but we now must reference the
   // base or parent members explicitly. Because we are friends, we
   // have direct access to the protected members.
   Tokenizer.SetString( m_pExpression->m_strExpression );
   Tokenizer.Tokenize();
   ...
   // Check for validity
   while( Tokenizer.GetToken( strToken ) )
   {
      switch( m_pExpression->GetTokenType( strToken ) )
      {
      ...
   ...
}
```

To maintain a correct reference count for the component object as a whole, the implementation class's `IUnknown` methods defer to the parent class implementation. This practice ensures that there is only one point where reference counting occurs. `QueryInterface` also defers to the `Expression` class implementation, because the `Expression` class contains all the appropriate Vtable pointers. The only change required to the implementation of the `IExpression` interface methods is to use the `m_pExpression` pointer to directly access the needed `Expression` members. Making them `friend` classes of the base class makes this change easy.

```
class ImpExpression2 : public IExpression2
{
   // Pointer to controlling object
   Expression* m_pExpression;
public:
   // Constructor
   ImpExpression2( Expression* pExp ) { m_pExpression = pExp; }

   // Interface
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
```

```
   virtual ULONG    AddRef();

   virtual CString  GetExpression();
   virtual void     SetExpression( CString strExp, BOOL bInfix );
   virtual BOOL     Validate();
   virtual long     Evaluate();
   virtual CString  ErrorString();
};

HRESULT ImpExpression2::QueryInterface( REFIID riid, void** ppv )
{
   // delegate to the main class
   return m_pExpression->QueryInterface( riid, ppv );
}
BOOL ImpExpression2::Validate()
{
   ...
   // Same as all previous examples, but we now must reference the
   // base or parent members explicitly. Because we are friends, we
   // have direct access to the protected members.
   Tokenizer.SetString( m_pExpression->m_strExpression );
   Tokenizer.Tokenize();
   ...
   // Check for validity
   while( Tokenizer.GetToken( strToken ) )
   {

      switch( m_pExpression->GetTokenType( strToken ) )
      {
      ...
   ...
}
```

In practice, you wouldn't typically implement the common interface functions (such as `Validate` and `SetExpression`) in both implementation classes. The only difference between the two interfaces is the addition of the `ErrorString` method, so the shared class methods would be implemented by a common class shared by the two interface implementation classes. This would also be the case in the class nesting example that we will demonstrate next.

The implementation of the second interface is similar to that of the first interface, `IExpression`, so it isn't detailed here. I've included an implementation of `QueryInterface` to show that it, too, delegates to the parent class.

Interface implementations are an easily understandable method of providing multiple COM interfaces. Interface implementations are easily managed if you have only a small number of interfaces, but in a class with more than a few interfaces, this approach can become complex. For a more detailed look at multiple inheritance and interface implementations, see Brockschmidt's *Inside OLE*. Our focus is on MFC's method: class nesting.

C++ Class Nesting

Instead of using multiple inheritance or interface implementations, the MFC libraries use a technique called *class nesting*. This approach provides multiple Vtables for the single nesting class by exposing the Vtables of the nested classes. This method also allows for easy management of reference counting. It is similar to interface implementations but does not require the use of forward and back pointers between classes. Class nesting saves eight bytes for each COM interface. Using class nesting, our example now looks like this:

```
class IExpression : public IUnknown
{
public:
   virtual CString GetExpression() = 0;
   virtual void SetExpression( CString str, BOOL bInfix ) = 0;
   virtual BOOL Validate() = 0;
   virtual long Evaluate() = 0;
};

class IExpression2 : public IExpression
{
public:
   virtual CString ErrorString() = 0;
};

class Expression : public IUnknown
{
// Class Implementation here, just as before
...
   // Only IUnknown implemented in main class
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

   class XExpression : public IExpression
   {
   public:
      virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
```

```
      virtual long     Release();
      virtual long     AddRef();

      virtual CString  GetExpression();
      virtual void     SetExpression( CString strExp, BOOL bInfix );
      virtual BOOL     Validate();
      virtual long     Evaluate();
   } m_xExpression;

   friend class XExpression;

   class XExpression2 : public IExpression2
   {
   public:
      virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
      virtual long     Release();
      virtual long     AddRef();

      virtual CString  GetExpression();
      virtual void     SetExpression( CString strExp, BOOL bInfix );
      virtual BOOL     Validate();
      virtual long     Evaluate();
      virtual CString  ErrorString();
   } m_xExpression2;
   friend class XExpression2;

};
```

Using nested classes, the implementations of the COM interfaces are contained in classes that are "nested" within the main `Expression` class. Each nested class has a Vtable, so there is a Vtable for each COM interface (just what we need). The implementations of the `IUnknown` methods are similar to the interface implementation technique described earlier. The members `m_xExpression` and `m_xExpression2` are embedded instances of the nested classes. Whenever an instance of the `Expression` class is created, two embedded instances of the nested class are created along with it. The embedded instances can be accessed only within the nesting class, because they exist only within its scope. Here's the `QueryInterface` for the `Expression` class:

```
HRESULT Expression::QueryInterface( REFIID riid, void** ppv )
{
    *ppv = NULL;

    if ( riid == IID_IUnknown )
        *ppv = this;
    else if ( riid == IID_IExpression )
        *ppv = &m_xExpression;
```

```
    else if ( riid == IID_IExpression2 )
        *ppv = &m_xExpression2;
    else
        return( E_NOINTERFACE );

    if ( *ppv )
    {
        ((LPUNKNOWN)*ppv)->AddRef();
        return S_OK;
    }
    return E_NOINTERFACE;
}
```

This technique is relatively straightforward. If the user requests an `IUnknown` interface, the Vtable pointer for the main class is returned. If either of the `IExpression` interfaces is required, a pointer to the appropriate nested class instance is returned. The Vtable pointer, `_vptr`, is the first entry of any C++ instance with virtual functions, so the address of an embedded class instance (such as `m_xExpression`) is a pointer to a pointer to the Vtable. The `AddRef` and `Release` functions for the nesting class are the same as in the interface implementation method, but they are slightly different for the nested classes, as we will see.

To understand the implementation of the nested classes, we need to review a couple of C++ features. To define a nested class member, you use the multiple scoping operators:

```
HRESULT Expression::XExpression::QueryInterface(...);
```

This code enables you to "scope" down to the nested class function. The other item is the C++ `offsetof` macro. The `offsetof` macro is used to address the base class members directly from inside the nested classes. The `offsetof` macro calculates the difference between a class member and the starting address of the class. MFC uses the `offsetof` macro to eliminate the need for a pointer to the outer class. We will discuss this in more detail in Chapter 5 when we detail MFC's implementation of OLE.

MFC defines a C macro, `METHOD_PROLOGUE`, that uses the `offsetof` macro to address back into the main class. It is defined like this:

```
#define METHOD_PROLOGUE(theClass, localClass) \
      theClass* pThis = \
            ((theClass*)((BYTE*)this - offsetof(theClass, m_x##localClass)));

METHOD_PROLOGUE( Expression, Expression )
// Expands to the following
Expression* pThis =
            ((Expression*)((BYTE*)this -
            offsetof( Expression, m_xExpression)));
```

When implementing the methods within the nested classes, you use this macro to initialize the `pThis` pointer, which allows access to the nesting ("outer") class members. Here is the implementation of `XExpression`'s `QueryInterface` and `AddRef` functions:

```
HRESULT Expression::XExpression::QueryInterface( REFIID riid, void** ppv )
{
   METHOD_PROLOGUE( Expression, Expression );
   return pThis->QueryInterface( riid, ppv );
}

ULONG Expression::XExpression::AddRef()
{
   METHOD_PROLOGUE( Expression, Expression );
   return pThis->AddRef();
}
```

`pThis` behaves like `this` for the main class, so `pThis` can be used to access the member functions as well as the member variables of `Expression`. Figure 4.4 illustrates how the `METHOD_PROLOGUE` macro calculates the `pThis` pointer. Here's how we implement the `Validate` function:

```
long Expression::XExpression::Validate()
{
   CStringStack stack;
   Tokenizer tokenizer;
   CString strToken;
   CString strTop;

   METHOD_PROLOGUE( Expression, Expression );

   tokenizer.SetString( pThis->m_strExpression );
   tokenizer.Tokenize();

   // Check for validity
   while( tokenizer.GetToken( strToken ) )
   {
      ...
   }
   ...
   return FALSE;
}
```

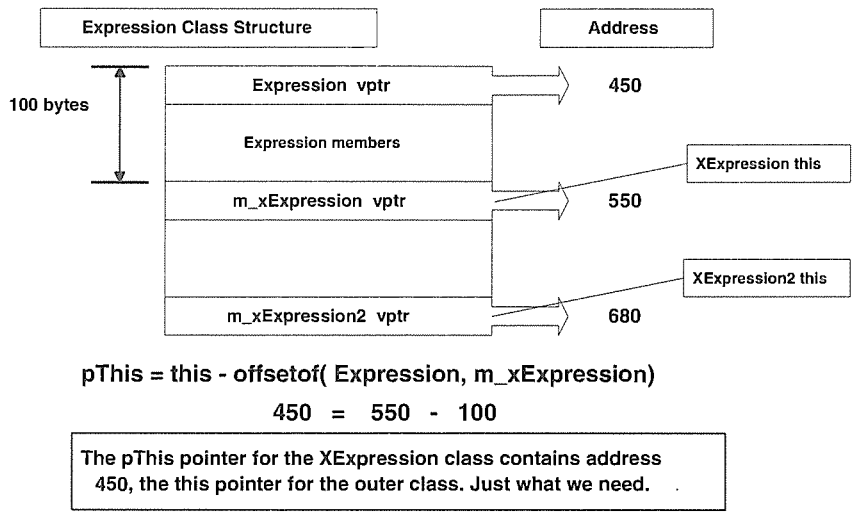


Figure 4.4 `pThis` calculation.

As you can see, the `pThis` pointer enables us to directly access the members of the outer, nesting class. Class nesting is the method used by MFC to implement multiple interfaces within one COM object. This is an introduction to the technique that we will cover again in Chapter 5.

GUIDs

With multiple component-based applications running on a component-based operating system, there can be hundreds of different components. In such an environment, it is imperative that there be a way to uniquely distinguish each component. COM uses the method described in the Open Software Foundation Distributed Computing Environment Standard for Remote Procedure Calls. This standard describes a *universally unique identifier* (UUID). Composed of 128 bits, the UUID is statistically guaranteed to be unique in every situation. It combines a unique network address (from your network card) with the then-current date and time. COM calls this value a *globally unique identifier*. GUIDs are used within COM to identify interfaces, component classes, type libraries, and property pages (we'll discuss the last two later). Here's an example of a GUID:

```
a988bd40-9f1a-11ce-8b9f-10005afb7d30
```

There are two primary uses of GUIDs. The first, a CLSID (class ID) is used to uniquely identify a specific COM component class. The second, an IID (interface ID) is used to uniquely identify a specific interface type (such as `IUnknown` or `IExpression`). The `DEFINE_GUID` macro is used to assign a GUID to a variable that is easier to use programmatically.

```
DEFINE_GUID( CLSID_Expression,
             0xA988BD40,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);
```

```
DEFINE_GUID( IID_IExpression,
             0xA988BD41,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);
```

It's important that the identifier be defined only once for the given executable or DLL. At the one point in your program where you intend to define (not declare) the identifier, you must include **INITGUID.H** before the **.H** file that includes the declarations. Here's what it looks like in our `Expression` example:

```
// ExpSvr.cpp
...

#include "stdafx.h"

#include <initguid.h>
#include "ExpSvr.h"

#include <stdio.h>
```

By including **INITGUID.H**, you change the meaning of the `DEFINE_GUID`. It no longer just declares the GUID's variable; it also defines and initializes it.

In our example, we defined two GUIDs: one is a CLSID that uniquely identifies the specific component class `Expression`, and the other is an IID to identify our new interface, `IExpression`. There are a few ways to generate GUIDs for use in your applications. The easiest way is to let MFC's ClassWizard or AppWizard generate a GUID whenever you add a new component or interface. Another way is to use the `CoCreateGuid` function provided by the COM API. But if you're working on a project that will use multiple GUIDs (as most do), you will probably want to generate a range of GUIDs. For the examples in this book, I ran the **UUIDGEN.EXE** utility provided with Visual C++ as follows:

```
c:\msdev\bin\uuidgen -n50 > bookids.txt
```

This produced 50 sequential GUIDs that we're using right now. The primary benefit of using sequential GUIDs is that they're easier to look up in the Registry, which we'll discuss in the next section.

There is one other way to generate GUIDs. Visual C++ provides the GUIDGEN.EXE program, which generates a single GUID at a time. It has a neat feature that formats the GUID as a C++ define so you can quickly paste it into your program code.

The COM API provides functions to make dealing with GUIDs much easier. Table 4.1 gives a quick view of what COM provides. A good reference for these functions and macros is the *OLE 2 Programmer's Reference, Volumes 1 and 2*, which are also available on-line with Visual C++: position the cursor over the function name and press **F1**.

Table 4.1 Useful GUID Helper Functions

Function	Purpose
`CoCreateGuid( GUID* pGuid )`	Programmatic way of generating one unique GUID
`IsEqualGUID( REFGUID, REFGUID )`	Compares two GUIDs
`IsEqualIID( REFIID, REFIID )`	Compares two IIDs
`IsEqualCLSID( REFCLSID, REFCLSID )`	Compares two CLSIDs
`CLSIDFromProgID( LPCOLESTR, LPCLSID )`	Returns the CLSID for the given ProgID
`ProgIDFromCLSID( REFCLSID, LPOLESTR*)`	Returns the ProgID from the CLSID

Note: 16-bit COM uses standard C style strings (LPSTR) and 32-bit OLE uses the Win32 Unicode standard (LPOLESTR) for all of its string handling. We'll discuss converting from ANSI to Unicode strings (and back) in the next few chapters.

We have many functions for manipulating GUIDs within a C or C++ program, but what about developers using Visual Basic (or a similar language)? Must they explicitly provide the 128-bit number to identify the specific component class that they require? No. COM provides a more human-readable method of accessing a component class. It's called the *program ID* (ProgID). ProgIDs are simple text strings that are associated, through the Windows Registry, with a specific CLSID. For our example I've chosen a ProgID of `Chap4.Expression.1`. This allows a Visual Basic developer to the following:

```
Dim obj as Object
Set obj = CreateObject( "Chap4.Expression.1" )
obj.SetExpression( "10 + 10 * 10" )
text1 = obj.Evaluate()
```

The `CreateObject` statement takes a ProgID that identifies the specific component. The number following `Expression` is used to indicate a version-specific component. One of the powerful features of COM is the ability it gives you to expose multiple versions of a component interface, thus making component upgrades and feature additions easy. How does Visual Basic know how to find and create the `Expression` component? It uses `CLSIDFromProgID`, which queries the Windows Registry.

The Windows Registry

The Windows Registry in Windows 95 and Windows NT plays a significant role in the management and configuration of the operating system. It includes system configuration information, user-specific information, and information about installed hardware and software. The Registry replaces many of the files that we're familiar with in Windows 3.1: **WIN.INI**, **CONFIG.SYS**, **SYSTEM.INI**, and so on. All these files and the software-specific **.INI** files that typically resided in the Windows directory have been combined into the Registry. This arrangement makes it much easier to manage multiple machines in a LAN environment. Both Windows 95 and Windows NT allow the use of a shared, or networked, Registry, eliminating the accumulation of hundreds of **.INI** files in the Windows directory that occurs under Windows 3.x.

We'll explore the features of the Registry because COM and ActiveX depend heavily on the functionality provided by it. CLSIDs for COM objects and various options associated with them are stored in the Registry. This provides nonvolatile storage of COM information that is required to "bootstrap" various COM executables and DLLs.

There are several ways to update the Registry with the CLSID information of our COM objects. The first approach is to create a text file of the form shown later in this section and then "merge" the information into the Registry using functions provided in the REGEDIT program. The second technique is to programmatically update the Registry using the Win32 Registry API functions. The third method is to manually add and edit the fields. Although this may be necessary when you're fixing a problem, it is not recommended. (A fourth method is to let MFC do it for you, as we'll see in Chapter 5.)

The Registry orders system information in a hierarchical manner, with several predefined top, or root, levels. The one we're interested in is `HKEY_CLASSES_ROOT`. This section of the Registry stores COM/ActiveX information, shell information such as file extension associations, and other software-related items. Information stored in the Registry is of the form `key = value`. This arrangement allows easy lookup of a value associated with a key in this hierarchical structure. The value is optional and is sometimes used when only the presence or absence of the key is important. Examples include the `Insertable`, `NotInsertable`, and `Control` keys that we will use.

An important subkey under `HKEY_CLASSES_ROOT` is `CLSID`. Under this subkey are all the public component CLSIDs currently registered on your system. All the COM system CLSIDs, any that are installed by third-party software, and those we will use in our projects are registered here. Figure 4.5 shows the NT version of REGEDIT displaying the Registry entries for the `Expression` component that we will develop shortly.

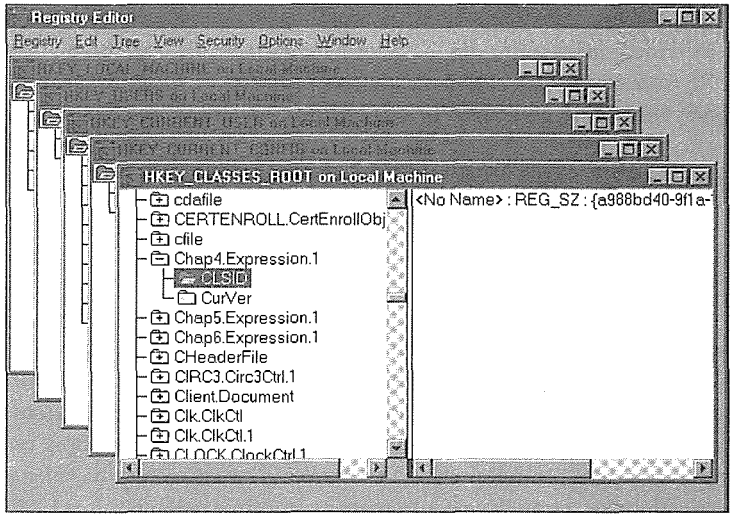


Figure 4.5 Windows NT 4.0 Registry.

There are several differences between the Windows NT and Windows 95 registries. The differences don't affect COM component information directly, but it is important to understand that the registry implementations are not identical. Here are some of the differences:

1. Windows NT has a different security model from that of Windows 95. The Windows 95 Registry does not contain security information.
2. The Windows 95 Registry does not completely replace the **CONFIG.SYS**, **WIN.INI**, and **SYSTEM.INI** files. Certain system information is contained in these files.
3. On Windows NT, the older INI API functions support a "pass-through" mechanism that will update the Registry when you're using the INI API functions. This feature is not currently supported in Windows 95.
4. The Registry hierarchy is slightly different between the two operating systems. The differences are most pronounced in Windows NT 3.5x.

The following lines show what is required to register our `Expression` component using the appropriate version of REGEDIT (see the Note). The lines are stored in a standard ASCII text file with an extension of **.REG**.

```
REGEDIT
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30} = Chap4 Expression Component
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\ProgID = Chap4.Expression.1
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\VersionIndependentProgID =
Chap4.Expression.1
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\InprocServer32 =
c:\chap4\Server\WinDebug\server.dll
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\NotInsertable

HKEY_CLASSES_ROOT\Chap4.Expression.1 = Chap4 Expression Component
HKEY_CLASSES_ROOT\Chap4.Expression.1\CLSID = {a988bd40-9f1a-11ce-8b9f-10005afb7d30}
HKEY_CLASSES_ROOT\Chap4.Expression.1\CurVer = Chap4.Expression.1
```

The following line creates the initial CLSID subkey for our component:

```
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30} = Chap4 Expression Component
```

The value `Chap4 Expression Component` is not actually necessary, but on a system with a large number of CLSIDs it's nice to quickly associate a CLSID with a particular component. The human-readable value is beneficial, and Registry browsers (such as OLEVIEW) will have something to show in the value field of your component's CLSID.

```
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\ProgID = Chap4.Expression.1
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\VersionIndependentProgID =
Chap4.Expression.1
```

```
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\InprocServer32 =
c:\chap4\Server\WinDebug\server.dll
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\NotInsertable
```

The rest of the lines set up various subkeys and values "below" the upper-level `{ a988bd40...}` CLSID key. Which subkeys are needed as well as their specific values depend on the type of component that is being registered. Some of the important CLSID Registry key entries are detailed in Table 4.2.

Table 4.2 Important Registry Key Entries

Entry	Purpose
`ProgID`	Identifies the ProgID string for the COM class. It must contain 39 characters or fewer and can contain periods.
`InprocServer32`	Contains the path and filename of the 32-bit DLL. It does not have to contain the path, but if it does not, it can be loaded only if it resides within Windows PATH. 16-bit versions do not include the "32" extension.
`LocalServer32`	Contains the path and filename of the 32-bit EXE. 16-bit versions do not include the "32" extension.
`CurVer`	The ProgID of the latest version of the component class.
`NotInsertable`	Indicates that this component will not display in the OLE standard Insert Object dialog box.

In the previous section, I showed you how a Visual Basic developer can access a specific component class by using its ProgID. For this to work, we need an entry in the Registry that maps the less specific ProgID to the more specific CLSID of the component. The following entries create a `Chap4.Expression.1` subkey off of `HKEY_CLASSES_ROOT`. Under `Chap4.Expression.1`, two additional subkeys—`CurVer` and `CLSID`—are also added. As you can imagine, the `CLSID` subkey and its value provide a cross-reference to the `HKEY_CLASSES_ROOT\CLSID` entry, where additional component information can be obtained.

```
HKEY_CLASSES_ROOT\Chap4.Expression.1 = Chap4 Expression Component
HKEY_CLASSES_ROOT\Chap4.Expression.1\CLSID = {a988bd40-9f1a-11ce-8b9f-10005afb7d30}
HKEY_CLASSES_ROOT\Chap4.Expression.1\CurVer = Chap4.Expression.1
```

We will use these entries when we register our example later in the chapter.


NOTE

The Registry editor program is **REGEDIT.EXE** in Windows 3.x and Windows 95, and **REGEDT32.EXE** in Windows NT. The method used to merge the file is also different on the three platforms. Under Windows 3.x, there is a **Merge Registration file** item on the File menu. Using Windows NT (3.51 and 4.0), you must invoke File Manager or the Explorer, locate the **.REG** file, and double-click to register the information in the Registry. Windows 95 provides a merge function within the Registry editor, similar to the Window 3.x REGEDIT program.

Class Factories

Because COM objects sometimes are located outside the component user's process space and must be accessed from various languages, a language-independent way of instantiating a component is required. In C++, the `new` operator is used to dynamically instantiate an object. COM supplies a standard interface, `IClassFactory`, that all components must provide if they are to be externally instantiated. Following is the definition of `IClassFactory`. Like all COM interfaces, it must implement `IUnknown`.

```
class IClassFactory : public IUnknown
{
   virtual HRESULT CreateInstance( LPUNKNOWN pUnk, REFIID riid, void** ppv ) = 0;
   virtual HRESULT LockServer( BOOL fLock ) = 0;
};
```

A *class factory* is a COM object whose sole purpose is to facilitate the creation of other, more useful COM objects. `CreateInstance` does what it says; it creates an instance of the specified component class and returns the requested interface on that instance. `LockServer` provides a way for a client to lock a server in memory. We will define the term *server* in detail shortly. Briefly, a server houses one or more COM objects within either a DLL or an executable (EXE). By locking a server in memory, the client ensures that it will be available when needed, even when there are no instantiated components within the server. Typically, a server is locked this way for performance reasons. Here's how the `Expression` class factory is implemented:

```
class ExpClassFactory : public IClassFactory
{
protected:
   // Reference count for the ClassFactory instance
   DWORD     m_dwRef;

public:
   ExpClassFactory();
   ~ExpClassFactory();

   // IUnknown implementation
   virtual HRESULT  QueryInterface( REFIID riid, void** ppv );
   virtual ULONG    Release();
   virtual ULONG    AddRef();

   // IClassFactory implementation
   virtual HRESULT CreateInstance( LPUNKNOWN pUnk, REFIID riid, void** ppv );
   virtual HRESULT LockServer( BOOL fLock );
};

HRESULT ExpClassFactory::CreateInstance(LPUNKNOWN pUnk, REFIID riid, void** ppv )
{
```

```
   Expression*    pExp;
   HRESULT        hr;
   // Initialize the returned pointer to
   // NULL in case there is a problem.
   *ppv = NULL;

   // Create a new instance of Expression
   pExp = new Expression;
   // Query the requested interface on the
   // new expression instance
   hr = pExp->QueryInterface( riid, ppv );

   if (FAILED( hr ))
      delete pExp;

   return hr;
}

STDMETHODIMP ExpClassFactory::LockServer( BOOLfLock )
{
  if ( fLock )
       g_dwLocks++;
  else
       g_dwLocks-;

  return NOERROR;
}
```

The user of the `Expression` component will first get a pointer to the `Expression` class factory. The user will then use the `IClassFactory` function `CreateInstance` to create an instance of the `Expression` class. If the user requests it, the `CreateInstance` function can also return the `IExpression` interface, which can be used to process algebraic expressions. The component user would do something like this:

```
main()
{
   LPCLASSFACTORY lpCF;
   Expression*  lpExp;
   HRESULT hr;
   // Get the class factory for the Expression class
   hr = CoGetClassObject( CLSID_Expression,
                          CLSCTX_INPROC,
                          NULL,
                          IID_IClassFactory,
                          &lpCF );
```

```
    // Using the class factory interface, create an instance of the
    // component and return the IExpression interface.
    lpCF->CreateInstance( NULL, IID_IExpression, &lpExp );
    // Release the class factory
    lpCF->Release();

    // Use the component to do some work
    lpExp->SetExpression( "1+2", TRUE );
    lpExp->Evaluate();

    // Release it when we're finished
    lpExp->Release();
}
```

`CoGetClassObject` is a COM API function that returns the class factory of the requested component (identified by the CLSID). `CoGetClassObject` then returns the class factory interface so that we can create an instance of the `Expression` component class. Once we've used the class factory to create an instance of `Expression`, we call `Release` through `IClassFactory`, and `Release` then deletes the class factory. This three-step process is performed often, so COM provides a helper function, `CoCreateInstance`, that encapsulates the steps. By using `CoCreateInstance`, you avoid dealing with the class factory interface; you just call `CoCreateInstance` with the specific component interface (such as `IExpression`) that you require.

Figure 4.6 illustrates what we've built so far. We have two COM components (the `Expression` component and its class factory), each with two interfaces. This is the minimum number of interfaces to encapsulate our C++ object in a binary standard wrapper.

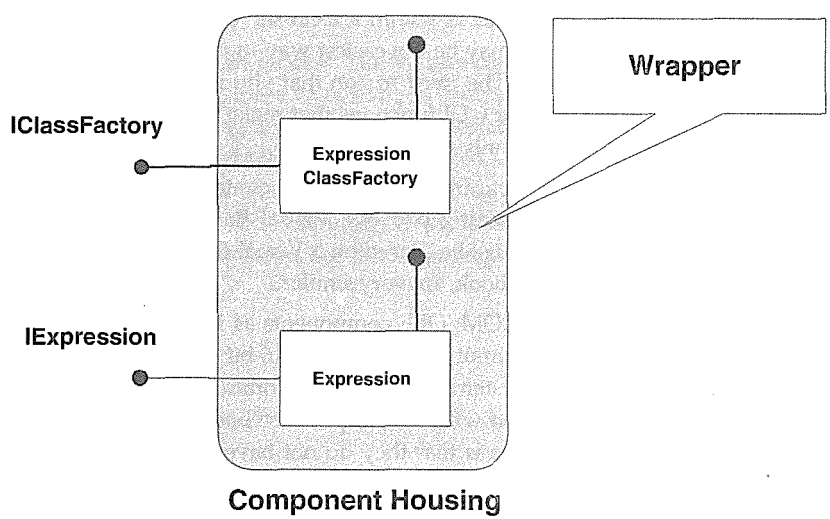


Figure 4.6 Binary standard wrapper for the `Expression` class.

Where Do Components Live?

COM objects are contained either within a Windows executable (EXE) or a Windows dynamic link library (DLL). Multiple COM objects can be maintained using either method. The user of a component doesn't need to know which technique is used to house the components. The component developer, on the other hand, implements things differently depending on which housing is used.

Most of the COM/OLE components that we will develop will be housed in Windows DLLs. ActiveX controls are almost always implemented as DLLs, because DLLs provide the best performance. OLE automation components are sometimes implemented in DLLs and sometimes in executables. Microsoft Word and Microsoft Excel are two popular Automation servers that are implemented as executables.

A Windows DLL that contains COM objects is called an *in-process*, or *inproc*, server. An executable that contains COM objects is called a *local server*. In the future, when distributed COM is available, an executable that resides on another machine on a network will be called a *remote server*. I'll use this terminology from this point on.

Local servers can be designed to function independently of COM. Many traditional applications have added OLE support by wrapping and exposing their internal methods with COM/OLE interfaces. This technique allows the application to be driven externally by another application or programming language. Visual Basic developers can incorporate these applications within their own VB applications because of Visual Basic's ability (using COM/OLE) to control and interoperate with OLE-compliant applications.

There are three reasons that, as a developer, you might implement your components within a local server. The most important one is that local servers provide easy 16-bit-to-32-bit interoperability. If you develop a 16-bit local server, its functionality can be harnessed, without change, by both 16-bit and 32-bit applications. For example, the 32-bit version of Visual Basic can interoperate with and control the 16-bit version of Microsoft Word. The reverse is also true: you can use 16-bit Visual Basic to drive 32-bit local servers. A second reason to implement your components within a local server applies if you're exposing existing functionality from a large application. This may be the easiest way to provide COM support in an existing legacy or monolithic Windows application. The third reason that you may require a local server applies if your components provide extensive visual or GUI functions that make heavy use of the Windows message queue. In this case, a local server may be the only choice.

In-process servers depend on the process in which they are contained. Their functionality cannot be harnessed without this context of a containing process. Visual Basic custom controls are similar to COM/OLE in-process servers in that they must be placed on a Visual Basic form to be useful. ActiveX controls, the topic of focus in the last half of this book, are very similar.

The primary benefit of implementing COM/OLE components as in-process servers is their small size and speed. The major problem is that they cannot be used in both 16-bit and 32-bit processes. (This issue will become less important as 32-bit operating systems become more common.) If your components require use in both Win16 and Win32 environments, you will need to produce both a 16-bit and a 32-bit implementation. Another drawback of in-process servers is that they do not have a true Windows message loop and must share the one in the containing process.

If you have the option of housing your components in either a local or an in-process server, you should probably choose the in-process type. They are far more efficient and perform significantly faster than local servers. Why? In-process servers require no marshaling, a prime factor in component performance.

In certain cases, a 16-bit local server may be required if you want to support legacy products within new 32-bit applications. Under Windows NT, there is not an easy-to-use 32-bit-to-16-bit thunking mechanism. One way to overcome this limitation is to implement your 16-bit-dependent services within a 16-bit COM-based Automation server. This method allows 32-bit applications (as well as 16-bit ones) to easily use older 16-bit code. If it's too much work or if you don't have source code to convert your 16-bit applications, this is one solution. I've done a lot of it. If you need help, read Chapter 6 and send me an email.

Marshaling

Marshaling is the process of transferring function arguments and return values across a process boundary. Marshaling requires that you copy the values to shared memory so that the other process can access it. Intrinsic types such as `short` and `long` are easy to marshal, but most others, such as pointers to structures, are a little more difficult. You can't just make a copy of a pointer, because its value (an address) has no meaning in the context of another process. You must copy the whole structure so that the other process can access it. The larger the structure, the greater the decrease in component performance.

We've observed that COM objects are housed in either a Windows executable or a Windows DLL. When the component housing is a DLL, it is called an in-process server to denote that the server component is executing within the context of the client's address space (executable). This is the most efficient method of interacting with a COM object, because no marshaling is necessary (both the client and server can freely exchange pointers).

Our examples so far have been in-process. The Vtables contain the addresses of the COM interface functions. When marshaling is required, these values are, of course, different. COM's marshaling hides this complexity from the developer by using a proxy for the interface's Vtable pointer. Figure 4.7 illustrates this mechanism. Internally, it appears to both processes that the COM object is in-process, but, in reality, COM is managing the proxy object and marshaling the function arguments using Win32's RPC capabilities. This mechanism allows COM objects to be distributed across networked machines. Marshaling is an interesting and complex topic but is beyond our scope. See the Bibliography for more details.

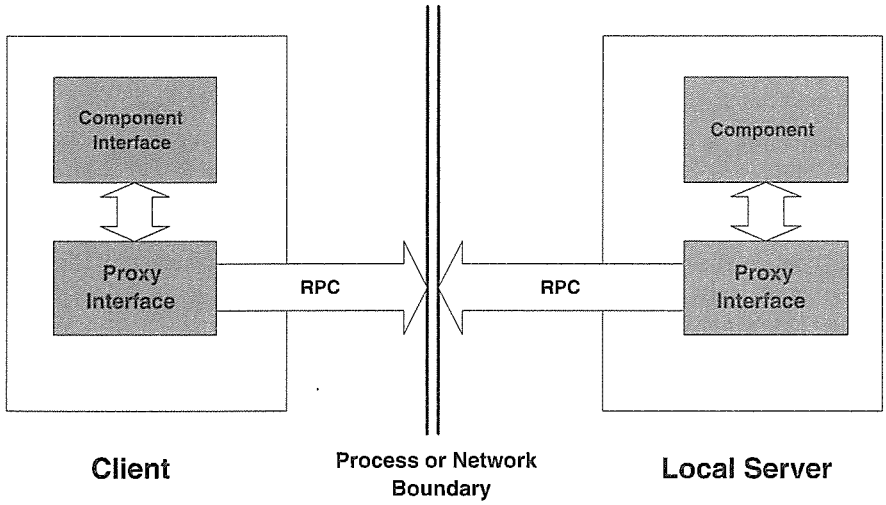


Figure 4.7 Cross-process marshaling with RPC.

Under 16-bit Windows, marshaling across process boundaries is done using LRPC, which means "lightweight" RPC. Using this method, window messages are passed back and forth between the two communicating processes. This is why you should increase you message queue buffer size to 96 to support COM/OLE under Win16.

Distributed COM

Figure 4.7 shows how RPC is used to facilitate cross-process communication with COM. Initially, this mechanism worked only between processes that were on the same machine. Today, with the release of Windows NT 4.0, the communicating COM-based processes can now reside on different machines within a network. This means that we can deploy a COM-based component on a machine in New York and access its services on a client machine in Kansas City. The client machine is oblivious to the location of the server component. It could reside locally on the client machine, in the LAN room in Kansas City, or in New York. The COM-client software doesn't care and doesn't need to know in what address space the server is executing.

Years ago, when COM was initially released, Microsoft told developers that if they would develop their components using the COM APIs, their components would eventually work across the network without requiring changes to the implementation. Microsoft has kept its promise. With the addition of one Registry key entry (`RemoteServerName`), a component that is implemented with marshaling support (such as `IDispatch`) will now work across distributed machines.

The term *distributed COM* is redundant and will eventually go away. COM was always expected to provide distributed object services, and it took time to implement them. COM now implements this feature, but DCOM will be used for a while for marketing and similar purposes.

Distributed COM support is provided by Windows NT 4.0. By the time you read this, DCOM support for Windows 95, will be available as an upgrade or service pack. This will allow transparent distributed object support on the two 32-bit Windows platforms. Additional COM parameters and functions have been added to the Win32 API to support the new DCOM features.

Custom COM Interfaces

Most of what COM (and ActiveX, as we'll see in the next chapter) provides are various interfaces that a component must implement to be binary standard and provide application functionality. We've investigated two of these interfaces in detail: `IUnknown` and `IClassFactory`. The `IExpression` interface is classified as a COM custom interface. Custom interfaces do not have COM's standard marshaling support, so we must implement the `Expression` component as an in-process server because such a server doesn't require marshaling. We could have implemented the `Expression` component within an executable, making it a local server, but writing a marshaling handler is beyond our scope.

In Chapter 6, when we use the `IDispatch` interface, you will see that it is easy to implement components within an executable housing when COM provides default marshaling. We will also look at dual interfaces, which combine a custom Vtable interface with a standard `IDispatch` interface; the client can then choose the most efficient interface to use. But how does a component client determine the type of interfaces a server provides? Good COM-based components advertise such information using standard COM interfaces.

Describing a Component

Once you've developed a COM-based component, there are certain techniques that your component should use to advertise its behavior. COM provides system-level functions and interfaces to allow a standard way of describing a component's behavior. There are two basic techniques. A component belongs to a *component category*, which provides a way to segregate components based on their capabilities. The second technique is to provide a *type library* for your component. A type library provides a granular description of your component: which interfaces it exposes, whether the interfaces are custom or standard, the method signatures, and so on.

Component Categories

The component category standard was introduced recently with the ActiveX SDK. Before its release, the only way of describing a component's gross functionality was through a few Registry keys. We discussed a few of these keys in the previous section on the Windows Registry. An ActiveX control might mark itself with the `Control` key, and an OLE document server might the `Insertable` key, indicating its ability to be inserted into an OLE document container.

The proliferation of COM-based components has made this mechanism too broad to describe all the various flavors of components. The component category standard extends the key mechanism, giving a

component a comprehensive way of describing its potential services. Component categories still use the Windows Registry, but a number of new keys have been added. The standard also allows a component to declare new, custom categories. In Chapter 7 we will discuss component categories in more detail.

Type Information

Once a component registers its specific category type, it should also provide a type library. A type library is a binary entity that can be shipped separately from the component (as a **.TLB** file) or can be tacked onto the end of the component's housing as a resource. A type library provides explicit details about the interfaces and data types used by a component.

A type information file is initially written using either Microsoft's Object Description Language (ODL) or the Interface Definition Language (IDL). The ODL or IDL script must be compiled with either MKTYPLIB (for ODL) or the MIDL (for IDL) compiler. After the type information is made available and advertised via the system Registry, the client can query for the information and determine how it should interact with the COM-based server. We will cover type information in more detail in Chapter 6.

COM Containment and Aggregation

As we discussed in Chapters 1 and 2, software and component reuse is very important. COM provides two methods for reusing component class objects: *containment* and *aggregation*. Containment and aggregation are similar to the C++ reuse techniques that we discussed in Chapter 2, but COM provides *binary* reuse as opposed to the compile-time, or *source-code–dependent*, reuse provided by C++. We won't cover these methods in detail—each one could easily fill an entire chapter—but we will encounter these terms throughout our discussion of COM, OLE, and ActiveX, so we'll cover them briefly.

COM object containment is similar to the C++ technique of class composition that we covered in Chapter 2. Containment and composition achieve reuse by using the services of a COM object or C++ class internally. The interface of the contained component is exposed only indirectly (if at all) via methods provided by the containing (or "outer") component. The interfaces of the internal (or "inner") COM component are used by the outer COM object in the implementation of its interfaces. The outer object can also, if it chooses, expose the inner object's interfaces. The lifetime of the inner object is controlled completely by the outer component just as in C++. A COM object need not do anything to support its use as an inner or contained object.

COM object aggregation is similar to COM containment except that the interface of the inner, or contained, COM object is directly exposed. The aggregate object doesn't need or use the functionality of the contained object internally; instead, it exposes the inner object's interfaces as if they were its own. The `IUnknown` interface of the outer aggregate object provides access to all the interfaces of the inner objects. This detail is what makes implementing aggregation a little complicated at times. The management of the lifetimes of the outer and inner objects must be coordinated through the `IUnknown` implementation. Successful lifetime management of the aggregate object requires that the inner objects provide support for a *controlling unknown*, which is the outer object in aggregation. When an inner COM object is created as part of

an aggregate, it is passed a pointer to the outer object's `IUnknown` implementation. The inner object then defers its `IUnknown` implementation to that of the outer object, providing a consistent approach to the management of the aggregate object's lifetime. A COM object supports aggregation if it includes support for deferring its `IUnknown` implementation to that of a controlling unknown.

The COM API

COM is primarily a model for component implementation. It provides only a few low-level functions to "get things going." Most of the true benefits of COM are provided when you build a rich layer on top of COM's services. ActiveX is such a layer. As an application-level implementation of the Component Object Model, ActiveX adds a rich set of interfaces and additional APIs beyond those provided by COM. We'll investigate features of ActiveX in Chapter 5, but first let's take a quick look at the functions provided by COM. At the end of this chapter we will go through some examples and investigate the various API functions. Table 4.3 describes COM functions that are important for our purposes. This is not a comprehensive list, but it includes those that are used most often when you're building software components.

Table 4.3 Useful COM Functions

Function ("Who Calls It")	Purpose
`CoBuildVersion` (client and server)	Gets the major and minor build numbers of the installed COM libraries. Used only by local servers. (16-bit only.)
`CoInitialize` (client and server)	Initializes the COM libraries for use by a process. Not used by in-process servers.
`CoUninitialize` (client and server)	Releases the COM libraries when its services are no longer needed. Not used by in-process servers.
`CoGetClassObject` (client)	Gets an instance of a class factory for a specific COM object.
`CoCreateGUID` (client and server)	Creates a new unique GUID.
`CoCreateInstance` (client), `CoCreateInstanceEx` (client)	Creates an instance of a specific COM object, which may be on a remote machine.
`CoRegisterClassObject` (server)	Registers the existence of a class factory for a particular COM object.
`DllCanUnloadNow` (in-process server)	Called periodically by COM to determine whether the DLL can be unloaded (when no objects are instantiated within the DLL housing). Implemented by in-process servers.
`DllGetClassObject` (server)	Entry point implemented by in-process servers so that its class factory interfaces can be obtained by client processes.

CoBuildVersion (16-bit Only)

`CoBuildVersion` is called by both client and server COM applications if the server isn't in-process. It should be called prior to `CoInitialize` to ensure that the COM libraries and DLLs are of the same major version number as when the client or server application was compiled. The minor version number may differ. If the major version numbers differ, the COM libraries cannot be used. The return value is a 32-bit integer, where the high-order 16 bits are the major build number and the low-order 16 bits are the minor build number. When you're compiling an application, the current build numbers are maintained in the COM system include file **OLE2VER.H**.

CoInitialize

`CoInitialize` is called after the COM build versions have been validated with `CoBuildVersion`. `CoInitialize` initializes the COM libraries and DLLs so that the APIs can be used. In 16-bit COM/OLE, `CoInitialize` takes one parameter, a pointer to a memory allocator. 32-bit COM and OLE do not allow a user-implemented memory allocator, so `NULL` should be passed. This approach uses the default `IMalloc` implementation.

CoUninitialize

`CoUninitialize` is called to free the use of the COM libraries and DLLs. `CoUninitialize` should be called only if `CoInitialize` has been successfully called previously.

CoRegisterClassObject

`CoRegisterClassObject` is called by a server to register its class factories as available. `CoRegisterClassObject` should be called for every class factory that a particular housing supports. This should be done as soon as possible, even before the Windows message loop is processed. `CoRegisterClassObject` is called only by local servers. An in-process server must export the `DllGetClassObject` function to allow retrieval of its component's class factories. Table 4.4 lists the `CoRegisterClassObject` parameters.

Table 4.4 `CoRegisterClassObject` Parameters

Parameter	Description
`REFCLSID rclsid`	The CLSID for the component class being registered.
`LPUNKNOWN pUnk`	Pointer to a controlling unknown.
`DWORD dwClsContext`	The requested context for the server housing. This can be one, two, three, or all of the following: `CLSCTX_INPROC_SERVER`, `CLSCTX_INPROC_HANDLER`, `CLSCTX_LOCAL_SERVER`, and `CLSCTX_REMOTE_SERVER`.

Table 4.4 `CoRegisterClassObject` Parameters (continued)

Parameter	Description
`DWORD` flags	`REGCLS` flags specify how multiple instances of the component should be created. Use one of the following: `REGCLS_SINGLEUSE`, `REGCLS_MULTIPLEUSE`, or `REGCLS_MULTI_SEPARATE`.
`LPDWORD lpdwRegister`	A value returned that must be used when deregistering the class object using the `CoRevokeClassObject` function.

The `flags` parameter of `CoRegisterClass` controls how requests for multiple instances of your component should be handled. This is important for local server implementations, and we will cover it in Chapter 6.

CoGetClassObject

`CoGetClassObject` is a low-level function that allows a client to get the `IClassFactory` interface of a specific COM object, thereby allowing the client to create an instance of a COM object. If the module that contains the component is not loaded (DLL) or is not running (EXE), `CoGetClassObject` will query the system Registry to determine the pathname for the component housing, either an in-process server (DLL) or a local server (EXE). Then the function loads the DLL and calls the entry point `DllGetClassObject` to get the requested class factory; or, if it is a local server, `CoGetClassObject` uses `CreateProcess()` to invoke a copy of the executable. Once the server is invoked and registers its class factories via `CoRegisterClassObject`, `CoGetClassObject` returns a pointer to the requested `IClassFactory`.

For Windows NT 4.0, the `COSERVERINFO` parameter is used to allow instantiation on remote servers. Prior to NT 4.0, this parameter was reserved and required a `NULL`. Table 4.5 lists the `CoGetClassObject` parameters.

Table 4.5 `CoGetClassObject` Parameters

Parameter	Description
`REFCLSID rclsid`	A reference to the CLSID for the specific component.
`DWORD dwClsContext`	The requested context for the server housing. This can be one, two, three, or all of the following: `CLSCTX_INPROC_SERVER`, `CLSCTX_INPROC_HANDLER`, `CLSCTX_LOCAL_SERVER`, and `CLSCTX_REMOTE_SERVER`.
`COSERVERINFO pServerInfo`	Pointer to `COSERVERINFO` structure.
`REFIID riid`	A reference to an IID for the specific interface to be returned from the created class object. This interface will normally be `IClassFactory` so that the client can create an instance of the required component.
`VOID** ppvObj`	A `void` pointer to return the specified interface.

CoCreateInstance and CoCreateInstanceEx

CoCreateInstance is used by a component user, or client application, to create an instance of the specified component class. It is a helper function that calls CoGetClassObject to get a class factory for the component and then uses the IClassFactory::CreateInstance method to create the component instance. You should use CoCreateInstance instead of performing the three-step process shown next unless you need to create multiple component instances or you need to explicitly lock the instance by calling IClassFactory::LockServer.

```
// What CoCreateInstance does internally
CoGetClassObject(..., &pCF );
pCF->CreateInstance(..., &pInt );
pCF->Release();
```

CoCreateInstance's parameters (Table 4.6) are similar to those required by CoGetClassObject. The primary difference is that the client using CoCreateInstance will ask for the specific interface on the component (such as IExpression) instead of an IClassFactory pointer. CoGetClassObject creates an instance of a component's class factory, whereas CoCreateInstance creates an instance of the requested component class. As we've discussed, a component and its class factory are separate COM objects.

Table 4.6 CoCreateInstance Parameters

Parameter	Description
REFCLSID rclsid	A reference to the CLSID for the specific component.
IUnknown* pUnkOuter	The controlling outer unknown (when you're using aggregation).
DWORD dwClsContext	The requested context for the server housing. This can be one, two, three, or all of the following: CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, CLSCTX_LOCAL_SERVER, and CLSCTX_REMOTE_SERVER.
REFIID riid	A reference to an IID for the specific interface to be returned from the created component object.
VOID** PPVObj	A void pointer to return the specified interface.

CoCreateInstanceEx is used to create an instance of the COM object on a remote machine. The fourth parameter specifies the remote machine. If the parameter is NULL and if there isn't a RemoveServerName Registry entry, the object is created locally. The COSERVERINFO parameter allows you to specify the name of the remote machine in UNC, DNS, or IP format. To improve efficiency, the client can query for multiple interfaces as part of the create. The MULTI_QI structure allows you to provide an array of IIDs. Upon creation, CoCreateInstanceEx returns the array filled with interfaces and the result of the queries. Both new structures are shown next. Table 4.7 lists the parameters of CoCreateInstanceEx.

```
typedef struct  _COSERVERINFO
{
   DWORD dwReserved1;
```

```
    LPWSTR pwszName;
    COAUTHINFO *pAuthInfo;
    DWORD dwReserved2;
}   COSERVERINFO;

typedef struct _MULTI_QI
{
    const IID*    pIID;
    IUnknown *    pItf;
    HRESULT       hr;
} MULTI_QI;
```

Table 4.7 `CoCreateInstanceEx` Parameters

Parameter	Description
`REFCLSID rclsid`	A reference to the CLSID for the specific component.
`IUnknown* pUnkOuter`	The controlling outer unknown (when you're using aggregation).
`DWORD dwClsContext`	The requested context for the server housing. This can be one, two, three, or all of the following: `CLSCTX_INPROC_SERVER`, `CLSCTX_INPROC_HANDLER`, `CLSCTX_LOCAL_SERVER`, and `CLSCTX_REMOTE_SERVER`.
`COSERVERINFO*` *`pServerInfo`*	Information about the remote server machine.
`ULONG`	Number of `QueryInterfaces` to perform for the `MULTI_QI` structure.
`MULTI_QI`	An array of `MULTI_QI` structures. This makes it more efficient to retrieve a series of interfaces from the create call.

DllCanUnloadNow

`DllCanUnloadNow` is implemented by in-process servers. Its purpose is to allow COM to periodically check to determine whether the DLL can be unloaded. `DllCanUnloadNow` takes no parameters and returns either `S_FALSE`, indicating to COM that the DLL cannot be unloaded, or `S_OK`, which indicates to COM that the DLL can be unloaded because there are no current references to it.

DllGetClassObject

`DllGetClassObject` is implemented by in-process servers to expose the class factories for its component objects. When a client application requests a component housed within an in-process server, COM calls the `DllGetClassObject` entry point within the DLL with the parameters shown in Table 4.8.

Table 4.8 `DllGetClassObject` Parameters

Parameter	Description
`REFCLSID rclsid`	A reference to the CLSID for the specific component.
`DWORD dwClsContext`	The requested context for the server housing. This can be one, two, or all of the following: CLSCTX_INPROC_SERVER, CLSCTX_INPROC_HANDLER, and CLSCTX_LOCAL_SERVER.
`LPVOID pvReserved`	Reserved. Must be `NULL`.
`REFIID riid`	A reference to an IID for the specific interface to be returned from the created COM object. This will normally be `IClassFactory` so that the client can create an instance of the requested component.
`VOID** ppvObj`	A `void` pointer to return the specified interface.

Client/Server Flow

Now that we've reviewed the major COM API functions, let's follow the flow of a simple client/server interaction. Table 4.9 describes the flow between a client executable and an in-process server.

Table 4.9 Client/Server Flow for In-Process Server

Client	COM/OS	In-Process Server
Initialize COM (`CoInitialize()`).		
Create an instance of a component's class factory (`CoGetClassObject()`).		
	Check Registry for path and filename of the in-process server DLL. If the DLL is not loaded for the client process, load it.	
		`DllGetClassObject()` is called with the CLSID of the requested component. Create class factory and return an `IClassFactory` pointer.
Use `IClassFactory` to create an instance of the component. Request the specific interface we need, such as `IExpression` (`(pCF->CreateInstance()`.)		

Table 4.9 Client/Server Flow for In-Process Server (continued)

Client	COM/OS	In-Process Server
		The component's class factory creates an instance of the component object and returns an interface pointer to `IExpression`.
Release the class factory instance (`pCF->Release()`).		
		`Release()` for the class factory causes the reference count to reach zero, so the class factory instance is deleted.
The interface on the component object is used to perform some tasks. When the tasks are finished, `Release()` is called.		
		`Release()` decrements the internal reference count to reach zero, and the component object destroys itself. The component object count also reaches zero because there are no object instances in the DLL.
At some point (such as at idle time) `CoFreeUnusedLibraries()` is called.		
	COM/OS calls `DllCanUnloadNow()`.	
		`DllCanUnloadNow()` returns `TRUE`.
	`DllCanUnloadNow()` returns `TRUE`, and COM unloads the DLL.	

In the next scenario, (Table 4.10), the server application is implemented as a local server. There are many possible scenarios with local servers. The EXE may or may not be running when the client asks for a component's services, and the REGCLS flags also play a role when multiple instances of a component are required. The scenario in Table 4.10 uses a single instance of the executable, and the executable isn't running when the client requests a component class. It should give you a good idea of what occurs when you're using a local server.

Table 4.10 Client/Server Flow for Local Server (Server Not Running)

Client	COM/OS	Server
Initialize COM (`CoInitialize()`).		
Create an instance of a component's class factory (`CoGetClassObject()`).		
	COM checks to see if the component has been registered within its internal tables. If it is, then the EXE is running. If it isn't, then query the Registry for the pathname of the local server EXE. Start up the EXE and wait for it to register its class factories.	
		COM starts up the EXE.
		Initialize COM (`CoInitialize()`).
		Register all the housed component's class factories as available using `CoRegisterClassObject`.
		A class factory instance is created for the requested component and is returned.
`CoGetClassObject` returns with a pointer to the component's class factory.		
Use `IClassFactory` to create an instance of the component. Request the specific interface we need, such as `IExpression` `pCF->CreateInstance`.		
		The component's class factory creates an instance of the component object and returns an interface pointer.
Release the class factory instance (`pCF->Release()`).		
		`Release()` for the class factory causes the reference count to reach zero, so the class factory instance is deleted.
The interface on the component object is used to perform some tasks. When the tasks are finished, `Release()` is called.		

Table 4.10 Client/Server Flow for Local Server (Server Not Running) (continued)

Client	COM/OS	Server
		`Release()` decrements the internal reference count to reach zero, and the component object destroys itself. The component object count also reaches zero, because there are no object instances in the EXE. The EXE unregisters its class factories by calling `CoRevokeClassObject` and then terminates.

COM C++ Macros, BSTRs, and So On

Until this point, I haven't been using the standard COM/OLE macros when declaring interfaces, because for instructional purposes they get in the way of understanding what's going on. COM, OLE, and MFC use C/C++ macros extensively to hide the implementation details of the various platforms. We're getting ready to write real code, so it's time I explained the macros we'll be using. There are four macros for COM/OLE interface declarations and definitions: `STDMETHOD`, `STDMETHOD_`, `STDMETHODIMP`, and `STDMETHODIMP_`. In an earlier example of `IExpression`, we declared it like this:

```
// public interface definition of our Expression component
// An abstract class
class IExpression
{
public:
   virtual CString GetExpression() = 0;
   virtual void SetExpression( CString str, BOOL bInfix ) = 0;
   virtual BOOL Validate() = 0;
   virtual long Evaluate() = 0;
};
```

Using the COM's macros and intrinsic data types, it would be declared as follows:

```
class IExpression
{
public:
   STDMETHOD_(BSTR,GetExpression()) PURE;
   STDMETHOD_(void, SetExpression(BSTR, BOOL)) PURE;
   STDMETHOD_(BOOL, Validate())      PURE;
   STDMETHOD_(long, Evaluate())      PURE;
};
```

The expansion of `STDMETHOD_` depends on the target platform and whether you're using C or C++. The expansion for Win32 using C++ is as follows:

```
// OBJBASE.H

#define STDMETHODCALLTYPE       __stdcall
...
#define STDMETHOD(method)       virtual HRESULT STDMETHODCALLTYPE method
#define STDMETHOD_(type,method) virtual type STDMETHODCALLTYPE method
#define PURE                    = 0

#define STDMETHODIMP            HRESULT STDMETHODCALLTYPE
#define STDMETHODIMP_(type)     type STDMETHODCALLTYPE
```

As you can see, our earlier example is similar to the expanded macro version, with the exception of the additional return type `__stdcall`. This Microsoft-specific calling convention is used by the Win32 API functions. It specifies that the callee will clean up the stack after the call. It isn't important to our understanding, but it's interesting reading if you have nothing else to do. As you can see, `PURE` equates to `= 0` and is just another way of making a function pure virtual. Most COM interfaces return `HRESULT`, so the special macro `STDMETHOD` defaults the return type to `HRESULT`. We didn't use it in the `IExpression` interface, but here is a quick example from `IClassFactory`:

```
STDMETHOD( LockServer(BOOL fLock) )  PURE;
// Expands to this
virtual HRESULT _stdcall LockServer(BOOL fLock) = 0;
```

`STDMETHOD` is also used in the declaration of interface methods within the implementing class. The only difference is that you don't need the `PURE` qualifier. The `STDMETHODIMP` macros are used when you implement the interface function. Here are the declarations of our `IExpression` methods within the `Expression` class:

```
class Expression : public IExpression
{
...
   // IExpression
   STDMETHOD_(BSTR, GetExpression());
   STDMETHOD_(void, SetExpression( BSTR, BOOL ));

   STDMETHOD_(BOOL, Validate());
   STDMETHOD_(long, Evaluate());
...
};
```

When we implement the functions in our **.CPP** file, we use the `STDMETHODIMP` macros as follows:

```
STDMETHODIMP_(BSTR) Expression::GetExpression()
{
   return ::SysAllocString( m_strExpression );
```

```
}

STDMETHODIMP_(void) Expression::SetExpression( BSTR bstrExp, BOOL bInfix )
{
   m_strExpression = bstrExp;
   m_bInfix = bInfix;
}

STDMETHODIMP_(BOOL) Expression::Validate()
{
...
}

STDMETHODIMP_(long) Expression::Evaluate()
{
...
}
```

BSTR

If you look closely, you'll see another difference in the preceding declarations. In the `GetExpression` function of `IExpression`, we return something declared as a `BSTR`. A `BSTR` is a binary string and is one of the standard data types used within Visual Basic. It is a string that stores its length in the first two or four bytes (depending on the platform) and the string data thereafter. `BSTR` is a standard data type that is used extensively in Automation. We're using it here to return a copy of the expression as well as to set the expression in `SetExpression`. The client application may not understand the structure of MFC's `CString` class, nor should it have to. COM and ActiveX define many standard data types that are provided by COM's standard marshaling support so that clients and servers have a standard way of passing data. We'll talk more about these types in Chapters 6 and 7. Custom data types are also supported within COM and are typically used with custom interfaces, but they're beyond the scope of this book.

One important aspect of COM interoperation between client and server processes is that the client is responsible for deallocating memory allocated by the server. Only the client knows when it will be finished with a piece of data returned through an interface method call, so the client is responsible for the deallocation. COM provides functions to make this fairly easy. MFC also makes it easy as by providing `BSTR` support within its `CString` class.

HRESULT and SCODE

Most COM interface methods and API functions return an `HRESULT`. An `HRESULT` in Win32 is defined as a `DWORD` (32 bits) that contains information about the result of a function call. The high-order bit indicates the success or failure of the function, the next 15 bits indicate the facility and provide a way to group related return codes, and the lowest 16 bits provide specific information on what occurred. (If you are an old DEC

VMS programmer, this is how the system error codes work there as well. David Cutler designed them both.) To check the gross success or failure of a function, you need only check the high-order bit of the return, and COM provides some macros to make this easy. The SUCCEEDED macro evaluates to TRUE if the function call was successful, and the FAILED macro evaluates to TRUE if the function failed. These macros aren't specific to COM and ActiveX but are used throughout the Win32 environment and are defined in **WINERROR.H**. Return values in Win32 are prefixed with S_ when they indicate success, and E_ to indicate failure.

```
// From WINERROR.H
...
// Generic test for success on any status value (nonnegative numbers
// indicate success).
//
#define SUCCEEDED(Status) ((HRESULT)(Status) >= 0)
//
// and the inverse
//
#define FAILED(Status) ((HRESULT)(Status)<0)
...
//
// Create an HRESULT value from component pieces
//
#define MAKE_HRESULT(sev,fac,code) \
    ((HRESULT) (((unsigned long)(sev)<<31) | ((unsigned long)(fac)<<16) | ((unsigned
long)(code))) )
#define MAKE_SCODE(sev,fac,code) \
    ((SCODE) (((unsigned long)(sev)<<31) | ((unsigned long)(fac)<<16) | ((unsigned long)(code))) )
```

An SCODE is a special return value used for Win16 COM/OLE functions, but under Win32 an SCODE is really an HRESULT. You should use HRESULT instead of SCODE when developing 32-bit applications.


NOTE

Win16 COM/OLE used the function's GetScode to map an HRESULT to an SCODE and ResultFromScode to map an SCODE to an HRESULT. Our examples will use these functions when providing Win16 support. For example, to return E_NOINTERFACE from QueryInterface, which returns an HRESULT, you would do this: return ResultFromScode(E_NOINTERFACE).

An Example

Now that we've discussed the details of COM, we need some good examples to tie all the concepts together. In the next few sections we will develop two programs. First, we will implement the `Expression` component as an in-process server. We won't implement it as a local server, because the custom interface, `IExpression`, would require special marshaling code. As we'll see in Chapter 6, a standard ActiveX interface, `IDispatch`, provides additional functionality, including marshaling, for the `Expression` class.

Then we will build a simple client application that will use the `Expression` component as an in-process server. We will use MFC but without MFC's COM/OLE support; instead, we will use native COM calls. We've delved into the details of COM so that we have a solid footing on which to continue. Later, when using MFC, we won't probe into the depths of COM again because the details have been nicely abstracted by the MFC libraries.

The Expression Class as a COM Component

Throughout this chapter we've been using the `Expression` class and its public interface for the example code. In this section, we'll develop a COM component that is contained within a DLL. This in-process server will allow the `Expression` object to be used by any COM-compliant client process. In the next section we will develop a C++ client application that uses the `Expression` component.

Start Visual C++, choose **File/New**, and select **New Project Workspace**. From the New dialog box, select **MFC AppWizard (dll)** as the project type and name the project **SERVER**. Then click the **Create** button to continue. Figure 4.8 shows the New Project Workspace dialog box.

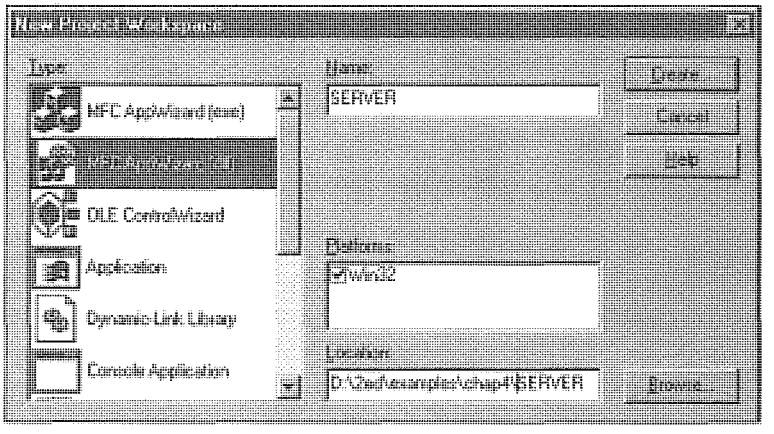


Figure 4.8 The New Project Workspace dialog box.

In the MFC AppWizard Step 1 of 1 dialog box, select the **Regular DLL with MFC statically linked** option and take the defaults on the others. Click **Finish** to create the project files. Make sure the options match those shown in Figure 4.9.

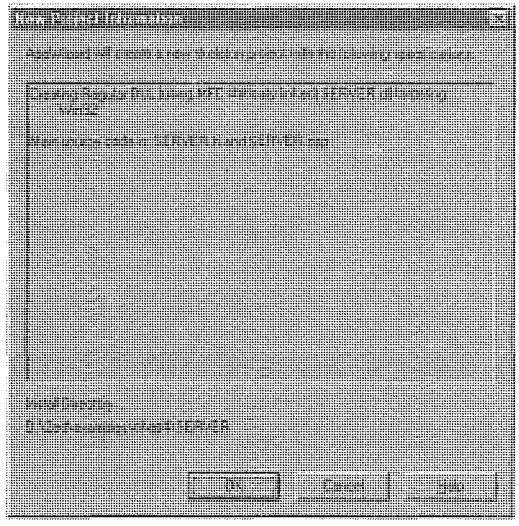


Figure 4.9 Server New Project Information dialog box.

These steps create a template DLL application that uses the MFC libraries. There isn't much code—basically just the include files needed for MFC support. The initial AppWizard-produced code from **SERVER.H** is shown next:

```
// Server.h : main header file for the SERVER DLL
//

#ifndef __AFXWIN_H__
   #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"

////////////////////////////////////////////////////////////////////////////
// CServerApp
// See server.cpp for the implementation of this class
//

class CServerApp : public CWinApp
{
public:
```

```
    CServerApp();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CServerApp)
    //}}AFX_VIRTUAL

    //{{AFX_MSG(CServerApp)
        // 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()
};
```

This code declares our CWinApp-derived application class. As we discussed in Chapter 3, all MFC applications derive an application class from CWinApp that is globally instantiated. The code here declares the application constructor CServerApp. Let's look at **SERVER.CPP**:

```
// Server.cpp : Defines the initialization routines for the DLL.

//

#include "stdafx.h"
#include "Server.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CServerApp

BEGIN_MESSAGE_MAP(CServerApp, CWinApp)
    //{{AFX_MSG_MAP(CServerApp)
        // 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()

/////////////////////////////////////////////////////////////////////////////
// CServerApp construction

CServerApp::CServerApp()
{
```

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

/////////////////////////////////////////////////////////////////////////////
// The one and only CServerApp object

CServerApp theApp;
```

There isn't much here either. Under Win32, when DLLs are initially loaded, the entry point `DllMain` is called with the `hInstance` parameter. The MFC libraries encapsulate this entry point and provide the function `InitInstance` to allow the application to perform any initializations. We don't need to initialize anything on startup, so we'll use the AppWizard code as provided. AppWizard has generated a basic DLL in which to implement our COM-specific code.

In-process servers must contain the function `DllGetClassObject`, which provides a standard entry point for clients that instantiate a class factory object for the specified component class (CLSID) and returns its `IClassFactory` interface. If an in-process server supports multiple COM objects, the `DllGetClassObject` function will contain code to identify the specific CLSID and will create the corresponding class factory. Our DLL contains only the `Expression` component and its supporting class factory, so it's pretty simple. `DllGetClassObject` provides the class factories for all the components in our housing, so we should add the following code to the end of **SERVER.CPP**.

```
STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid, void** ppv )
{
   HRESULT            hr;
   ExpClassFactory    *pCF;

   pCF = NULL;

   // Make sure the CLSID is for our Expression component
   if ( rclsid != CLSID_Expression )
      return( E_FAIL );

   pCF = new ExpClassFactory;

   if ( pCF == NULL )
      return( E_OUTOFMEMORY );

   hr = pCF->QueryInterface( riid, ppv );

   // Check for failure of QueryInterface
   if ( FAILED( hr ) )
   {
      delete pCF;
      pCF = NULL;
   }

   return hr;
}
```

The function checks for CLSID_Expression. If the caller is requesting an unrecognized component class, we return E_FAIL. If the correct CLSID is provided, we instantiate an Expression class factory and query for the requested interface, typically IClassFactory. We then return the HRESULT of QueryInterface.

The second COM function that must be implemented in an in-process server is DllCanUnloadNow. This function provides a way for COM to periodically check to determine whether the DLL can be unloaded. Here's our implementation of DllCanUnloadNow. It also belongs in **SERVER.CPP**.

```
STDAPI DllCanUnloadNow(void)
{
    if ( g_dwObjs || g_dwLocks )
        return( S_FALSE );
    else
        return( S_OK );
}
```

We're responsible for keeping track of how many COM objects are in use at any given time. The global variables g_dwObjs and g_dwLocks keep track of the instantiated Expression objects and the current number of LockServer calls. LockServer, a member of the IClassFactory interface, provides a way for a client to lock a DLL in memory even if it is not currently being used. We declare these variables globally outside any function. (They must hang around even when there are no instantiated objects.)

```
// Server.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "Server.h"

#include <initguid.h>
#include "expsvr.h"

DWORD       g_dwObjs = 0;
DWORD       g_dwLocks = 0;

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

These global variables are for a particular instance of the DLL. Under Win32, every process that loads this DLL gets a fresh copy with the global variables initialized to zero. This isn't exactly true under Win16, because DLLs exist at a system level and aren't owned by a particular process, but this method works under Win16 as well. We've also added the include for the **EXPSVR.H** file that contains the definitions and IID for our custom interface, IExpression. I've highlighted the changes from the Chapter 3 implementation of the Expression class (in the file **EXPRESS.H**). It would be beneficial to copy the **EXPRESS.H** from the Chapter 3 project to your Chapter 4 SERVER project directory and rename it **EXPSVR.H**.

```
//
// ExpSvr.h
//

// access to the global variables in SERVER.CPP
extern DWORD g_dwObjs;
extern DWORD g_dwLocks;

DEFINE_GUID( CLSID_Expression,
             0xA988BD40,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);
DEFINE_GUID( IID_IExpression,
             0xA988BD41,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);

//
// A Stack class that supports CStrings
//
class CStringStack : public CObject
{
protected:
   DECLARE_DYNCREATE( CStringStack )
   CStringList m_StringList;

public:

   CStringStack();
   CStringStack( CStringStack& stack );
   ~CStringStack();

   CStringStack& operator=( const CStringStack& lhs );

   virtual void Dump( CDumpContext& );

   void     Push( CString );
   BOOL     Peek( CString& );
   BOOL     Pop( CString& );
   BOOL     IsEmpty();
   void     Clear();
};

//
// Tokenizes an algebraic expression string
//
class Tokenizer : public CObject
{
protected:
```

```
   DECLARE_DYNCREATE( Tokenizer )
   char         m_szBuffer[256];
   CStringList  m_TokenList;

public:
   Tokenizer();
   Tokenizer( const CString& strString );
   ~Tokenizer();

public:
   void     SetString( const CString& str );
   short    Tokenize();
   BOOL     GetToken( CString& str );
   BOOL     PeekToken( CString& str );
   void     ClearTokens();

};

class IExpression : public IUnknown
{
public:
   STDMETHOD_(BSTR,GetExpression()) PURE;
   STDMETHOD_(void, SetExpression(BSTR, BOOL)) PURE;
   STDMETHOD_(BOOL, Validate())      PURE;
   STDMETHOD_(long, Evaluate())      PURE;
};

class Expression : public IExpression
{
protected:
  enum TokenType
   {
      BogusToken,
      OperatorToken,
      OpenParenToken,
      CloseParenToken,
      NumberToken
   };

protected:
   CString        m_strExpression;
   BOOL           m_bInfix;
```

```
    // Reference count
    DWORD           m_dwRef;

public:
    // Constructors
    Expression();
    Expression( CString str, BOOL bInfix );
    // Destructor
    ~Expression();
    // Copy constructor
    Expression( Expression& x );
    // assignment operator
    Expression& operator=( Expression& rhs );

protected:
    BOOL            IsNumber( const CString& strToken );
    TokenType       GetTokenType( const CString& strToken );
    int             Precedence( const CString& strToken );
    BOOL            InfixToPostfix();

public:
    // IUnknown
    STDMETHOD(QueryInterface( REFIID, void** ));
    STDMETHOD_(ULONG, AddRef());
    STDMETHOD_(ULONG, Release());
    // IExpression
    STDMETHOD_(BSTR, GetExpression());
    STDMETHOD_(void, SetExpression( BSTR, BOOL ));
    STDMETHOD_(BOOL, Validate());
    STDMETHOD_(long, Evaluate());
};

class ExpClassFactory : public IClassFactory
{
protected:
    DWORD           m_dwRef;

public:
    ExpClassFactory();
    ~ExpClassFactory();

    // IUnknown
    STDMETHOD( QueryInterface(REFIID, void** ));
```

```
    STDMETHOD_(ULONG, AddRef());
    STDMETHOD_(ULONG, Release());

    // IClassFactory
    STDMETHOD( CreateInstance(LPUNKNOWN, REFIID, void**));
    STDMETHOD( LockServer(BOOL));
};
```

The new items expose our C++ component using COM. We now need to add the implementation of the preceding functions to **EXPSVR.CPP**. Only a few things have changed, so I include only the pertinent code. Items not shown are taken directly from the Chapter 3 implementation in **EXPRESS.CPP**. The original implementations of `CStringStack` and `Tokenizer` and the protected member functions `InfixToPostfix`, `Precedence`, `IsNumber`, and `GetTokenType` have not changed.

```
//
// ExpSvr.cpp
//

#include "stdafx.h"

// To support Unicode conversion in 4.0
#if ( _MFC_VER >= 0x400 )
    #include <afxpriv.h>
#endif

#include <stdio.h>
...
```

We first need to include the MFC **AFXPRIV.H** header file. With the release of MFC 4.0, we now must manage all Unicode conversions ourselves. Fortunately, **AFXPRIV.H** provides a number of macros that make the process much simpler.

Remember, all Win32 COM/OLE functions are inherently Unicode. Our project, however, is built using ANSI (or multibyte) strings. As you will see, we will need to use Unicode when passing strings to and from the COM functions. Also, the `BSTR` type expects Unicode strings. Here are the rest of the changes to **EXPSVR.CPP**:

```
ExpClassFactory::ExpClassFactory()

{
    m_dwRef = 0;
}

ExpClassFactory::~ExpClassFactory()
{
}
```

```
STDMETHODIMP ExpClassFactory::QueryInterface( REFIID riid, void** ppv )
{
   *ppv = NULL;

   if ( riid == IID_IUnknown || riid == IID_IClassFactory )
      *ppv = this;

   if ( *ppv )
   {
      ( (LPUNKNOWN)*ppv )->AddRef();
      return NOERROR;
   }

   return ResultFromScode(E_NOINTERFACE);
}

STDMETHODIMP_(ULONG) ExpClassFactory::AddRef()
{
    return ++m_dwRef;
}

STDMETHODIMP_(ULONG) ExpClassFactory::Release()
{
   if ( --m_dwRef )
      return m_dwRef;
   else
      delete this;

    return 0;
}

STDMETHODIMP ExpClassFactory::CreateInstance
( LPUNKNOWN pUnkOuter, REFIID riid, void** ppvObj )
{
    Expression* pExpression;
    HRESULT     hr;

    *ppvObj = NULL;

    pExpression = new Expression;

    if ( pExpression == NULL )
        return E_OUTOFMEMORY;

    hr = pExpression->QueryInterface( riid, ppvObj );
```

```
    if ( FAILED( hr ) )
        delete pExpression;
    else
        g_dwObjs++;  // Increment the global object count

    return hr;
}

STDMETHODIMP ExpClassFactory::LockServer( BOOL fLock )

{
    if ( fLock )
        g_dwLocks++;
    else
        g_dwLocks--;

    return NOERROR;
}

// Constructors
Expression::Expression()
{
   m_dwRef = 0;
   m_bInfix = TRUE;
}

Expression::Expression( CString strExpression, BOOL bInfix )
{
    m_dwRef = 0;
    m_strExpression = strExpression;
    m_bInfix = bInfix;
}

Expression::~Expression(void)
{
    g_dwObjs--;
}

STDMETHODIMP_(void) Expression::SetExpression( BSTR bstrExp, BOOL bInfix )
{
   m_strExpression = bstrExp;
}

STDMETHODIMP_(BOOL) Expression::Validate()
{
```

```
...
// Implementation the same, only the declaration is different
...
}

STDMETHODIMP_(long) Expression::Evaluate()
{
...
// Implementation the same, only the declaration is different
...
}

STDMETHODIMP Expression::QueryInterface( REFIID riid, void** ppv )
{
    *ppv = NULL;

    if ( riid == IID_IUnknown || riid == IID_IExpression )
        *ppv = this;

    if ( *ppv )
    {
        ( (LPUNKNOWN)*ppv )->AddRef();
        return( S_OK );
    }
    return (E_NOINTERFACE);
}

STDMETHODIMP_(ULONG) Expression::AddRef()
{

    return ++m_dwRef;
}

STDMETHODIMP_(ULONG) Expression::Release()
{
    if ( -m_dwRef )
        return m_dwRef;

    delete this;
    return 0;
}
```

I saved `GetExpression` for last, because we encounter Unicode again. **AFXPRIV.H** includes several macros that make dealing with Unicode easier. The macros are described in detail in *MFC Tech Note 59*, but we're using just one of them here. `T2OLE` converts an ANSI string to Unicode. The required `USES_CONVER-`

SION line declares some local storage for the conversion macros. Once we get a Unicode string, we pass it to the COM SysAllocString function.

Remember, too, that we will allocate storage for the returned string, and the client is responsible for eventually releasing the memory. This is an important rule to understand when you're working with COM.

```
STDMETHODIMP_(BSTR) Expression::GetExpression()
{
#if ( _MFC_VER >= 0x400 )
   USES_CONVERSION
   LPCOLESTR lpOleStr = T2COLE( m_strExpression );
   return ::SysAllocString( lpOleStr );
#else
   return ::SysAllocString( m_strExpression );
#endif
}
```

We need to export the COM API functions so that they can be called outside the DLL. Visual C++ creates a default **.DEF** file as part of the project, but you must export the two COM-specific functions that we added to **SERVER.CPP**. Open **SERVER.DEF** and add the highlighted code:

```
; Server.def : Declares the module parameters for the DLL.

LIBRARY      SERVER
DESCRIPTION  'SERVER Windows Dynamic Link Library'

EXPORTS
    ; Explicit exports can go here
                DllGetClassObject       @2
                DllCanUnloadNow         @3
```

We're almost finished. Next, we add the new **EXPSVR.CPP** file to the project. Click **Insert/Files into Project** and add **EXPSVR.CPP**. After this, go ahead and compile and link the SERVER DLL.

Register the Component

Before we can use the component, it must be registered in the Windows Registry. Later, we'll do this programmatically, but for now we'll need to use **REGEDIT.EXE** (Windows 95), File Manager (NT 3.51), or Explorer (Windows 95 or NT 4.0), depending on your operating system. Included on the accompanying CD-ROM is a **.REG** file (**WIN32.REG**) that you can use to enter the keys into the Registry. Following is the text of the **WIN32.REG** file for the Chap4 Expression Component. You will have to edit **WIN32.REG** and modify the highlighted line with the path of your project.

```
REGEDIT
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30} = Chap4 Expression Component
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\ProgID = Chap4.Expression.1
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\VersionIndependentProgID =
Chap4.Expression.1
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\InprocServer32 =
c:\chap4\Server\Debug\server.dll
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}\NotInsertable

HKEY_CLASSES_ROOT\Chap4.Expression.1 = Chap4 Expression Component
HKEY_CLASSES_ROOT\Chap4.Expression.1\CLSID = {a988bd40-9f1a-11ce-8b9f-10005afb7d30}
HKEY_CLASSES_ROOT\Chap4.Expression.1\CurVer = Chap4.Expression.1
```

After we've registered the component, we can quickly test the server with a nice utility: **OLEVIEW.EXE**.

A Quick Test of the In-Process Server

Included with Visual C++ is a program called **OLEVIEW.EXE** (It may instead be named **OLE2VW32.EXE**.) OLEVIEW is also included with the ActiveX SDK. It is good for initially testing COM/OLE components. We will use one of its features here to ensure that our new **SERVER.DLL** can be loaded. We will also verify that it has been successfully entered into the Registry.

Start **OLEVIEW.EXE** using your favorite method (mine is **start oleview** from a command prompt) and locate the entry for our server `Chap4 Expression Component`. If you can't find it, make sure that you are displaying noninsertable objects. You will also need to set **OLEVIEW.EXE** to **Expert Mode** to locate the component. It will be under the All Objects tree. Once you've located our server, click on it once to highlight it. The Registry information should display as shown in Figure 4.10.

Now, to quickly test to see whether we've done everything right, double-click or expand the entry. This will attempt to perform a `CoGetClassObject`, which calls our `DllGetClassObject` function to instantiate a class factory. If this succeeds, you'll see the IUKnown interface displayed under the entry as in Figure 4.10. If it fails, you'll see something like this "CO_E_SERVER_EXEC_FAILURE" on the status bar. This means either that the path to the DLL is wrong or the DLL isn't there. Another possibility is that you forgot to export the `DllGetClassObject` function from your DLL.

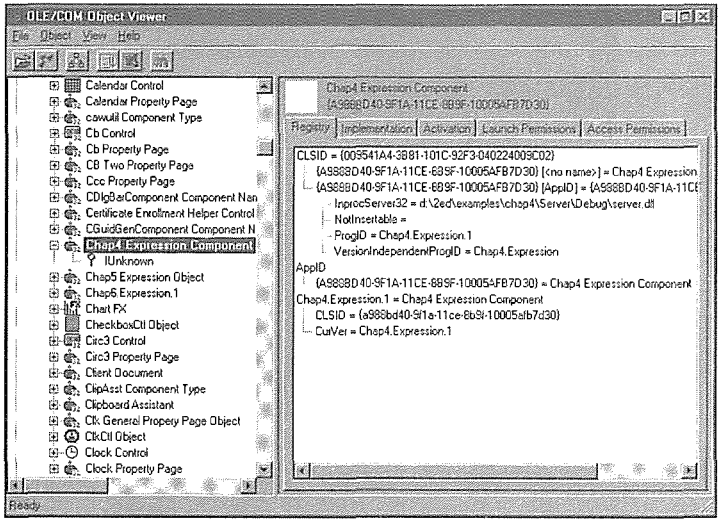


Figure 4.10 Successful call to `CoGetClassObject`.

A COM Client Application

Next we'll build an application that uses the `Expression` in-process server that we developed in the previous section. In Chapter 3, we developed an MFC application that used the original `Expression` C++ class from Chapter 2. Our client application is going to behave, on the surface, just as it did in Chapter 3. The application user types an expression and then validates or evaluates the expression using the command buttons.

We'll provide the same functionality, but instead of incorporating the C++ class into the application, we will access the `Expression` COM object for the needed functionality.

Start Visual C++, go into AppWizard, and create a new project just as we did at the end of Chapter 3. Call this new project **CLIENT**. Make it an SDI application with no OLE support and be sure to derive the `View` class from `CFormView` so that we can easily place our controls. When you get to the last AppWizard screen, check to make sure it matches the screen shown in Figure 4.11.

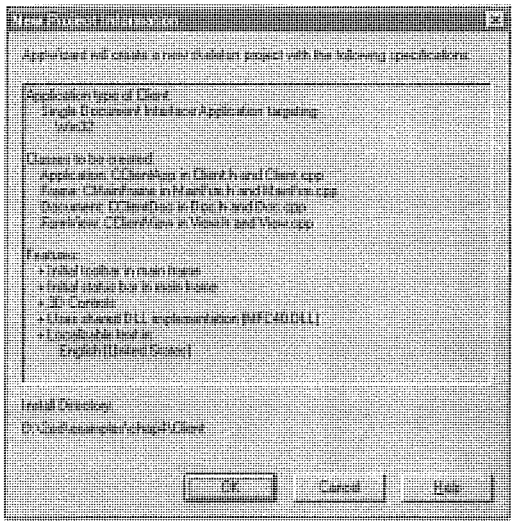


Figure 4.11 Client New Project Information dialog box.

Just as in Chapter 3, edit the **.RC** file and add an entry field and two command buttons to the `IDD_CLIENT_FORM` dialog. Name the command buttons `IDC_VALIDATE` and `IDC_EVALUATE` and the entry field `IDC_EXPRESSION`. Close the resource editor and save the changes. Now start ClassWizard, go to the `CClientView` class, and tie the `IDC_VALIDATE` and `IDC_EVALUATE` `BN_CLICKED` events to a function. Take the default `OnValidate` and `OnEvaluate`. All this is the same as in Chapter 3. Now compile and link the application and make sure it looks similar to the screen shown in Figure 4.12.

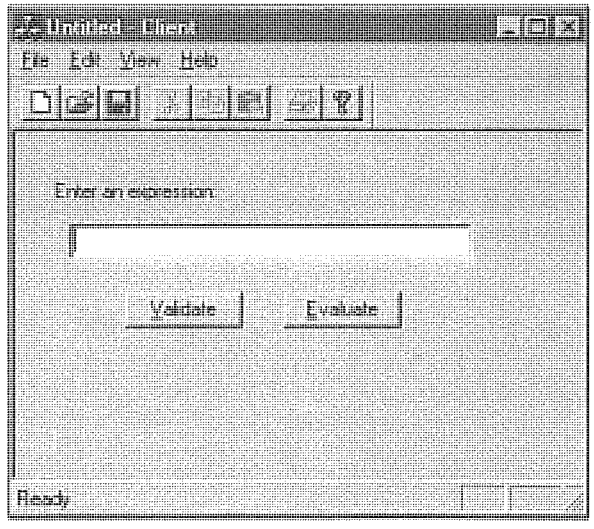


Figure 4.12 Client application.

Before we can access our COM component, we must set up the COM/OLE environment by initializing the various DLLs. COM provides the `CoInitialize` API functions to perform this task. Edit the `InitInstance` member of **CLIENT.CPP**. `InitInstance` is called only once during the startup of the application. This file provides a perfect place to initialize COM.

```
// client.cpp
...
BOOL CClientApp::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.

   // Initialize COM
   HRESULT hr = ::CoInitialize( NULL );
   if ( FAILED( hr ) )
   {
      AfxMessageBox( "Unable to Initialize COM, exiting" );
      return FALSE;
   }

   Enable3dControls();
...
}
```

`CoBuildVersion` is required only for 16-bit applications. Under Win32, this call is not required and is documented as obsolete. If you're writing 16-bit code, you should check the version of the COM/OLE DLLs on the executing system. Here's how:

```
// Check the version of the COM DLLs
DWORD dwBV = ::CoBuildVersion();
if ( HIWORD( dwBV ) != rmm || LOWORD( dwBV ) < rup )
{
   AfxMessageBox( "COM Version is too old to continue" );
   return FALSE;
}
```

The symbols `rmm` and `rup` refer to the COM/OLE build number and are defined in the **OLE2VER.H** include file.

```
// client.cpp
...
#include "view.h"
#include <ole2ver.h>
#ifdef _DEBUG
```

We also need to include the **AFXOLE.H** file in **STDAFX.H** so that we have access to the various COM/OLE API functions.

```
// stdafx.h
...
#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions

#include <afxole.h>
```

We should terminate and release the COM libraries when our application terminates. MFC provides `ExitInstance`, an overridable member of `CWinApp` that is called just before application termination. Using ClassWizard, select `CClientApp` from the dropdown listbox. Select `CClientApp` in the Objects IDs listbox, and all the overridable members of `CWinApp` are shown. Select `ExitInstance` from the Messages listbox and click the **Add Function** button. Then click **Edit Code** and add the following:

```
int CClientApp::ExitInstance()
{
    // Shutdown COM
    ::CoUninitialize();

   return CWinApp::ExitInstance();
}
```

Now we have added basic COM support to our application. Go ahead and compile, link, and run the application in debug to get a sense of what occurs. Not much, right? COM is easy.

Now let's add the functionality from the `Expression` in-process server. Because the `IExpression` interface is a custom interface not provided by COM, we need to include the interface definition. Open a new file in Visual C++ and call it **IEXP.H**. Either type in or copy the following code from **EXPSRV.H**.

```
//
// iexp.h - contains the IID and interface definition for the
//          IExpression custom interface.

#ifndef IEXP_H_
#define IEXP_H_

#ifdef INITGUID
#include <initguid.h>
#endif

DEFINE_GUID(IID_IExpression,
            0xA988BD41,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);

class IExpression {
public:
```

```
    STDMETHOD(QueryInterface(REFIID, void**)) PURE;
    STDMETHOD_(ULONG, AddRef())               PURE;
    STDMETHOD_(ULONG, Release())              PURE;

    STDMETHOD_(BSTR,GetExpression()) PURE;
    STDMETHOD_(void, SetExpression(BSTR, BOOL)) PURE;
    STDMETHOD_(BOOL, Validate())      PURE;
    STDMETHOD_(long, Evaluate())      PURE;
};

#endif  // inclusion guard
```

The information in **IEXP.H** is all we need to access the functionality of the `Expression` component. It defines the interface ID (IID) and each of the public methods provided by the component. In Chapter 6, we will implement the `Expression` component using the standard ActiveX interface `IDispatch`. Using `IDispatch`, we eliminate the client's need for any static component information like in the preceding code.

To use the `Expression` component, we need to instantiate a copy of it and obtain a pointer to the `IExpression` interface. We will add code to do this in the `CClientView` class contained in **VIEW.H** and **VIEW.CPP**. First, we need to forward declare the `IExpression` class; then we declare a member variable, `m_pIExp`, that will contain the `IExpression` interface pointer.

```
// view.h : interface of the CClientView class
//
/////////////////////////////////////////////////////////////////////////////

class IExpression;

class CClientView : public CFormView
{
protected: // create from serialization only
    CClientView();
    DECLARE_DYNCREATE(CClientView)

    // Add a pointer to IExpression
    IExpression* m_pIExp;

public:
    //{{AFX_DATA(CClientView)
    enum{ IDD = IDD_CLIENT_FORM };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA
...
};
```

When the `View` class is instantiated, we create an instance of the `Expression` component using its class factory. To obtain a pointer to the `Expression` class factory interface, we must use `CoGetClassObject`. `CoGetClassObject` requires a CLSID, so we first call `CLSIDFromProgID` to convert `Chap4.Expression.1` to the unique 128-bit CLSID. `CLSIDFromProgID` does the conversion by looking for the `Chap4.Expression.1` key in the Registry and returning the associated CLSID.

`CoGetClassObject` first looks in COM's table of registered class objects. If the object isn't found, `CoGetClassObject` queries the Registry to determine how to invoke the EXE or DLL. If the component is housed in a DLL, as in our case, it checks to see whether the DLL is loaded. If it is not, the function calls `LoadLibrary` to load the DLL. Once the DLL is loaded, the entry point `DllGetClassObject` is called with the CLSID that is passed. Add the following to **VIEW.CPP**:

```
// view.cpp : implementation of the CClientView class
//

#include "stdafx.h"
#include "client.h"
#include "document.h"
#include "view.h"

// Define INITGUID so that IID_IExpression is defined only once
#define INITGUID
#include "iexp.h"

// To support Unicode conversion in 4.0
#if ( _MFC_VER >= 0x400 )
   #include <afxpriv.h>
#endif
...

CClientView::CClientView()
   : CFormView(CClientView::IDD)
{
   //{{AFX_DATA_INIT(CClientView)
      // NOTE: the ClassWizard will add member initialization here
   //}}AFX_DATA_INIT
   // TODO: add construction code here

   LPCLASSFACTORY lpClassFactory;
   HRESULT hr;
   CLSID Clsid;

   m_pIExp = NULL;

// Convert the file contents to Unicode for ver 4.0
```

```
#if ( _MFC_VER >= 0x400 )
   USES_CONVERSION
   LPCOLESTR lpOleStr = T2COLE( "Chap4.Expression.1" );
   hr = ::CLSIDFromProgID( lpOleStr, &Clsid );
#else
   hr = ::CLSIDFromProgID( "Chap4.Expression.1", &Clsid );
#endif
   if( FAILED( hr ))
   {
      AfxMessageBox( "CLSIDFromProgID failed\n" );

      return;
   }

   hr = ::CoGetClassObject( Clsid,
                            CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER,
                            NULL,
                            IID_IClassFactory,
                            (LPVOID FAR *) &lpClassFactory );
   if ( FAILED( hr ) )
   {
      AfxMessageBox( "CoGetClassObject failed" );
      return;
   }

   hr = lpClassFactory->CreateInstance( NULL,
                                        IID_IExpression,
                                        (LPVOID FAR *) &m_pIExp );
   if ( FAILED( hr ) )
   {
      m_pIExp = NULL;
      AfxMessageBox( "ClassFactory->CreateInstance failed" );
      return;
   }

   lpClassFactory->Release();
}
```

Once we have a pointer to `IClassFactory`, we call `CreateInstance`, requesting the `IID_IExpression` interface. If all goes well, the interface pointer is returned. We are then finished with the class factory object and call `Release` through its interface, `IClassFactory`. At this point **SERVER.DLL** is loaded, and it contains an instance of the `Expression` class.

When the application terminates, we need to clean up by releasing the `IExpression` interface, which allows the DLL to unload. Add the following code to the `CClientView` destructor:

```
CClientView::~CClientView()
{
   if ( m_pIExp )
      m_pIExp->Release();
}
```

When the application user clicks the **Validate** or **Evaluate** button, we need to call the appropriate function in the `Expression` component. We already have a pointer to the `IExpression` interface, so this is easy. The code is very similar to that in Chapter 3, where we were using the C++ class directly. The only difference is that we now use an interface pointer instead.

```
void CClientView::OnEvaluate()
{
   CString strExpression;
   char szTemp[128];

   if ( m_pIExp == NULL )
   {
      AfxMessageBox( "No Interface to Expression" );
      return;
   }

   // Get the expression from the entry field
   CWnd* pWnd = GetDlgItem(IDC_EXPRESSION);
   pWnd->GetWindowText( strExpression );

   TRACE1( "OnEvaluate: Expression is %s\n", strExpression );

   BSTR bstrExp = strExpression.AllocSysString();
   m_pIExp->SetExpression( bstrExp, TRUE );
   ::SysFreeString( bstrExp );

   long lResult = m_pIExp->Evaluate();
   sprintf( szTemp, "%ld", lResult );

   pWnd->SetWindowText( szTemp );

   // Set focus back to the entry field

   GetDlgItem( IDC_EXPRESSION )->SetFocus();
}

void CClientView::OnValidate()
```

```
{
   CString strExpression;

   if ( m_pIExp == NULL )
   {
      AfxMessageBox( "No Interface to Expression" );
      return;
   }

   // Get the expression from the entry field
   CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
   pWnd->GetWindowText( strExpression );

   TRACE1( "OnValidate: Expression is %s\n", strExpression );

   BSTR bstrExp = strExpression.AllocSysString();
   m_pIExp->SetExpression( bstrExp, TRUE );
   ::SysFreeString( bstrExp );

   if (! m_pIExp->Validate() )
      AfxMessageBox( "Invalid Expression, try again" );

   //Set focus back to the entry field
   GetDlgItem(IDC_EXPRESSION)->SetFocus();
}
```

After adding all the preceding code, compile and link the CLIENT application.

Debugging the Client Application

Because our server is in-process, it is easy to debug both the client and server pieces. When running the client executable, you can step into the `IExpression` interface calls and go directly into the server code. Debugging a local server process is more difficult, but with Windows NT you can load multiple copies of the Visual C++ debugger and debug two processes simultaneously.

Summary

In this chapter our goal was to understand what Microsoft's Component Object Model is, what it can provide, and how it is implemented. We learned that COM provides a binary standard way for various languages and processes to interoperate and that this is done primarily through component interfaces. Component class interfaces are best developed using C++ because COM's internal structure is heavily dependent on a structure called a Vtable; Vtables are built using the virtual function mechanisms of C++. All

COM objects must implement an interface called `IUnknown` that provides the ability for a client process to query for an interface on a given component. `IUnknown` also provides methods that help with the management of a component object's lifetime.

Almost all COM objects expose multiple interfaces. To provide multiple interfaces in C++, we need to provide multiple Vtables for our class. We looked at three methods of implementing multiple interfaces in C++: multiple inheritance, interface implementations, and class nesting. Class nesting is the method used by MFC.

GUIDs are 128-bit unique identifiers that are used to identify COM object classes as well as the interfaces that these classes expose. The Windows Registry is used by COM to register component classes and provides a mechanism to locate a component by its unique CLSID or by a ProgID, which is a less unique, readable string that also identifies a component class.

Each COM object also must provide a class factory. The class factory is itself a COM object whose sole purpose is to create instances of another component class. These two components—the component class and its associated class factory—are the minimum requirements to provide a COM binary standard wrapper. These objects must be implemented within a component housing so that the components can be executed within the operating system.

There are two methods of housing components. In-process servers are implemented as DLLs and are typically faster than the other housing type, the local server or executable, because in-process servers don't require marshaling. Marshaling, the process of copying function arguments across these process or network boundaries, can greatly affect the performance of a component.

COM is a robust, system-level system standard on which higher-level application standards, such as ActiveX, can be built.

Chapter 5

COM, OLE, ActiveX, and the MFC Libraries

In Chapter 4 we discussed Microsoft's Component Object Model (COM). In this chapter, we will investigate the relationship of COM, OLE, and ActiveX. Once we understand that, we will look deep into the MFC libraries to see how MFC implements its abstracted view of these technologies. As we go along, we will convert the client and server applications of Chapter 4 to use MFC's COM-based classes instead of going directly to the COM/OLE APIs.

Our purpose in this chapter is to understand how MFC implements its COM support. With this knowledge, we can then move forward with a solid understanding of MFC-based COM technologies. In particular, we will answer two questions: what exactly are OLE and ActiveX? How are they related and how are they different?

This chapter focuses on the low-level details of how MFC implements support for COM, OLE, and ActiveX. You don't really need to master this low-level detail to build software components, but if you understand the low-level implementation, it's much easier to understand the how to effectively implement software based on these technologies.

What Is COM?

As the "Model" part of its name indicates, COM is a model for binary standard software development. Other vendors are free to implement the model using whatever mechanisms they choose, as long as the resulting implementation adheres to the model. Microsoft has provided an implementation of COM within its Windows operating systems. Windows provides COM-based system-level services that all software developers can use.

What Is OLE?

That's a difficult question to answer. A few years ago, OLE was an acronym for Object Linking and Embedding. OLE version 1.0 (circa 1991) was focused only on the linking and embedding of word processing and spreadsheet documents, but OLE version 2.0 (circa 1993) added many features that had nothing to do with *compound* documents. After the release of OLE 2.0, Microsoft stated that OLE was no longer an acronym for Object Linking and Embedding but instead was an umbrella term to describe all the features provided by OLE. These features include OLE automation, OLE controls, and several others—technologies that don't fit into the compound document area. The concept of a version number was also dropped. The technology can be, and is, constantly updated without affecting existing software.

In short, OLE is a well-defined set of COM-based interfaces (and a set of API functions that facilitate the use of these interfaces). That's it. OLE provides a robust, application-level implementation of COM that gives developers a new tool in the struggle to provide reusable software. The COM API functions we looked at in Chapter 4 provide the foundation on which to build this robust implementation on all the Windows platforms.

As an architecture, OLE is highly extensible. If new features are needed within OLE or in an operating system (such as Windows) that uses COM or OLE internally, it is easy to define a new OLE interface to provide the functionality. It can be delivered to users by providing a new in-process server or by adding the interface (and its supporting COM object and so on) to an existing system DLL (such as **OLE32.DLL**). This arrangement makes it easy to augment the OLE system environment and provides easy upgrading of existing capabilities. Windows 95 and Windows NT 4.0 make extensive use of OLE within their GUI shells. You extend the shell by writing DLLs, which are OLE in-process servers. And because of OLE's ability to expose additional interfaces on the same object, this addition of functionality in no way harms or affects existing software that may use the augmented interface. Once the COM infrastructure is in place, extension of various operating system functions is rather easy. Future Windows operating systems will make extensive use of this capability as Microsoft continues its quest to evolve Windows into a true object-oriented operating system.

What Is ActiveX?

Another difficult question. Before 1996, the term *OLE* was used to describe nearly all of the COM-based Windows development technologies. In April 1996, Microsoft unveiled its new Internet-based technology and changed many of the terms it had previously used to describe various COM-based technologies. The implementation and use of the technologies didn't change, but their names changed. ActiveX became the term used to describe Microsoft's Web-based technologies. Instead of OLE controls, we had ActiveX controls. Instead of OLE automation, the term was now Automation.

These technologies existed before April 1996, and their implementations still contain all the OLE interfaces from before; only the name has been changed. In addition, several new technologies were announced. ActiveX scripting is a new technology, although it is a collection of COM-based interfaces and possibly a few new APIs.

In a nutshell, COM is a system-level service provided by the Windows operating systems. OLE is a series of well-defined COM-based interfaces and a few APIs that provide a standard way of providing compound document support to applications. Similarly, ActiveX is a series of COM-based interfaces and a few APIs that provide a large number of application-level services to applications developers, especially those who focus on Web-based development. In many cases, the OLE and ActiveX interfaces intersect. Both technologies may define and use the same set of interfaces.

In other words, OLE and ActiveX (and DirectX) are a way of categorizing the technologies. In reality, they are just a collected group of COM-based interfaces and maybe a few APIs. That's it. Figure 5.1 illustrates the relationships among these technologies.

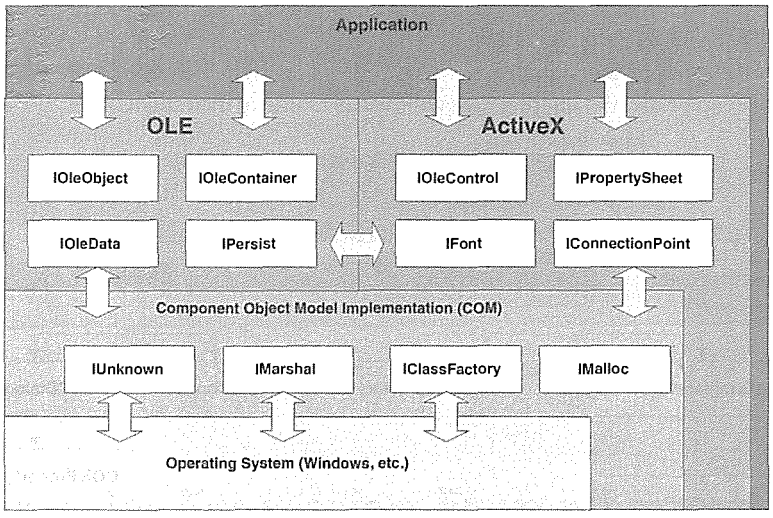


Figure 5.1 COM, OLE, and ActiveX relationships.

To make things simple, throughout the rest of the book I will use the term *ActiveX* because the technologies that we're focusing on come from this category. Automation and ActiveX controls are our primary focus. If I regress and use *OLE*, I'll explain why. I will also use the term *COM* when discussing functionality that supports both OLE and ActiveX. It is a good general term when you're discussing COM-based software. Also, the use of *ActiveX* can be confusing, because many of the interfaces and API calls are prefixed with "OLE" and not "ActiveX." Just keep in mind that, in most occurrences, OLE = ActiveX.

MFC and ActiveX

Even though ActiveX is a collection of COM-based interfaces, their large number and implementation requirements can make ActiveX a complex technology to grasp. The MFC libraries provide a way for a developer to start using ActiveX without having to understand all the details of what's going on "under the

covers." As we discussed in Chapter 3, one of the primary purposes of application frameworks is to provide an abstracted view of the low-level implementation of system technologies. MFC does this, and in this chapter I'll show you how MFC does it. We'll look under the covers of MFC so that we can use what we learned in Chapter 4.

Chapters 4 and 5 concern the low-level implementation details of COM, ActiveX, and MFC. You may not understand everything at first, but these chapters are here when you need them. After this chapter, we'll focus more on how to use ActiveX. As you gain experience with ActiveX, you will see more and more opportunities to use the technology in many areas of the software you develop. It is a powerful technology.

You will also find that MFC may not provide an abstraction or encapsulated class for the ActiveX features you require. When this occurs, a good understanding of what is going on at a low level will allow you to implement the technology yourself, with or without the help of MFC.

Figure 5.2 shows some of the important ActiveX classes provided by MFC. We will discuss the highlighted ones in this chapter. In later chapters we will discuss some of the others.

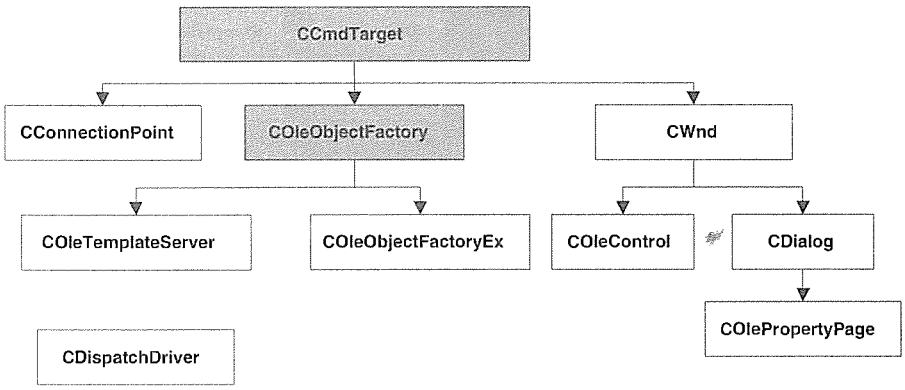


Figure 5.2 Important MFC ActiveX classes.

Interfaces and Grouped Functionality

The OLE and ActiveX technologies contain several hundred interfaces, and the number continues to grow as new capabilities are added. Many of these interfaces are grouped to describe various application technologies. Examples include compound documents, structured storage, drag and drop, Automation, ActiveX documents, and others. Most developers are familiar with OLE's compound document technology but may not be familiar with all the new things added in the latest ActiveX specification. We will focus on two technologies: Automation and ActiveX controls. As you'll see, they use many of the interfaces currently described by ActiveX. Figure 5.3 shows a view of some of the ActiveX interfaces we'll deal with when developing Automation servers and ActiveX controls.

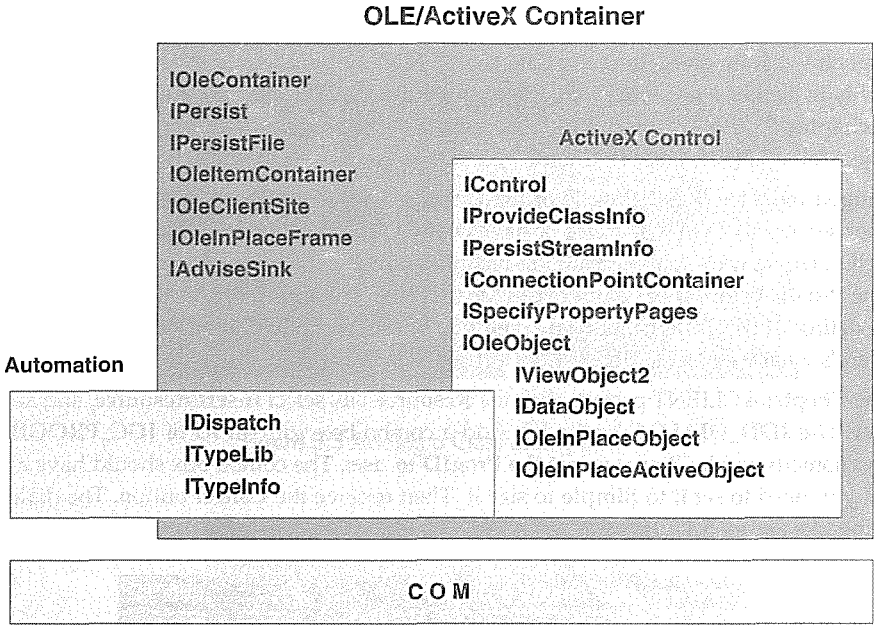


Figure 5.3 COM-based interfaces and application functionality.

Converting the Expression Examples to Use MFC

To illustrate how MFC implements COM/ActiveX, we will convert the CLIENT and SERVER applications from Chapter 4 to use the MFC libraries. Instead of using the native COM API functions and straight C++ constructs for the implementation, we will use MFC's abstracted implementation. This arrangement will illustrate the differences in the techniques and provide a better understanding of how MFC implements its ActiveX support.

Because we were doing certain things at a very low level, not everything will convert directly to MFC. MFC currently does not provide a wrappered ActiveX class for every ActiveX technology. This applies only to certain functions in the CLIENT module; everything in the SERVER will convert right over. As with all application frameworks, MFC cannot implement changes as quickly as they are added to the OS environment. There are areas of ActiveX that are not implemented in MFC, so the developer must use C++ and the APIs explicitly. When we encounter such situations, we will know enough to implement them ourselves.

Converting the Chapter 4 Client Application

There isn't much work involved in converting the CLIENT application from Chapter 4. The most difficult part is probably in deciding how you want to go about doing it. The changes are so minor you might simply modify the chapter CLIENT project as the following sections demonstrate. Or you could build a new CLIENT project following the steps in Chapter 4 and then continue with the items in the following section.

I've changed the CLSID and ProgID of the Chapter 5 server application so that we can distinguish it from the other servers that we will create or have created. In our client application we have hard coded the specific ProgID that we want to use. What I'm recommending is that we add a modal dialog box that allows us to enter the ProgID before we attempt to call the `CLSIDFromProgID` function. Using this method, we can modify the existing CLIENT application and continue to use it with both the Chapter 4 and Chapter 5 server examples. Here's what we need to do.

Open the Chapter 4 CLIENT project, click the **Resource** tab, select **Insert/Resource**, and add a new DIALOG resource. Use **IDD_DIALOG** as the ID. Add a combo box with an ID of **IDC_PROGID** and a static field that says something like **Please select the ProgID to use:**. The combo box should have a style of **Drop List**, but first you need to set it to **Simple** to size it. Then remove the **Cancel** button. The dialog box should look something like Figure 5.4.

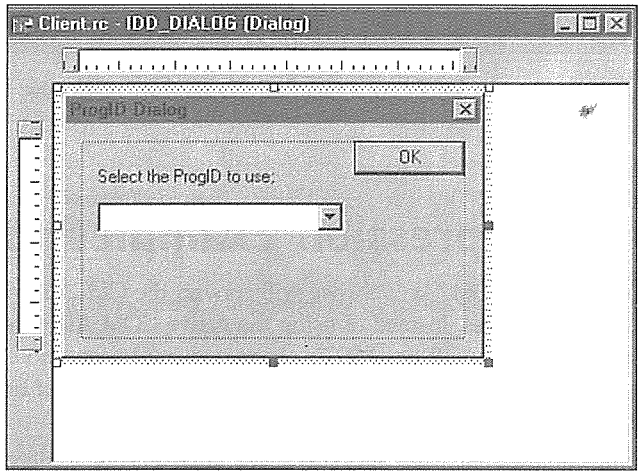


Figure 5.4 Server selection dialog box.

Now we need to create a class that encapsulates the dialog box we just built. We can easily do this using ClassWizard. If you invoke ClassWizard when editing a dialog resource, you will get a screen like that in Figure 5.5. Click **OK** to get the New Class dialog box. Add a new class with the name **CProgIDDlg**, deriving it from `CDialog`, and name the implementation files **PROGDLG.H** and **PROGDLG.CPP**. Then press **Create a new class**. Figure 5.6 depicts the screen before you click **Create**.

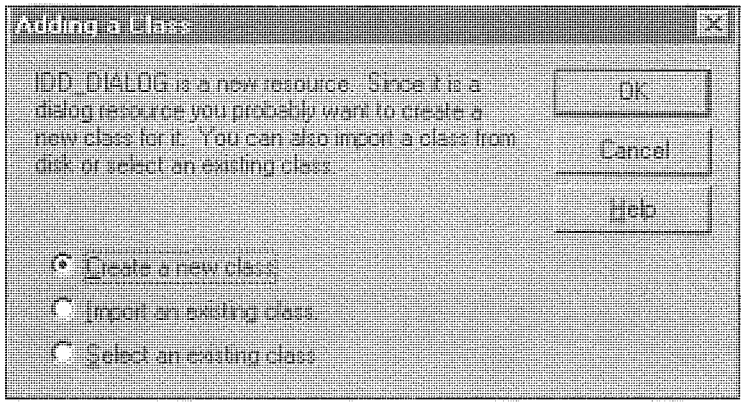


Figure 5.5 Adding a Class dialog box.

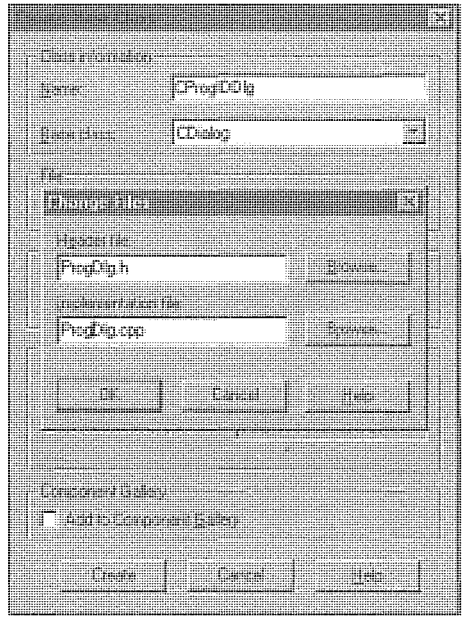


Figure 5.6 Adding a class with ClassWizard.

We've added a new class to the CLIENT project. Now we need to fill the combo box with valid ProgIDs when the dialog box is initially loaded. Go back into ClassWizard, select the **Message Map** tab, and choose **CProgIDDlg** as the class name. Override the `WM_INITDIALOG` message by selecting it and clicking the **Add**

Function button. This action adds a function called `OnInitDialog`. Add the following code to **PROGDLG.CPP** to load our combo box.

```
BOOL CProgIDDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // TODO: Add extra initialization here
   CComboBox* pCB = (CComboBox*) GetDlgItem( IDC_PROGID );
   // Add the valid strings to the combo box
   pCB->AddString( "Chap4.Expression.1" );
   pCB->AddString( "Chap5.Expression.1" );

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

NOTE

Drop list–style combo boxes can also be prefilled using the resource editor. After choosing the drop list style, switch to the General Properties page and you can enter the strings in a listbox on the right. Using this method, you won't have to override `OnInitDialog` just to prefill the list.

We need a variable in the `CProgIDDlg` class to contain the combo box selection when the user presses the **OK** button. Go into ClassWizard again and select the **Member Variable** tab with **CProgIDDlg** as the class name. Select **IDC_PROGID** and click the **Add Variable** button. Name the variable **m_strProgID** and select a category of **Value** and a type of **CString**. Click **OK**. Your screen should look like Figure 5.7.

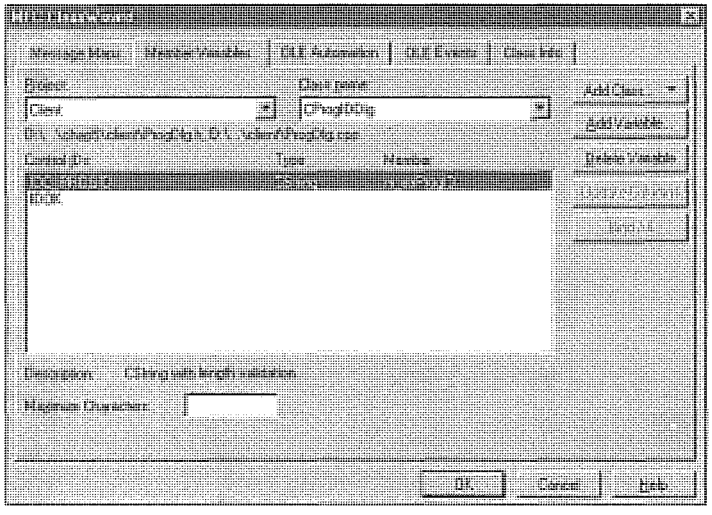


Figure 5.7 ClassWizard Member Variables dialog box.

Now add the following code to **VIEW.CPP**. This creates an instance of the `CProgIDDlg` class and then calls its `DoModal` method to display the dialog box. After the user presses **OK**, the `DoModal` method returns with the `m_strProgID` variable containing the combo box selection. We then use that value to get our CLSID.

```
//
// View.cpp : implementation of the CClientView class
//

...

// Define INITGUID so that IID_IExpression is defined only once
#define INITGUID
#include "iexp.h"
// To support Unicode conversion in 4.0
#if ( _MFC_VER >= 0x400 )
   #include <afxpriv.h>
#endif

// Include the definition of our new dialog
#include "progdlg.h"

...

CClientView::CClientView() : CFormView(CClientView::IDD)
{
...
   // Create an instance of our new dialog class
   CProgIDDlg Dlg;
   // Invoke it modally
   Dlg.DoModal();

// To support Unicode conversion in 4.0
#if ( _MFC_VER >= 0x400 )
   USES_CONVERSION;
   hr = ::CLSIDFromProgID( T2COLE( Dlg.m_strProgID ), &Clsid );
#else
   hr = ::CLSIDFromProgID( Dlg.m_strProgID, &Clsid );
#endif
   if (FAILED( hr ))
   {
      AfxMessageBox( "CLSIDFromProgID failed" );
      return;
   }
...
}
```

We can now use our CLIENT application to connect to either the Chapter 4 or Chapter 5 server example. Compile, link, and run the application to make sure that you can connect to the Chapter 4 server. You can try to connect to the Chapter 5 server, but because we haven't built it yet, you'll most likely get an error.

Initializing the ActiveX Environment

In Chapter 4, our CLIENT application had to initialize the COM libraries before using any COM components. As we've discussed, ActiveX is the technology that provides most of the application-level functionality we need when we're developing software components. Instead of initializing the COM environment, from now on we will initialize the ActiveX environment. The OLE/ActiveX API function `OleInitialize` is very similar to `CoInitialize`. `OleInitialize` initializes not only COM but also the ActiveX libraries. MFC provides a helper function, `AfxOleInit`, that checks the version of the ActiveX DLLs and initializes the environment by calling `OleInitialize`. In **CLIENT.CPP**, we comment out the explicit COM API calls and use MFC's `AfxOleInit` function instead.

```
// Initialize COM
//HRESULT hr = ::CoInitialize( NULL );
//if ( FAILED( hr ) )
//{
//  AfxMessageBox( "Unable to Initialize COM, exiting" );
//   return FALSE;
//}

// Call this MFC function instead
if ( AfxOleInit() == 0 )
{
    AfxMessageBox( "Unable to Initialize OLE, exiting" );
    return FALSE;
}
```

We also terminated the use of the COM environment by calling `CoUninitialize`. We no longer need to worry about this when we use the MFC `AfxOleInit` function. MFC ensures that the appropriate uninitialize functions are called when the application terminates, so we comment out the `CoUninitialize` call.

```
int CClientApp::ExitInstance()

{

   // Shutdown COM
      // No need to do this if we use MFC's AfxOleInit()
   //::CoUninitialize();
   return CWinApp::ExitInstance();
}
```

That does it for the client piece. Because we are accessing a COM object with an ActiveX custom interface, little is required in the client code to use an ActiveX component. Most of the work required is in the implementation of the ActiveX server, be it an in-process server or a local server. Let's convert the server application from Chapter 4 to use MFC exclusively.

Converting the Chapter 4 Server Application

In the next few sections we will convert the Chapter 4 server application. In the process, we will investigate how MFC implements COM and ActiveX. The first MFC class that we will encounter is `CCmdTarget`, which provides much of the ActiveX functionality we need when dealing with ActiveX interfaces. The second MFC class that we'll explore is `COleObjectFactory`. Aptly named for what it does, `COleObjectFactory` implements COM's `IClassFactory` interface as well as MFC's implementation of the class factory. MFC takes care of instantiating our MFC ActiveX objects, so we don't have to write very much code to get class factory capabilities.

Following is the **EXPSVR.H** file from the `Expression` component of Chapter 4.

```
//
// ExpSvr.h
//

DEFINE_GUID( CLSID_Expression,
             0xA988BD40,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);
DEFINE_GUID( IID_IExpression,
             0xA988BD41,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);

// implementation class declarations...
// Token, CStringStack, etc.
...

class IExpression : public IUnknown {
public:
   STDMETHOD_(BSTR,GetExpression()) PURE;
   STDMETHOD_(void, SetExpression(BSTR, BOOL)) PURE;
   STDMETHOD_(BOOL, Validate())      PURE;
   STDMETHOD_(long, Evaluate())      PURE;
};

class Expression : public IExpression
{
protected:
   enum TokenType
```

```
    {
        BogusToken,
        OperatorToken,
        OpenParenToken,
        CloseParenToken,
        NumberToken
    };

protected:
    CString         m_strExpression;
    BOOL            m_bInfix;

protected:
    DWORD           m_dwRef;

public:
    Expression();
    ~Expression();

protected:
    BOOL            IsNumber( const CString& strToken );
    TokenType       GetTokenType( const CString& strToken );
    int             Precedence( const CString& strToken );
    BOOL            InfixToPostfix();

public:
    STDMETHOD( QueryInterface( REFIID, void** ));
    STDMETHOD_(ULONG, AddRef());
    STDMETHOD_(ULONG, Release());

    STDMETHOD_(BSTR, GetExpression());
    STDMETHOD_(void, SetExpression( BSTR, BOOL ));
    STDMETHOD_(BOOL, Validate());
    STDMETHOD_(long, Evaluate());
};

class ExpClassFactory : public IClassFactory
{
protected:
    DWORD           m_dwRef;

public:
    ExpClassFactory();
    ~ExpClassFactory();
```

```
    // IUnknown
    STDMETHOD( QueryInterface(REFIID, void** ));
    STDMETHOD_(ULONG, AddRef());
      STDMETHOD_(ULONG, Release());
    // IClassFactory
    STDMETHOD( CreateInstance(LPUNKNOWN, REFIID, void**));
    STDMETHOD( LockServer(BOOL));
};
```

We need to do quite a few things here to convert the interface, the implementation, and the class factory so that they use MFC's ActiveX classes. You will see that much of our original code is not required when we use MFC's implementation. Let's go through each line of code and convert it to MFC.

```
//
// ExpSvr.h
//

//DEFINE_GUID( CLSID_Expression,
//             0xA988BD40,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,
//             0x5A,0xFB,0x7D,0x30);

DEFINE_GUID( IID_IExpression,
             0xA988BD41,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,
             0x5A,0xFB,0x7D,0x30);
```

We comment out the `DEFINE_GUID` macro that defines our CLSID. We will use MFC's method of defining a CLSID later when we convert our class factory. The `IID_IExpression` `DEFINE_GUID` macro will stay the same, because we are implementing a custom ActiveX interface and MFC has no knowledge of the IID of the component.

```
class IExpression : public IUnknown {
public:
   STDMETHOD_(BSTR,GetExpression()) PURE;
   STDMETHOD_(void, SetExpression(BSTR, BOOL)) PURE;
   STDMETHOD_(BOOL, Validate())      PURE;
   STDMETHOD_(long, Evaluate())      PURE;
};
```

Nothing changes here, either, for the same reasons. This is a custom interface, and the `IExpression` class provides the Vtable definition for use by the component user.

The majority of the changes are to the `Expression` component class. The declaration for the class changes from this:

```
class Expression : public IExpression
```

to this:

```
class Expression : public CCmdTarget
```

This change—deriving from `CCmdTarget` instead of `IExpression`—automatically provides significant functionality for the `Expression` component class. Let's take a closer look at `CCmdTarget`.

CCmdTarget

The `CCmdTarget` class is a workhorse when it comes to the implementation of ActiveX within MFC. Not only does it provide a great deal of ActiveX functionality, but it also provides the message mapping mechanism we discussed in Chapter 3 and the Automation `IDispatch` mapping features that we will discuss in Chapter 6. `CCmdTarget` provides basic mapping support for MFC in general. What we need to understand about `CCmdTarget` for now is how it implements the `IUnknown` interface and its functions: `QueryInterface`, `AddRef`, and `Release`.

Following is a partial list of the `CCmdTarget` class declaration from **AFXWIN.H**:

```
class CCmdTarget : public CObject
{
...
// ActiveX interface map implementation
public:
   // data used when CCmdTarget is made ActiveX aware
   DWORD     m_dwRef;
   LPUNKNOWN m_pOuterUnknown;  // external controlling unknown if != NULL
   DWORD     m_xInnerUnknown;  // place-holder for inner controlling unknown

   DECLARE_INTERFACE_MAP()

public:
   // these versions do not delegate to m_pOuterUnknown
      DWORD InternalQueryInterface(const void*, LPVOID* ppvObj);
   DWORD InternalAddRef();
   DWORD InternalRelease();
   // these versions delegate to m_pOuterUnknown
   DWORD ExternalQueryInterface(const void*, LPVOID* ppvObj);
   DWORD ExternalAddRef();
   DWORD ExternalRelease();

   // implementation helpers
   LPUNKNOWN GetInterface(const void*);
...
};
```

There are a few things in CCmdTarget that we recognize; the reference count variable (m_dwRef) and some methods that look similar to what we need for the IUnknown interface except that they are prefixed with Internal and External. What's this all about? We'll see in a moment. CCmdTarget supports reference counting and the IUnknown interfaces using a technique called *interface maps*. Each class derived from CCmdTarget inherits this capability, but to use it certain steps must be followed. Did you notice the macro DECLARE_INTERFACE_MAP? It expands (via the preprocessor) to this:

```
private: \
   static const AFX_INTERFACEMAP_ENTRY _interfaceEntries[]; \
protected: \
   static AFX_DATA const AFX_INTERFACEMAP interfaceMap; \
   virtual const AFX_INTERFACEMAP* GetInterfaceMap() const; \
```

The DECLARE_INTERFACE_MAP macro provides a static array of interface IDs (IIDs) for a given class that is derived from CCmdTarget. This table is used by QueryInterface to look up the interface asked for in the REFIID parameter. CCmdTarget itself has a base interfaceMap table that is queried if the derived class does not provide an implementation. I'm sure this is a little confusing at this point, but we had to start somewhere. Back to the Expression class.

```
class Expression : public CCmdTarget
{
...
public:
   STDMETHOD( QueryInterface( REFIID, void** ));
   STDMETHOD_(ULONG,  AddRef());
   STDMETHOD_(ULONG,  Release());

   STDMETHOD_(BSTR, GetExpression());
   STDMETHOD_(void, SetExpression( BSTR, BOOL ));
   STDMETHOD_(BOOL, Validate());
   STDMETHOD_(long, Evaluate());
};
```

To provide an interface map for the Expression class, we need to use DECLARE_INTERFACE_MAP so that we have an interface table that contains our IUnknown and IExpression interfaces. Because every interface must also contain an IUnknown interface, MFC provides some easy-to-use macros for declaring interfaces. These macros use the C++ nested class idiom that we described in the Chapter 4. The preceding interface declaration becomes:

```
DECLARE_INTERFACE_MAP()

BEGIN_INTERFACE_PART( Expression, IExpression )
   STDMETHOD_(BSTR, GetExpression());
   STDMETHOD_(void, SetExpression( BSTR, BOOL ));
   STDMETHOD_(BOOL, Validate());
```

```
   STDMETHOD_(long, Evaluate());
END_INTERFACE_PART( Expression )
```

This code declares a nested class with the name `XExpression` and provides the declaration for the three methods of `IUnknown`. Here we derive from our `IExpression` interface class to ensure that our Vtable is present in the nested class. MFC's macros tend to obscure my understanding at times, so I've "preprocessed" them for you. Here's what's really going on:

```
#define BEGIN_INTERFACE_PART(localClass, baseClass) \
   class X##localClass : public baseClass \
   { \
   public: \
      STDMETHOD_(ULONG, AddRef)(); \
   STDMETHOD_(ULONG, Release)(); \
   STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj); \
```

`BEGIN_INTERFACE_PART` declares the nested class and provides the declaration for the `IUnknown` members. It also builds the nested class name by prepending an `X` to the outer class name. In this case, it becomes `XExpression`.

Then you declare your own interface methods using the standard COM `STDMETHOD_` macro as before:

```
  STDMETHOD_(BSTR, GetExpression());
     STDMETHOD_(void, SetExpression( BSTR, BOOL ));
  STDMETHOD_(BOOL, Validate());
  STDMETHOD_(long, Evaluate());

#define END_INTERFACE_PART(localClass) \
   } m_x##localClass; \
   friend class X##localClass; \
```

Once all the interface methods are declared, you use the MFC macro `END_INTERFACE_PART( localClass )` to end the class declaration, nest an instance of the class, and finally make the nested class a `friend` of the parent class, just as we did in Chapter 4 when implementing multiple COM interfaces. MFC always uses class nesting even if there is only one interface (not counting `IUnknown`). Following is declaration before and after preprocessing:

```
// Before
BEGIN_INTERFACE_PART( Expression, IExpression )
   STDMETHOD_(BSTR, GetExpression());
   STDMETHOD_(void, SetExpression( BSTR, BOOL ));
   STDMETHOD_(BOOL, Validate());
   STDMETHOD_(long, Evaluate());
END_INTERFACE_PART( Expression )

// After preprocessing
```

```
class XExpression : public IExpression
{
   STDMETHOD_(ULONG, AddRef)();
   STDMETHOD_(ULONG, Release)();
   STDMETHOD(QueryInterface)(REFIID iid, LPVOID* ppvObj);

   STDMETHOD_(BSTR, GetExpression());
   STDMETHOD_(void, SetExpression( BSTR, BOOL ));
   STDMETHOD_(BOOL, Validate());
   STDMETHOD_(long, Evaluate());
} m_xExpression;
friend class XExpression;
```

There you have it. We have declared our new MFC-compatible COM interface. The declaration for the `IUnknown` methods comes from our earlier, unchanged declaration of `IExpression`. We're not quite finished with the `Expression` class. As we will see, `CCmdTarget` takes care of our reference counting, so we can remove the member variable that was responsible for that.

```
class Expression : public CCmdTarget
{
...
protected:
   // Remove the reference count variable
   //DWORD           m_dwRef;

...
};
```

We've almost completed the conversion of the `Expression` class declaration to MFC. Now let's look at the implementation.

In **EXPSVR.H** we declared an interface map using `DECLARE_INTERFACE_MAP`. This macro declared `interfaceMap` and `InterfaceEntries`, two static member variables of `Expression`. Because these variables were declared static, they must be initialized at compile time. We do this in **EXPSVR.CPP**:

```
// ExpSvr.cpp
...
BEGIN_INTERFACE_MAP( Expression, CCmdTarget )
   INTERFACE_PART( Expression, IID_IExpression, Expression )
END_INTERFACE_MAP()
```

That's all there is to it. The `BEGIN_INTERFACE_MAP` macro defines the `GetInterfaceMap` member function and begins the table entry for our interface map. `BEGIN_INTERFACE_MAP` also defines our `interfaceMap` member, which points to `CCmdTarget` (the parent class) and to `Expression` (the derived

class). This makes it easy to search through the derivation hierarchy looking for a particular COM interface when a client uses the `QueryInterface` function.

```
#define BEGIN_INTERFACE_MAP(theClass, theBase) \
    const AFX_INTERFACEMAP* theClass::GetInterfaceMap() const \
       { return &theClass::interfaceMap; } \
   const AFX_DATADEF AFX_INTERFACEMAP theClass::interfaceMap = \
      { &theBase::interfaceMap, &theClass::_interfaceEntries[0], }; \
   const AFX_DATADEF AFX_INTERFACEMAP_ENTRY theClass::_interfaceEntries[] = \
 {\
```

Each `INTERFACE_PART` macro defines one COM interface entry in our map. The `offsetof` macro is used to directly access the address of the Vtable of the nested class instance, `m_xExpression`, within `Expression`.

```
#define INTERFACE_PART(theClass, iid, localClass) \
   { &iid, offsetof(theClass, m_x##localClass) }, \
```

Finally, the `END_INTERFACE_MAP` macro terminates the interface map with an identifiable signature.

```
#define END_INTERFACE_MAP() \
   { NULL, (size_t)-1 } \
   }; \
```

After preprocessor expansion, we get the following:

```
const AFX_INTERFACEMAP* Expression::GetInterfaceMap() const
{ return &Expression::interfaceMap; }
const AFX_DATADEF AFX_INTERFACEMAP Expression::interfaceMap =
{ &CCmdTarget::interfaceMap, &Expression::_interfaceEntries[0], };
const AFX_DATADEF AFX_INTERFACEMAP_ENTRY Expression::_interfaceEntries[] =
{
   { &IID_IExpression, offsetof( Expression, m_xExpression ) },
   { NULL, (size_t) - 1 }
};
```

For debugging purposes it's nice to go ahead and preprocess the code by hand so that you can step through it much more easily. In the example programs for this chapter, I've provided the preprocessed output and have commented out the MFC macro. This makes debugging much more understandable. But once you understand what's going on, you need to use the macros provided by MFC, because their implementation may change with newer versions of the libraries.

Figure 5.8 illustrates what the `INTERFACE_MAP` macros have created for our `Expression` class.

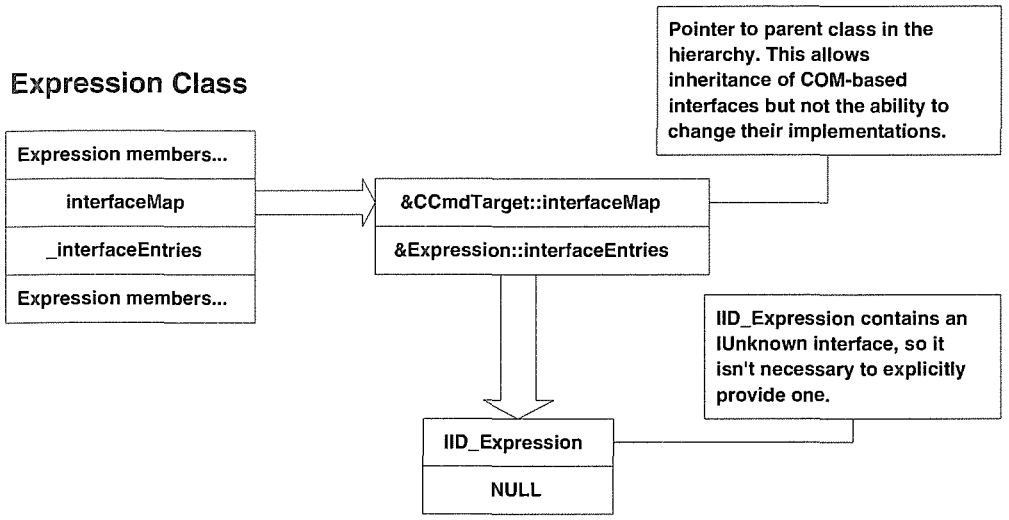


Figure 5.8 Interface maps.

To get a sense of how this works, let's look at `CCmdTarget`'s implementation of `QueryInterface`. The following code provides the gist of `CCmdTarget`'s implementation. (Some of the details have been left out to protect the innocent.) As you will see in the next section, your class will implement the true `QueryInterface` method, but it will defer to one of `CCmdTarget`'s `InternalQueryInterface` or `ExternalQueryInterface` methods:

```
// The QueryInterface that is exported to normal clients
// This is the method we will use when implementing our
// our QueryInterface
DWORD CCmdTarget::ExternalQueryInterface(const void* iid, LPVOID* ppvObj)
{
   // delegate to controlling unknown if aggregated
   if (m_pOuterUnknown != NULL)
      return m_pOuterUnknown->QueryInterface(*(IID*)iid, ppvObj);

   // If we're not part of an aggregate, call the
   // InternalQueryInterface method below
   return InternalQueryInterface(iid, ppvObj);
}

// real implementation of QueryInterface
DWORD CCmdTarget::InternalQueryInterface(const void* iid, LPVOID* ppvObj)
{
   // check local interfaces. If the IID is found
```

```
   // assign it to the provided void** pointer
   if ((*ppvObj = GetInterface(iid)) != NULL)
   {
      // interface was found - add a reference
      ExternalAddRef();
      return NOERROR;
   }
...
   // interface ID not found, fail the call
   return (DWORD)E_NOINTERFACE;
}

// This function spins through the InterfaceEntries of all of the classes
// derived from CCmdTarget looking for the requested interface.
LPUNKNOWN CCmdTarget::GetInterface(const void* iid)
{
...
   // Get the InterfaceMap of the most derived class
   // GetInterfaceMap was implemented by the BEGIN_INTERFACE_MAP macro
   const AFX_INTERFACEMAP* pMap = GetInterfaceMap();
...
   // Walk the interface map to find the IID
   do
   {
      // Walk through each interface entry for the class
      const AFX_INTERFACEMAP_ENTRY* pEntry = pMap->pEntry;
      while (pEntry->piid != NULL)
      {
         if (*(IID*)pEntry->piid == *(IID*)iid)
         {
            // check INTERFACE_ENTRY macro
            LPUNKNOWN lpUnk = GetInterfacePtr(this, pEntry);

            // check vtable pointer (can be NULL)
            if (*(DWORD*)lpUnk != 0)
               return lpUnk;
         }

         // entry did not match - keep looking
         ++pEntry;
      }
```

```
    // While there are more entries in the map
    // This walks "backwards" in the map from the most derived
    // class up the hierarchy eventually to CCmdTarget itself.
    } while ((pMap = pMap->pBaseMap) != NULL);

    // interface ID not found, fail the call
    return NULL;
}
```

Let's look into the implementation of each method of `Expression`. Following are the default constructor and destructor for `Expression`. Only a default constructor is supplied, because MFC will dynamically create an instance of `Expression` whenever `IClassFactory::CreateInstance` is called. We'll discuss `IClassFactory` in a moment. `CCmdTarget` provides the `AfxOleLockApp` and `AfxOleUnlockApp` functions, which are similar to the `IClassFactory::LockServer` method. Microsoft recommends that you call `AfxOleLockApp` in the constructor, and `AfxOleUnlockApp` in the destructor, of a COM component class. The `CCmdTarget` method `OnFinalRelease` is called when the reference count of an object reaches zero. The default implementation in `CCmdTarget` calls `delete this` to destroy the object.

```
Expression::Expression()
{
    // Increment the active object count
    AfxOleLockApp();
    m_bInFix = TRUE
}

Expression::~Expression()
{
    // decrement the active object count
    AfxOleUnlockApp();
}
```

The `BEGIN_INTERFACE_PART` macro created a nested class named `XExpression` and declared the `IUnknown` members, so let's implement them:

```
STDMETHODIMP_(ULONG) Expression::XExpression::AddRef()
{
    METHOD_PROLOGUE( Expression, Expression )
    return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) Expression::XExpression::Release()
{
    METHOD_PROLOGUE( Expression, Expression )
    return pThis->ExternalRelease();
```

```
}

STDMETHODIMP Expression::XExpression::QueryInterface( REFIID iid, LPVOID far *pvvObj )
{
    METHOD_PROLOGUE( Expression, Expression )
    return (HRESULT) pThis->ExternalQueryInterface( &iid, pvvObj );
}
```

This code should look familiar. In Chapter 4, we discussed the use of nested classes. The `METHOD_PROLOGUE` macro uses the `offsetof` macro to calculate the `this` pointer of the nesting class. In Chapter 4 we delegated the `IUnknown` methods to the nesting class just as we are doing here, but now we're delegating to `CCmdTarget`'s `External` functions. `CCmdTarget` provides two implementations of each `IUnknown` method: an internal and an external function. These differences concern the support of component aggregation. In most cases, it's beneficial to provide aggregation ability—it doesn't cost much—so you should use the `External` methods provided by `CCmdTarget`.

We also need to change the implementation of our exposed interface methods. Some of the implementation of the expression functionality is in the outer, or nesting class, but the interface is provided via the nested class. So we have to scope down using multiple scope resolution operators when implementing the methods, just as we did above with the IUnknown interface. We also need to use the METHOD_PROLOGUE macro to access the implementation members in the nesting Expression class. Here are the changes for our four IExpression interface methods:

```
STDMETHODIMP_(BSTR) Expression::XExpression::GetExpression()
{
   METHOD_PROLOGUE( Expression, Expression )

// To support Unicode conversion in 4.0 and above
#if ( _MFC_VER >= 0x400 )
   USES_CONVERSION;
   return ::SysAllocString( T2COLE( pThis->m_strExpression ) );
#else
   return ::SysAllocString( pThis->m_strExpression );
#endif
}

STDMETHODIMP_(void) Expression::XExpression::SetExpression( BSTR bstrExp, BOOL bInfix )
{
   METHOD_PROLOGUE( Expression, Expression )
   pThis->m_strExpression = bstrExp;
   pThis->m_bInfix = bInfix;
}
```

```
STDMETHODIMP_(BOOL) Expression::XExpression::Validate()
{
   METHOD_PROLOGUE( Expression, Expression )
...
   tokenizer.SetString( pThis->m_strExpression );
   // Tokenize our expression
   tokenizer.Tokenize();

   // Check for validity
   while( tokenizer.GetToken( strToken ) )
   {
      switch( pThis->GetTokenType( strToken ) )
      {
      ...
      }
    ...
   }
...
}

STDMETHODIMP_(long) Expression::XExpression::Evaluate()
{
   METHOD_PROLOGUE( Expression, Expression )
...
   if ( pThis->m_bInfix )
   {
      pThis->InfixToPostfix();
   }

   tokenizer.SetString( pThis->m_strExpression );
   tokenizer.Tokenize();

   // While there are tokens to process
   while( tokenizer.GetToken( strToken ) )
   {
      switch( pThis->GetTokenType( strToken ) )
      {
      ...
      }
     }
 ...
}
```

By deriving from `CCmdTarget`, we no longer need to keep track of our reference counts inside the implementing class. We also no longer need to keep track of a controlling unknown for aggregation. Now let's turn our attention to the class factory, `ExpClassFactory`.

Class Factories

As you may recall from Chapter 4, each COM object requires the services of a class factory (which itself is a COM object). This arrangement allows client applications to create instances of your COM object. Client applications gain access to the class factory by using the function `CoGetClassObject`, which uses one of two methods. If the component object is housed within an in-process server, `CoGetClassObject` calls the `DllGetClassObject` entry point within the DLL. If the component object is housed in a local server, COM queries the active object table to determine whether the component's class factory is already running and therefore registered. If it is, COM returns the class factory, and so on. The important thing is that the class factories for a given housing, be it in-process or executable, are always available for a client process to access.

In our Chapter 4 server example, we provided a C++ class, `ExpClassFactory`, that implemented the `CreateInstance` function of `IClassFactory` that created instances of our `Expression` class. Here's the declaration from **EXPSVR.H**:

```
// Expsvr.h
...
class ExpClassFactory : public IClassFactory
{
protected:
   DWORD           m_dwRef;

public:
   ExpClassFactory();
   ~ExpClassFactory();

   // IUnknown
   STDMETHODIMP         QueryInterface(REFIID, void** );
   STDMETHODIMP_(ULONG) AddRef();
   STDMETHODIMP_(ULONG) Release();
   // IClassFactory
   STDMETHODIMP         CreateInstance(LPUNKNOWN, REFIID, void**);
   STDMETHODIMP         LockServer(BOOL);
};
```

MFC implements the class factory for a component by adding a static class member for each component class in your server application. This static member is an instance of the MFC class `COleObjectFactory`. Adding a class factory to your component is easy; you just use a few MFC macros. When you're using MFC, it isn't necessary to explicitly declare and implement a separate class factory class for each of your components as we did in Chapter 4. Let's look at `COleObjectFactory` in more detail.

COleObjectFactory

COleObjectFactory implements each component's class factory, provides facilities to register each class factory with COM, and provides a mechanism for programmatically updating the system Registry (so that you don't have to distribute a **.REG** file with your component).

As we discussed earlier, MFC implements its COM/ActiveX interface classes by deriving from CCmdTarget. Such is the case with the COleObjectFactory class. Its declaration goes something like this:

```
class COleObjectFactory : public CCmdTarget
{
...
   DECLARE_DYNAMIC(COleObjectFactory)
...
public:
   COleObjectFactory* m_pNextFactory;  // list of factories maintained

protected:
   DWORD m_dwRegister;              // registry identifier
   CLSID m_clsid;                   // registered class ID
   CRuntimeClass* m_pRuntimeClass; // runtime class of CCmdTarget derivative
   BOOL m_bMultiInstance;           // multiple instance?
   LPCTSTR m_lpszProgID;            // human readable class ID

// Interface Maps
public:
   BEGIN_INTERFACE_PART(ClassFactory, IClassFactory)
       INIT_INTERFACE_PART(COleObjectFactory, ClassFactory)
       STDMETHOD(CreateInstance)(LPUNKNOWN, REFIID, LPVOID*);
       STDMETHOD(LockServer)(BOOL);
   END_INTERFACE_PART(ClassFactory)

   DECLARE_INTERFACE_MAP()

    ...
};
```

The MFC COleObjectFactory provides a great deal of functionality for the developer. One purpose of COleObjectFactory is to maintain a list of all the COM component classes within a given component housing (EXE or DLL). We'll get to some of the details in a minute. For now let's focus on those aspects of COleObjectFactory that are important to our conversion of the SERVER example.

The COleObjectFactory constructor takes four arguments. Each argument is described in Table 5.1.

Table 5.1 COleObjectFactory Parameters

Parameter	Description
REFCLSID riid	A reference to the CLSID for the component that this class factory will support.
CRuntimeClass* pRuntimeClass	A pointer to the run-time class. This provides the class factory with the ability to dynamically create instances of the associated component class.
BOOL bMultiInstance	This flag indicates whether the component class supports multiple instances within the same housing. If it is FALSE, which is the MFC default, the housing supports multiple instances. If this flag is TRUE, multiple instances of the housing will be launched for each new instance of a component class.
LPCSTR lpszProgID	The ProgID for the component class. The ProgID is a readable string that makes it easier for client applications to create instances of the component object.

You won't use the COleObjectFactory constructor directly, because it is implemented by the IMPLEMENT_OLECREATE() macro, which is used to provide a class factory within your component class.

MFC provides a global class factory in that all the COM objects within a housing use the same method to expose their IClassFactory interface to client applications. This is done by using a set of (guess what?) MFC macros. So, for starters, we need to remove the ExpClassFactory declaration in **EXPSVR.H** and the implementation in **EXPSVR.CPP**. In its place we'll add the following code to the Expression class declaration in **EXPSVR.H**:

```
class Expression : public CCmdTarget
{
    DECLARE_DYNCREATE( Expression )
...
};

// macro definitions from AFX.H
#define DECLARE_DYNCREATE(class_name) \
        DECLARE_DYNAMIC(class_name) \
        static CObject* PASCAL CreateObject();
#define DECLARE_DYNAMIC(class_name) \
public: \
    static AFX_DATA CRuntimeClass class##class_name; \
    virtual CRuntimeClass* GetRuntimeClass() const; \
```

The DECLARE_DYNCREATE(classname) macro provides MFC with a way to dynamically create classes whenever required during the execution of your application. The macro also includes the DECLARE_DYNAMIC(classname) macro. This macro also provides certain dynamic capabilities that all classes derived from CObject inherit. In particular it provides a nonstandard run-time type information (RTTI) capability within MFC.

What is RTTI? It's a new addition to the proposed C++ standard that allows C++ developers to easily determine the type of an object at run time. For example, all MFC classes that derived from `CObject` and include `DECLARE_DYNAMIC()` can use the `IsKindOf` function of `CObject`. This allows a developer to determine at run time the identity of a class instance. For example, dynamic classes derived from `CObject` can do something like this:

```
Expression* pExp = new Expression;
...
// At run time check the explicit type of the class
if ( pExp->IsKindOf( RUNTIME_CLASS( Expression ) ))
   // do something
```

The preferred and object-oriented method would be to use the polymorphism capabilities of C++ so that explicit checking isn't required. That's why RTTI is debatable. RTTI is currently part of the draft C++ standard. It is expected to be approved and is currently implemented in most commercial compilers, including Visual C++.

After we ensure that a class instance can be created by MFC, we need to declare a class factory for our component. We use the `DECLARE_OLECREATE( classname )` macro, adding the declaration after `DECLARE_DYNCREATE()` in **EXPSVR.H**.

```
// Expsvr.h
...
class Expression : public CCmdTarget
{
   DECLARE_DYNCREATE( Expression )
   DECLARE_OLECREATE( Expression )
...
};

// macro definition from AFXDISP.H
#define DECLARE_OLECREATE(class_name) \
protected: \
  static AFX_DATA COleObjectFactory factory; \
  static AFX_DATA const GUID AFX_CDECL guid; \
```

The `DECLARE_OLECREATE` macro declares two static members within the `Expression` class: a `COleObjectFactory` variable (`factory`) and a `GUID` variable called (appropriately) `guid`. Because these variables are declared static, there is only one instance and they exist outside the scope of any particular `Expression` instance.

That completes the modifications to **EXPSVR.H**. We now need to define the static variables within **EXPSVR.CPP**.

As you're probably beginning to realize, MFC uses various macros to declare static members of your classes. You use the DECLARE macros in the declaration of your class in the **.H** file, and you must also define or initialize the members using MFC's IMPLEMENT macros in your **.CPP** file.

```
// Expsvr.cpp

#include "stdafx.h"
#include "expsvr.h"

IMPLEMENT_DYNCREATE( Expression, CCmdTarget )

// macro definition from AFXDISP.H
#define IMPLEMENT_DYNCREATE(class_name, base_class_name) \
  void PASCAL class_name::Construct(void* p) \
      { new(p) class_name; } \
  _IMPLEMENT_RUNTIMECLASS(class_name, base_class_name, 0xFFFF, \
      class_name::Construct)
```

The IMPLEMENT_DYNCREATE(class, baseclass) macro initializes the run-time class information needed for dynamic construction. You can see how the Construct method is implemented in the macro expansion using the new operator. Next, we initialize our COleObjectFactory variable. Here's the use of the COleObjectFactory constructor:

```
#define IMPLEMENT_OLECREATE(class_name, external_name, l, w1, w2, \
                            b1, b2, b3, b4, b5, b6, b7, b8) \
  AFX_DATADEF COleObjectFactory class_name::factory(class_name::guid, \
    RUNTIME_CLASS(class_name), FALSE, _T(external_name)); \
    const AFX_DATADEF GUID AFX_CDECL class_name::guid = \
    { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } }; \
```

The IMPLEMENT_OLECREATE macro takes three parameters. The first one is the class name Expression. This identifies the class whose class factory object we're going to initialize. The second parameter is the ProgID for the class. You remember ProgIDs. They provide a readable identifier for accessing the CLSID of a component object. The third parameter is the CLSID broken up into 11 long, word, and byte chunks. I've changed the CLSID slightly from the one that we used in Chapter 4, so we can have distinct Chapter 4 and Chapter 5 versions of the Expression component and our client application can access either one. The first parameter of the GUID assignment is now 0xA988BD42 instead of the 0xA988BD40 in Chapter 4.

```
// Expsvr.cpp

IMPLEMENT_OLECREATE( Expression, "Chap5.Expression.1", 0xA988BD42,
                     0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30 )
//AFX_DATADEF COleObjectFactory Expression::factory( Expression::guid,
//            RUNTIME_CLASS( Expression ),
//            FALSE,
```

```
//            _T( "Chap5.Expression.1" ));
//const AFX_DATADEF GUID AFX_CDECL Expression::guid = { 0xA988BD42, 0x9F1A, //0x11CE, 0x8B, 0x9F,
0x10, 0x00, 0x5A, 0xFB, 0x7D, 0x30 };
```

COleObjectFactory::Register

The `Register` method registers a particular class factory with the COM environment so that client applications can access it. `Register` calls the COM API function `CoRegisterClassObject` for local-server implementations. Run-time registration isn't necessary for in-proc servers, but MFC still requires a call to the `Register` function; it sets an internal variable, `m_dwRegister`, that indicates the factory is registered. This requirement also provides consistency, from the developer's viewpoint, between local-server and in-process implementations (you always call it).

COleObjectFactory::RegisterAll

The `RegisterAll` method is similar to the `Register` function except that it registers all known class factories within an MFC application. Again, this is required for both local-server and in-process server implementations. `RegisterAll` should be called as soon as possible after an application is launched, usually in the `InitInstance` method of your `CWinApp`-derived application class.

To do this in our converted server application, we add a call to `RegisterAll` in our `InitInstance`. In Chapter 4, we didn't need to do anything during startup, so we didn't override `InitInstance`, but we need to now. To do that, go into ClassWizard (**Ctrl-W**), select the **Message Map** tab, and double-click on the **InitInstance Message** to add the function to the `CServerApp` class. Then add the following highlighted code:

```
//
// server.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "server.h"
...

BOOL CServerApp::InitInstance()
{
   // TODO: Add your specialized code here and/or call the base class
   COleObjectFactory::RegisterAll();

    return CWinApp::InitInstance();
}
```

Now when our server is initially loaded, all its contained class factories will be registered with COM. In the case of an in-process server, this call initializes MFC's internal variables, because it does not need to directly register with COM.

COleObjectFactory::Revoke and RevokeAll

Local servers that register their class factories with the COM environment are required to `Revoke` them as the application terminates. MFC handles this automatically for your application by calling `RevokeAll`, which in turn calls `Revoke` for each class factory that is registered. For in-process servers, this method does nothing.

COleObjectFactory::UpdateRegistry and UpdateRegistryAll

The `Register` and `Revoke` methods involve the dynamic updating of the COM running object table. `UpdateRegistry` and `UpdateRegistryAll` allow your application to easily update the Windows Registry. By using these functions, you don't need to build **.REG** files to distribute with your applications. MFC makes it easy to populate the Registry with all the pertinent component information. Depending on your component housing, you must use different techniques of performing this automatic updating. Local servers should call `UpdateRegistryAll` whenever they are executed. In-process servers must export the function `DllRegisterServer`. Let's update our server to use this new capability. Our Chapter 4 server required the use of a **.REG** file to register our CLSID and ProgID in the Windows Registry. If we implement and export the function `DllRegisterServer` and call `UpdateRegistryAll`, the component registration can be handled by a standard utility such as **REGSVR32.EXE**. Here's the implementation that we add to **SERVER.CPP**, and the new export for the **.DEF** file:

```
// Add this to SERVER.CPP
// by exporting DllRegisterServer, you can use regsvr32.exe
STDAPI DllRegisterServer(void)
{
   COleObjectFactory::UpdateRegistryAll();
   return ( S_OK );
}
```

```
; Server.def : Declares the module parameters for the DLL.
LIBRARY      "SERVER"
DESCRIPTION  'SERVER Windows Dynamic Link Library'

EXPORTS
       ; Explicit exports can go here
            DllGetClassObject       @2
            DllCanUnloadNow         @3
            DllRegisterServer       @4
```

If your application is an in-process server, you can use **REGSVR32.EXE** like this:

```
c:>regsvr32 server.dll
```

REGSVR32.EXE is a simple application (supplied with VC++ and the ActiveX SDK) that calls the DLL's `DllRegisterServer` entry point. The DLL does the work of registering itself in the Registry. You can also register your new server by using the Visual C++ **Tools/Register Control** menu item. The preceding command calls REGSVR32 with the project's DLL as a parameter. It's a quick way to register in-process servers. As you'll see in a later chapter, ActiveX controls are really just in-process servers with a number of standard interfaces.


NOTE

REGSVR32 isn't a very complicated program. If you're having problems registering a component housed within a DLL and you want to really understand what REGSVR32 is doing, here's some code that may help.

```
//
// RegisterServer takes as a parameter the
// explicit path and filename of the OLE
// server that you want to register.
// E.g., c:\winnt\system32\clock.ocx
// This function loads the DLL/OCX and calls
// the DllRegisterServer function.
//
DWORD RegisterServer( char* szPath )
{
   HINSTANCE hInstance = ::LoadLibrary( szPath );
   if ( 0 == hInstance )
   {
      return ::GetLastError();
   }

   typedef void (FAR PASCAL *REGSERVER)(void);
   REGSERVER RegServer = (REGSERVER)
         ::GetProcAddress( hInstance, _T( "DllRegisterServer" ));

   if ( 0 == RegServer )
   {
      return ::GetLastError();
   }

   RegServer();

   ::FreeLibrary( hInstance );
```

```
    return 0;
}
```

If a server application is implemented as a local server, COM standards recommend that the local server update the Registry every time the application is run. This is easy to do using MFC. You would do something like the following. (MFC will add this function automatically when building a local server that has COM support, but we're doing it the hard way right now.)

```
BOOL CMFCApp::InitInstance()
{
...
    // When a server application is launched stand-alone, it is a good idea
    //  to update the system registry in case it has been damaged.
    COleObjectFactory::UpdateRegistryAll();
}
```

We'll discuss this in more detail in Chapter 6 when we build a local-server application that uses Automation.

MFC COM Helper Functions

MFC has certain "helper" functions that aren't members of any class but provide COM-specific functionality for your application's use. We need to use two of these functions to complete our server application.

`AfxDllGetClassObject` provides the implementation of COM's `DllGetClassObject` function. MFC internally maintains a pointer to a list of all class factories in your application. As we saw earlier, `COleObjectFactory` contains a member variable, `m_pNextFactory`, that implements a linked list of your `COleObjectFactory` instances. MFC maintains a pointer to the beginning of this list. `AfxDllGetClassObject` searches this list for the requested CLSID, creates an instance of the internal class factory, and returns a pointer to the requested Interface ID (IID), just as we did in Chapter 4. We need to change our implementation of `DllGetClassObject` in **SERVER.CPP** as follows:

```
//
// Server.cpp
//
...
STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid, void** ppv )
{
   //HRESULT             hr;
   //ExpClassFactory    *pCF;
   // pCF = 0;
   //if ( rclsid != CLSID_Expression )
   //   return( E_FAIL );
```

```
    //pExpression = new ExpClassFactory();
    //if ( pCF == NULL )
    //    return( E_OUTOFMEMORY );
    //hr = pCF->QueryInterface( riid, ppv );
    //if ( FAILED( hr ) )
    //    delete pExpression
    //return( hr );

    // Let MFC do the work
    return AfxDllGetClassObject( rclsid, riid, ppv );
}
```

`AfxDllCanUnloadNow` is the other MFC helper function that we need to use in our application. Because we now let MFC keep track of the reference counts on our component objects, we don't have the knowledge to implement the COM `DllCanUnloadNow` function. So we need to change our implementation of `DllCanUnloadNow` in **SERVER.CPP**.

```
STDAPI DllCanUnloadNow(void)
{
    //if ( g_dwObjs || g_dwLocks )
    //    return( S_FALSE );
    //else
    //    return( S_OK );

    // Let MFC do the work
    return AfxDllCanUnloadNow();
}
```

After making all the changes to the server, build it and test it using the client application we developed earlier in the chapter. You should be able to access both the Chapter 4 and Chapter 5 server programs.

A Recap of the Example Applications

We've been working with our heads down in this chapter, and I think we need an overview of what we've done. First, we added a modal dialog box to our Chapter 4 client application that lets us choose a specific ProgID on startup. We then removed the COM initialization functions and replaced them with a call to MFC's `AfxOleInit` function. The rest of the COM/ActiveX code we left alone, because there wasn't a direct MFC replacement.

The server application required more changes. We first derived our component class, `Expression`, from `CCmdTarget` instead of `IExpression`. In the process, we learned quite a bit about the mother of all ActiveX classes in MFC: `CCmdTarget`. It provides the `IUnknown` interface plus an interface mapping mech-

anism that makes it easy to provide `QueryInterface`. `CCmdTarget` uses nested classes and encapsulates the reference counting of our components classes, so we don't have to.

Next, we explored the MFC `COleObjectFactory` class and how it provides class factories for our component classes. `COleObjectFactory` automatically keeps track of all the class factories and provides a function, `AfxDllGetClassObject`, to use to expose them to client processes. `COleObjectFactory` provides other features, including the ability to automatically update the Registry with a component's **.REG** information. Throughout this discussion, we learned a great deal about how MFC provides COM/ActiveX support.

Summary

In this chapter, we have looked at how the MFC libraries implement COM and ActiveX support. In particular, we looked at various MFC helper functions—`AfxOleInit`, `AfxDllGetClassObject`, `AfxDllCanUnloadNow`, and so on—that make implementing ActiveX applications easier. We also investigated two important MFC classes that provide COM functionality: `COleObjectFactory` and `CCmdTarget`. In doing so we converted our client and server examples to use MFC's COM support. From this point on, we will use MFC to develop all of our software components.

These first five chapter have provided a solid base for building ActiveX software using the MFC libraries. In the next chapter, we'll use Automation. Automation and MFC make it very easy to wrap existing C++ functionality so that it can be accessed by various other languages.

Chapter 6

Automation

Now it's time to start using this wonderful technology called ActiveX. In this chapter, we'll investigate Automation, an ActiveX technology that is a boon for developers. Automation makes it easy to wrap existing software modules for use outside C and C++ and provides a powerful mechanism to expose an application's functionality to programmatic users. This makes the application an ActiveX component.

What Is ActiveX Automation?

Automation is many things, but it provides two broad techniques that we will explore in this chapter. The first one, which we've been working toward since the first chapter, is the ability to wrap C++ classes and expose their functionality to other, non-C++ language users. This feature alone provides a significant new ability in the development of Windows software.

We will also explore the other technique: using Automation to drive another application or process. This was one of Microsoft's original uses for automation, and it makes it easy to build a generic macro language that will work with many different application products. Microsoft's Visual Basic for Applications is an example of using automation for this purpose.

The ability to expose an application's functionality in a standard way makes an application a software component. If you need the services of a good word processor in your application, you can purchase a word processor that exposes its capabilities via Automation and use it within your application. Microsoft Word, Microsoft Project, and Internet Explorer provide access to nearly all their capabilities through automation.

Automation is one of the most important technologies in ActiveX. It allows for a truly dynamic method whereby two applications can interoperate. In our previous examples, the user or client application had to know just a little about the `Expression` component—namely, the declaration of the `IExpression` interface.

Automation removes this last constraint and provides dynamic determination and invocation of a component's interface methods. Automation provides a true *late-binding* mechanism not only within a single language but also between languages and between local or remote processes. This late-binding mechanism is implemented by adding a level of indirection to a component's Vtable pointer implementation. I promised that in these later chapters we wouldn't spend as much time covering the low-level details, and I'm going to keep my promise, but occasionally we will drop down to investigate how MFC implements various features.

Automation Controllers

In earlier chapters we described the user of a component as the *client*. Clients that use and interact with Automation components are called *automation controllers*. This term implies that the client is controlling the component, and in many ways this is correct. Before the specification for ActiveX controls was complete, Automation using the `IDispatch` interface allowed only one-way interaction between client and server applications. This one-way communication constraint explains the use of the term *controller*.

The ActiveX controls specification added the concept (and implementation) of an "outgoing" interface where events that occur within a component can be communicated to the controlling component. In this realm, the concept of who is controlling the interaction blurs, as both components communicate as equals. With the addition of ActiveX events, the relationship between controller and server becomes peer-to-peer instead of the earlier master-slave.

In this chapter, we will use the master-slave type of automation. We will use the `IDispatch` interface to build an application that also contains our `Expression` component class. At first the idea of one-way interoperation may seem constrictive, but in reality that is how we typically develop programs. A normal function call in C++ is synchronous and occurs with one-way communication. You call the function, and your program waits until it returns. The only communication coming back is the value of the return code and any parameters that were passed by reference.

A C++ example of two-way communication is the use of a callback Function, which allows an event to be communicated "out of band" of the normal flow of your application. A callback function takes as a parameter a pointer to a function that you write. As the function executes, it "calls back" into your function with specific information. This arrangement also provides a notification mechanism. A simple example of a callback is the C run-time function `qsort`. For the typical use of an `IDispatch` interface, when only one process is in control of the interaction, the term *controller* describes the use of the interface quite effectively.

The archetypal Automation controller is Visual Basic, which provides a robust controller environment. Visual Basic isn't the only controller available, nor is its use required. The MFC libraries make it easy to develop an application that acts as a controller and drives other applications, although MFC doesn't currently provide the dynamic invocation capabilities of Visual Basic unless you write the code yourself. We'll take a look at what is required in all these circumstances.


NOTE

Actually, the event mechanism is implemented using a reverse (or incoming) dispatch interface. However, I don't want to add too much complexity here as we begin discussing the `IDispatch` interface. We will explore the concept of an ActiveX event in later chapters.

Visual versus Nonvisual Components

The `Expression` class is an example of a nonvisual component. The services it provides have little to do with any GUI aspect of an application. We've been using its capabilities within a Windows entry field, but that need not be the case. The class could be used internally by any number of applications that need the ability to parse and evaluate algebraic expressions. Nonvisual components are usually best implemented as Automation servers. Visual components are probably best implemented as ActiveX controls. There are exceptions, of course, but this is a good general rule to follow.

The IDispatch Interface

Automation is based on an ActiveX interface called `IDispatch`, which provides the ability to dynamically invoke methods on a component within a server. This arrangement differs slightly from the earlier COM custom interface technique of our `Expression` component. When you're using custom interface implementations, the client application requires the `IExpression` declaration to use the component. The `IExpression` declaration (its Vtable structure) is used at compile time to define the Vtable structure of the `Expression` interface. This implements *early binding* of the component's interface. If a new method is added to `IExpression`, the client application must be recompiled to use the new capability.

A server component that implements its methods using `IDispatch` instead of using a custom interface provides a number of additional capabilities. By using `IDispatch`, the interaction between the client and server applications uses late binding of the component's interface methods. The client doesn't require compile-time declarations of the server's component interface. This makes it easy for a server component to change its interface (even at run time!) without requiring the client application to be recompiled or relinked. Of course, any changes to the interface methods should be communicated to the client application so that it can take advantage of the new features. To understand how `IDispatch` provides language-independent late binding, we need to look at its implementation.

Another significant feature added by using `IDispatch` is that of standard marshaling. The default Windows COM implementation contains a number of data types that can be used by components that use `IDispatch`. Intrinsic support for these data types makes it easy to build components that work across local and remote processes.

Most COM interfaces are like the `IExpression` example. They provide a structure that requires a rigid implementation of an abstract class. COM defines the abstract class (interface), but the application developer must provide a unique implementation of that abstract class. `IDispatch` is a little different, because it adds a level of indirection to the Vtable-style interfaces that we've studied so far. One term for this new interface type is *dispinterface*, for dispatch interface. That term succinctly describes how `IDispatch` differs from the standard Vtable implementation. The client does not access a component's functionality through the Vtable pointer, as we did with `IExpression`. Instead, the client first looks up the function, provides a key to the desired function, and finally invokes or dispatches the function (see Figure 6.1).

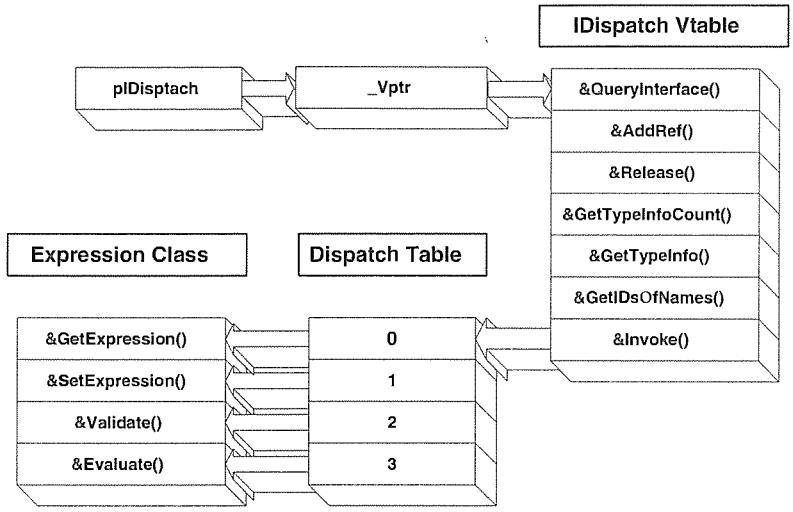


Figure 6.1 `IDispatch` Vtable and dispatch table.

As you can see from Figure 6.1, there's a bit more going on here with the `IDispatch` interface than there was with a custom COM interface in the previous chapters. The client still gets a Vtable pointer, but now that the Vtable doesn't have direct access to the `Expression` methods, the request must first go through the `IDispatch::Invoke` method. `Invoke` contains a parameter that maps the method call to a specific entry in the dispatch map. This additional level of indirection is what provides for late binding to the component's methods. Let's take a quick look at `IDispatch`'s methods.


NOTE

Remember, the server component implements the `IDispatch` methods, and the controller (or client) calls that implementation through the `IDispatch` server's pointer. Our next example will describe this process in detail.

Invoke

`Invoke` provides most of the functionality of the `IDispatch` interface. It takes eight parameters, the most important of which is the DISPID. The DISPID is mapped to a specific offset within the dispatch table and determines which component class method is invoked. Table 6.1 lists the parameters of `Invoke`. We aren't going to study them in detail yet. We just need to understand that the controller calls `Invoke` with these parameters instead of calling a component's methods directly through a Vtable implementation.

Table 6.1 `IDispatch::Invoke` Parameters

Parameter	Purpose
`DISPID dispidMember`	The numeric ID of the specific member within the dispatch table that the controller wants to access.
`REFIID riid`	Reserved; must be `IID_NULL`.
`LCID lcid`	A locale ID that indicates the locale if the component supports National Language Support.
`USHORT wFlags`	The type of `Invoke` call. This flag indicates which type of operation it wants to perform (access a property, invoke a method, and so on). It must be one of the following: `DISPATCH_METHOD`, `DISPATCH_PROPERTYGET`, `DISPATCH_PROPERTYPUT`, and `DISPATCH_PROPERTYPUTREF`.
`DISPPARMS FAR* pdispparms`	The parameter structure that contains an array of parameters for the function.
`VARIANT FAR* pvarResult`	Where to store the result of the call. `NULL` if the method returns `void`. Ignored on `PROPERTY_PUT`s.
`EXCEPINFO FAR* pexcepinfo`	Storage for any exception information if an exception occurs.
`UINT FAR* puArgErr`	Indicates the array element that contains an invalid type when a type mismatch occurs.

As you can see, the controller must provide a great deal of information about the function call to the server's component object. This is one of the drawbacks of late binding. Because the controller doesn't have this information at compile time, it must provide it at run time to use the component's methods. This requirement adds overhead to every method call made on a component and directly affects performance.

Late binding is great, but how can the controller obtain enough information about the component's methods to use the `Invoke` method? The three other methods of `IDispatch` help in this regard.

GetIDsOfNames

`GetIDsOfNames` provides a facility for the controller to map the textual automation server property or method name, such as `GetExpression`, to its numeric DISPID. The DISPID can then be used with the `Invoke` function to access the property or method of the server. Calling `GetIDsOfNames` before every call to `Invoke` can get rather expensive. It is recommended that the controller call `GetIDsOfNames` only once for each property or method and cache the returned DISPID. This technique will speed the interaction. Table 6.2 lists the parameters of `GetIDsOfNames`.

Table 6.2 `IDispatch::GetIDsOfNames` Parameters

Parameter	Purpose
`REFIID riid`	Reserved, must be `IID_NULL`.
`OLECHAR** rgszNames`	An array of member names for which the caller wants the corresponding DISPID. The first element is the name of the member property or method, and the subsequent names are for named parameters.
`unsigned int cNames`	The number of member names in the array.
`LCID lcid`	A locale ID that indicates the locale if the component supports National Language Support.
`DISPID* rgdispid`	An array of DISPIDs. The first element returned is the DISPID of the property or method, and the subsequent elements contain DISPIDs of any named parameters.

GetTypeInfo

Typically, a controller that provides dynamic lookup and calling of automation methods won't have all the type information necessary to populate the `Invoke dispparams` structure. An automation server should call `GetTypeInfoCount` to determine whether the component can provide type information and, if it can, should then call `GetTypeInfo` to get the type information.

For a component to be useful in various development tool implementations, it should provide type information. The type information can be provided by creating an ODL file that describes each property and method along with its return types and parameter types. This file can be compiled into a binary form that makes it easy for client applications to obtain detailed type information for the component. We will discuss this in more detail later in this chapter.

MFC's current implementation of Automation support (aside from ActiveX controls) does not provide type information and returns `E_NOTIMPL`.

GetTypeInfoCount

`GetTypeInfoCount` is used by the controller to determine whether the component object contains type information that can be provided to the controller. Setting the passed-in parameter to 1 indicates that type information is available, and zero indicates that no type information is available. In either case, the method should return `S_OK`. Even without type information, the controller can use the other `IDispatch` methods.

Automation Properties and Methods

The concept of an Automation *property* is very similar to the concept of a C++ member variable. To the user of an Automation property, it will behave and act like a variable in the implementing language. Property

values can easily be assigned and retrieved. You would normally use a property if there is an internal C++ variable that it can be associated with. Here's how an Automation property is used within Visual Basic:

```
Dim btn as object
Dim szCaption as String
' Create an instance of a "Button" class
Set btn = CreateObject( "AutoButton" )
' The button has a Caption property
' Set the caption
btn.Caption = "Cancel"
' Get the caption value
szCaption = btn.Caption
```

As you can see, the syntax for working with the `Caption` property is similar to the syntax you use when you're using an intrinsic data type. In Automation, properties can also have parameters. The parameters typically are used as indexes into an ordered data type. Here's an example of a property with parameters:

```
Dim lbx as object
Dim szString as String
' Create an instance of a "Listbox" class
Set lbx = CreateObject( "AutoListbox" )
' The listbox has a "Text" property
' Fill the list box
For i = 0 to 9
   ' Set the item text
   lbx.Text(i) = "Item " + i
Next
' Get the 5th listbox item text
szString = lbx.Text(4)
```

Automation methods are analogous to C++ class methods. They perform some action within the component when invoked by a controller. The addition of parameters somewhat blurs how Automation properties are different from Automation methods. The primary syntactic difference is that you can't assign a value to an Automation method.

```
Dim lbx as object
Dim szString as String
' Create an instance of a "Listbox" class
Set lbx = CreateObject( "AutoListbox" )
' Empty the Listbox with the clear method
lbx.Clear
' The following makes no sense, and generates an error
lbx.Clear = 10
```

Automation Data Types

Automation provides standard marshaling of parameters and return values across process boundaries. This arrangement makes it easy to implement your components within a local server. To provide marshaling for Automation, COM defines and limits the data types that can be used. I've outlined these data types in Table 6.3. Not all of these types are supported by Visual C++'s ClassWizard, but most of them are.

Table 6.3 Automation-Compatible Data Types

Type	Description
`BSTR`	A binary string that stores the length of the string in the first two or four bytes of the structure and stores the actual string directly following.
`Byte`, `int`, `short`, `long`, `BOOL`, `float`, `double`, `char*`	Intrinsic types.
`CY`	A currency value stored in eight-byte integer. Used for fixed-point representation of a number. The integer represents the value multiplied by 10,000.
`DATE`	A date stored in a `double`, where January 1, 1900, is 2.0, January 2 is 3.0, and so on. Functions are provided by ActiveX to help in the manipulation of the date type.
`SCODE`	A 32-bit error code.
`LPUNKNOWN`	A pointer to an `IUnknown` interface.
`LPDISPATCH`	A pointer to an `IDispatch` interface.
Safe Array	An array of one of the preceding data types, including an array of `Variants`.
`VARIANT`	A union containing any one of the preceding types.

In certain cases, the supported types in Table 6.3 can be passed by reference, thus allowing the callee to modify the parameter value. In many cases, the values are stored in Automation's `VARIANT` data type structure. The `VARIANT` data type allows the controller and the server to communicate.

The VARIANT Data Type

The `VARIANT` data type is the most important Automation data type. It provides a convenient and effective technique for passing parameters between COM-based components. Also, because of its implementation, a variant can be used to overload Automation methods. Sometimes it's easiest to describe something by showing the code. Here's a condensed definition of the `VARIANT` structure.

```
struct tagVARIANT
{
   VARTYPE vt;
   union
   {
```

```
        LONG lVal;
        BYTE bVal;
        SHORT iVal;
        FLOAT fltVal;
        DOUBLE dblVal;
        VARIANT_BOOL boolVal;
        SCODE scode;
        CY cyVal;
        DATE date;
        BSTR bstrVal;
        IUnknown *punkVal;
        IDispatch *pdispVal;
        SAFEARRAY *parray;
        BYTE *pbVal;
        SHORT *piVal;
        LONG *plVal;
        FLOAT *pfltVal;
        DOUBLE *pdblVal;
        SCODE *pscode;
        CY *pcyVal;
        DATE *pdate;
        BSTR *pbstrVal;
        IUnknown **ppunkVal;
        IDispatch **ppdispVal;
        SAFEARRAY **pparray;
        VARIANT *pvarVal;
        PVOID byref;
        CHAR cVal;
    }
};

typedef tagVARIANT VARIANT;
typedef VARIANT VARIANTARG;
```

A VARIANT is just a big union of different types and an identifier indicating which type is within the union. The vt member indicates the actual data type, thereby allowing the receiver to extract the data from the appropriate union element. Several COM APIs and macros make working with variants easier. Table 6.4 shows the most useful API calls. For a look at the macros, peruse the **OLEAUTO.H** file in the \MSDEV\INCLUDE directory. MFC also provides a COleVariant class that makes using the variant data type a little easier.

Table 6.4 Useful Variant-Based APIs

Function	Purpose
`VariantInit( VARIANT* )`	Initializes a variant to `VT_EMPTY`.
`VariantClear( VARIANT* )`	Initializes a variant by first freeing any memory used within the variant. This is useful when client-side applications need to clear a variant passed from a server.
`VariantChangeType( ... )`	Coerces a variant from one type to another. For example, this method will change a `long` (I4) to a `BOOL` and even an `IDispatch` pointer to a `BSTR`.
`VariantCopy( VARIANT*, VARIANT* )`	Makes a copy of a variant and frees any existing memory in the destination before making the copy.

I mentioned that variants can be used to overload Automation methods. Here's an example in a Visual Basic–like language:

```
objCollection.Add( "Jessica" )
' objName is an automation object
objCollection.Add( objName )
' txtControl is an edit control
objCollection.Add( ByRef txtControl.Text )
```

The `Add` Automation method takes one parameter, which is a variant. Because the method takes a variant, the method user can pass different variant-supported data types to the method. The preceding example first passes a `BSTR`, and then an Automation object, which is an `IDispatch` pointer, and finally a `BSTR` by reference. The `Add` method is implemented to handle all these types, thus making the method easy to use. This concept gives the server developers access to the powerful object-oriented technique of overloading.

The Safe Array

The safe array data type and its associated APIs let you move arrays of Automation-compatible types between local and remote processes. A number of safe array functions are provided by COM. For more information, take a look at the on-line documentation or the *OLE Programmer's Reference Guide, Volume Two.*

A Native IDispatch-Based Component

Now it's time for an example to solidify our understanding of the `IDispatch` interface. We've used the `Expression` class to build a COM-based component with a custom interface, using native API calls and then using MFC. Now let's implement the `Expression` class using native Automation, implementing both the server and client sides of the `IDispatch` interface. After that, we'll do it again using MFC. First, let's build the server.

The Expression Class as an ActiveX Component

In Chapter 4, we turned the `Expression` class into a COM component. We used a COM custom interface to expose the `Expression` functionality. Now we'll expose its functionality via a standard ActiveX interface: `IDispatch`. We've discussed the benefits of using `IDispatch`, so we'll focus primarily on its implementation. The changes are based on using the **EXPSVR.H** and **EXPSVR.CPP** files from the Chapter 4 SERVER project. As you'll see, only slight modifications are necessary to convert a component that uses a custom COM interface into one that uses a standard ActiveX interface. Later, when we discuss dual interfaces, you'll see how to build a component that exposes both interfaces, providing a component user with tremendous implementation flexibility.

Building the Visual C++ Project

We'll house our new component in a DLL, making it a COM in-process server. In the MFC-based version of the project, we will implement it within an EXE housing. By implementing a component within an EXE, you gain cross-process and cross-network capabilities without any additional work. By using `IDispatch`, we automatically get cross-process marshaling. That's one of its major benefits.

We've used Visual C++ quite a bit by now, so I'll just hit the high points of building a project. Start VC++ and perform the following steps to create our initial Automation server:

1. Create a new Project Workspace and specify an **MFC AppWizard (dll)** project.
2. Name the project **Server** and take the provided defaults. Select **Regular DLL using Shared MFC DLL**, but no **OLE Automation** or **Windows Sockets** support.
3. After you click the **Create** button, your screen should be similar to Figure 6.2.

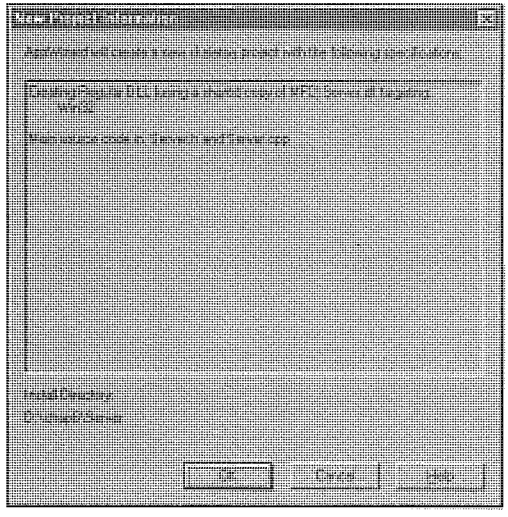


Figure 6.2 SERVER project information dialog box.

Copy the **EXPSVR.H** and **EXPSVR.CPP** files from your Chapter 4 SERVER project or from the accompanying CD-ROM. What follows are changes required to convert the `Expression` component from implementing a custom `IExpression` interface to use `IDispatch` instead.

Updating SERVER.H and SERVER.CPP

The initial **SERVER.CPP** file does not have the COM support that we added in Chapter 4. For a quick review, here's what you need to add to **SERVER.CPP**:

```
//
// Server.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "server.h"

#include <initguid.h>
#include "expsvr.h"

DWORD       g_dwObjs = 0;
DWORD       g_dwLocks = 0;
...
```

We need to include **EXPSVR.H** to define the `Expression` class factory and the `Expression` class, because `DllGetClassObject`, the COM function that exposes our component, needs them. The two global variables keep track of the number of instantiated objects within the housing and the number of `LockServer` calls made by the client. Here are the changes for **SERVER.CPP**:

```
STDAPI DllGetClassObject( REFCLSID rclsid, REFIID riid, void** ppv )
{
   HRESULT              hr;
   ExpClassFactory     *pCF;

   pCF = 0;

   // Make sure the CLSID is for our Expression component
   if ( rclsid != CLSID_Expression )
      return( E_FAIL );

   pCF = new ExpClassFactory;

   if ( pCF == NULL )
      return( E_OUTOFMEMORY );

   hr = pCF->QueryInterface( riid, ppv );
```

```
    // Check for failure of QueryInterface
    if ( FAILED( hr ) )
    {
        delete pCF;
        pCF = NULL;
    }

    return hr;
}

STDAPI DllCanUnloadNow(void)
{
    if ( g_dwObjs || g_dwLocks )
        return( S_FALSE );
    else
        return( S_OK );
}
```

`DllGetClassObject` and `DllCanUnloadNow` are part of Chapter 4's SERVER application. COM requires that a DLL housing support these two functions. They allow the client (through COM) to create an instance of our component. Also, don't forget to export these functions in you **.DEF** file:

```
; Server.def : Declares the module parameters for the DLL.

LIBRARY      "SERVER"
DESCRIPTION  'SERVER Windows Dynamic Link Library'
EXPORTS
    ; Explicit exports can go here
    DllGetClassObject
    DllCanUnloadNow
```

Modifying EXPSVR.H and EXPSVR.CPP

That was pretty easy. Now let's do the difficult part: implement `IDispatch` within the `Expression` class. Here are the changes to make to **EXPSVR.H**:

```
//
// ExpSvr.h - Expression header
//

// access to the global variables in SERVER.CPP
extern DWORD g_dwObjs;
extern DWORD g_dwLocks;
```

```
DEFINE_GUID( CLSID_Expression,
             0xA988BD40,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);
//DEFINE_GUID( IID_IExpression,
//              0xA988BD41,0x9F1A,0x11CE,0x8B,0x9F,0x10,0x00,0x5A,0xFB,0x7D,0x30);

const DISPID_GETEXPRESSION = 1001;

const DISPID_SETEXPRESSION = 1002;
const DISPID_EVAL = 1003;
const DISPID_VALIDATE = 1004;
```

We don't need the GUID for the custom interface; we need only a CLSID to identify the Automation object. Next, we've added some constants that we'll use later as the DISPIDs for the `Expression` object. The Automation specification already has a `DISPID_EVALUATE` definition so we use `DISPID_EVAL` instead.

```
//class IExpression : public IUnknown
//{
//public:
//    STDMETHOD_(BSTR,GetExpression()) PURE;
//    STDMETHOD_(void, SetExpression(BSTR, BOOL)) PURE;
//    STDMETHOD_(BOOL, Validate())      PURE;
//    STDMETHOD_(long, Evaluate())      PURE;
//};

class Expression : public IDispatch // public IExpression
{
protected:
...
```

We're now implementing the `IDispatch` interface so we can comment out the `IExpression` declaration and derive our `Expression` class interface from `IDispatch`. Like all other COM interfaces, `IDispatch` derives from `IUnknown`. Because we derive `Expression` from `IDispatch`, we must declare and implement seven public methods (our interface) in the `Expression` class: `QueryInterface`, `AddRef`, `Release`, `GetTypeInfoCount`, `GetTypeInfo`, `GetIDsOfNames`, and `Invoke`. Here are the changes:

```
public:
   // IUnknown
   STDMETHOD(QueryInterface( REFIID, void** ));
   STDMETHOD_(ULONG, AddRef());
   STDMETHOD_(ULONG, Release());

   // IDispatch
   STDMETHOD(GetTypeInfoCount( UINT* pctinfo ));
   STDMETHOD(GetTypeInfo( UINT itinfo,
```

```
                              LCID lcid,
                              ITypeInfo** pptinfo ));
    STDMETHOD(GetIDsOfNames( REFIID riid,
                              OLECHAR** rgszNames,
                              UINT cNames,
                              LCID lcid,
                              DISPID* rgdispid ));
    STDMETHOD(Invoke( DISPID dispid,
                         REFIID riid,
                         LCID lcid,
                         WORD wFlags,
                         DISPPARAMS FAR* pDispParams,
                         VARIANT FAR* pvarResult,
                         EXCEPINFO FAR* pExcepInfo,
                         unsigned int FAR* puArgErr ));

    // IExpression
    //STDMETHOD_(BSTR, GetExpression());
    //STDMETHOD_(void, SetExpression( BSTR, BOOL ));
    //STDMETHOD_(BOOL, Validate());
    //STDMETHOD_(long, Evaluate());

protected:
    // Here's the classes true functionality. It is now
    // exposed only through our new IDispatch methods.
    BSTR      GetExpression();
    void      SetExpression( BSTR bstr, BOOL bInFix );
    BOOL      Validate();
    long      Evaluate();
```

The `IUnknown` declarations are unchanged, but we now declare the four `IDispatch` methods instead of the custom `IExpression` methods that we used previously. The class must provide its `Expression` functionality, but the methods are not part of its public interface and so are moved to a protected section. That completes the **EXPSVR.H** file. Now let's modify **EXPSVR.CPP**.

```
STDMETHODIMP Expression::QueryInterface( REFIID riid, void** ppv )
{
   *ppv = NULL;

//    if ( riid == IID_IUnknown || IID_IExpression )
    if ( riid == IID_IUnknown || riid == IID_IDispatch )
       *ppv = this;
```

```
    if ( *ppv )
    {
        ( (LPUNKNOWN)*ppv )->AddRef();
        return( S_OK );
    }
    return (E_NOINTERFACE);
}
```

The first change is in the implementation of `QueryInterface`. Instead of supporting `IExpression`, we now support `IDispatch`. That's easy enough. What else needs to be done?

```
STDMETHODIMP Expression::GetTypeInfoCount( UINT* pctinfo )
{
   // Indicate that we don't expose any type information
   *pctinfo = 0;
   return S_OK;
}
```

Here's our implementation of the `GetTypeInfoCount` method. To simplify the example, we don't support exposing our type information. We indicate this by setting the `pctinfo` parameter to zero and returning `S_OK`.

```
STDMETHODIMP Expression::GetTypeInfo( UINT itinfo,
                                      LCID lcid,
                                      ITypeInfo** pptinfo )
{
   // Not implemented
   if ( pptinfo )
      *pptinfo = 0;
   return( E_NOTIMPL );
}
```

`GetTypeInfo` is just as easy. We set the `pptinfo` parameter to zero and return `E_NOTIMPL` to indicate that type information is not supported. Here comes the essence of Automation.

```
STDMETHODIMP Expression::GetIDsOfNames( REFIID riid,
                                        OLECHAR** rgszNames,
                                        UINT cNames,
                                        LCID lcid,
                                        DISPID* rgdispid )
{
   // To make things simple, we only support 1 name at a time
   if ( cNames > 1 )
```

```
        return( E_INVALIDARG );

    // Convert the member name to regular ANSI
    USES_CONVERSION;
    char* szAnsi = OLE2T( rgszNames[0] );

    // Compare the member name to see if it's one that we have
    // and return the correct DISPID
    if ( strncmp( "GetExpression", szAnsi, 14 ) == 0 )
        rgdispid[0] = DISPID_GETEXPRESSION;
    else if ( strncmp( "SetExpression", szAnsi, 14 ) == 0 )
        rgdispid[0] = DISPID_SETEXPRESSION;
    else if ( strncmp( "Validate", szAnsi, 8 ) == 0 )
        rgdispid[0] = DISPID_VALIDATE;
    else if ( strncmp( "Evaluate", szAnsi, 8 ) == 0 )
        rgdispid[0] = DISPID_EVAL;
    else
        return( DISPID_UNKNOWN );

    // Everything worked!
    return S_OK;
}
```

In Automation the client application need not have any knowledge of a server's methods at compile and link time. As we discussed earlier, Automation enables late binding of methods and their parameters. To provide this capability, we've implemented the `GetIDsOfNames` method.

The client can now call `GetIDsOfNames` at run time to get the specific DISPID for a method within the `Expression` component. We defined our DISPIDs earlier. Our job now is to map the member name to a specific DISPID. This is easy. The `rgszNames` parameter is an array of member names provided by the calling application. The array consists of a method or property name in the first element, optionally followed by named parameter elements. I won't complicate the issue by discussing named parameters, but that's why we are provided an array of member names.

All COM interface calls use native Unicode, so we convert the Unicode string to ANSI to facilitate the comparison. Then we do a simple string comparison. If we get a match, we return the respective DISPID. If we don't get a match, we return the required `DISPID_UNKNOWN` error.

```
STDMETHODIMP Expression::Invoke( DISPID dispid,
                                 REFIID riid,
                                 LCID lcid,
                                 WORD wFlags,
                                 DISPPARAMS FAR* pDispParams,
                                 VARIANT FAR* pvarResult,
```

```
                                EXCEPINFO FAR* pExcepInfo,
                                unsigned int FAR* puArgErr )
{
   switch( dispid )
   {
      case DISPID_GETEXPRESSION:
         if ( pvarResult )
         {
            VariantInit( pvarResult );
            V_VT( pvarResult ) = VT_BSTR;
            V_BSTR( pvarResult ) = GetExpression();
         }
         return S_OK;

      case DISPID_SETEXPRESSION:
         if ( !pDispParams ||
               pDispParams->cArgs != 2 )
            return( DISP_E_BADPARAMCOUNT );

         // We don't support named arguments
         if ( pDispParams->cNamedArgs > 0 )
            return( DISP_E_NONAMEDARGS );

         HRESULT hr;
         // Coerce the variant into the desired type
         // In this case we would like a BSTR
         hr = VariantChangeType(  &(pDispParams->rgvarg[1]),
                                  &(pDispParams->rgvarg[1]),
                                  0,
                                  VT_BSTR );

         // If we can't get a BSTR return type mismatch
         if ( FAILED( hr ))
            return( DISP_E_TYPEMISMATCH );

         // Coerce the variant into the desired type
         // In this case we would like an I4 or long
         hr = VariantChangeType( &(pDispParams->rgvarg[0]),
                                  &(pDispParams->rgvarg[0]),
                                  0,
                                  VT_I4 );
```

```
        if ( FAILED( hr ))
            return( DISP_E_TYPEMISMATCH );

        // Finally call our method with the correct parameters
        // Remember the parameters come in right to left
        SetExpression( pDispParams->rgvarg[1].bstrVal,
                       pDispParams->rgvarg[0].iVal );

        return( S_OK );

    case DISPID_VALIDATE:
        if ( pvarResult )
        {
            VariantInit( pvarResult );
            V_VT( pvarResult ) = VT_I4;
            V_I4( pvarResult ) = Validate();
        }
        else
            Validate();

        return S_OK;

    case DISPID_EVAL:
        if ( pvarResult )
        {
            VariantInit( pvarResult );
            V_VT( pvarResult ) = VT_I4;
            V_I4( pvarResult ) = Evaluate();
        }
        else
            Evaluate();
        return S_OK;

    default:
        return( DISP_E_MEMBERNOTFOUND );
    }
}
```

The `Invoke` method provides most of the functionality of the `IDispatch` interface. First, we determine which member was invoked by the client, as indicated by the specific DISPID. Our simple server doesn't expose any type information, so the only way for the client to get the DISPID is to explicitly call the `GetIDsOfNames` method.

Based on the DISPID passed, we call the appropriate `Expression` method. The only difficulty is in how to interpret and handle all those parameters used in the `Invoke` method. Let's examine each one in detail.

```
case DISPID_GETEXPRESSION:
   if ( pvarResult )
   {
      VariantInit( pvarResult );
      V_VT( pvarResult ) = VT_BSTR;
      V_BSTR( pvarResult ) = GetExpression();
   }
   return S_OK;
```

The `GetExpression` method takes no parameters and returns a `BSTR` containing the current `Expression` value. The sixth parameter of `Invoke` is a `VARIANT` allocated by the caller for the purpose of passing the return value of the call. The client need not provide this parameter, so we check to ensure that it is a nonzero pointer before we do our work. `GetExpression` has no side effects, so there is no need to process the call if the caller does not provide storage to return the result. If a variant is provided, we initialize it, set the variant type, and assign its value with a call to `GetExpression`.

Two of the other `Expression` methods do not take parameters and return a result. `Evaluate` and `Validate` are very similar.

```
case DISPID_VALIDATE:
   if ( pvarResult )
   {
      VariantInit( pvarResult );
      V_VT( pvarResult ) = VT_I4;
      V_I4( pvarResult ) = Validate();
   }
   else
      Validate();
   return S_OK;

case DISPID_EVAL:
   if ( pvarResult )
   {
      VariantInit( pvarResult );
      V_VT( pvarResult ) = VT_I4;
      V_I4( pvarResult ) = Evaluate();
   }
   else
      Evaluate();
   return S_OK;
```

The primary difference between these two implementations is that they must perform some action even if the caller does not provide storage for the result. The two methods can be called in two ways using a controller language such as Visual Basic:

```
nResult = objExpression.Evaluate()
' or
objExpression.Evaluate
```

The first method uses the result and the second one does not. Make sure that you handle all the various conditions when mapping your methods.

```
case DISPID_SETEXPRESSION:

   if ( !pDispParams ||
          pDispParams->cArgs != 2 )
      return( DISP_E_BADPARAMCOUNT );

   // We don't support named arguments
   if ( pDispParams->cNamedArgs > 0 )
      return( DISP_E_NONAMEDARGS );

   HRESULT hr;
   // Coerce the variant into the desired type
   // In this case we would like a BSTR
   hr = VariantChangeType(  &(pDispParams->rgvarg[1]),
                            &(pDispParams->rgvarg[1]),
                            0,
                            VT_BSTR );

   // If we can't get a BSTR return, invalidate argument
   if ( FAILED( hr ))
      return( DISP_E_TYPEMISMATCH );

   // Coerce the variant into the desired type
   // In this case we would like an I4 or long
   hr = VariantChangeType( &(pDispParams->rgvarg[0]),
                           &(pDispParams->rgvarg[0]),
                           0,
                           VT_I4 );

   if ( FAILED( hr ))
      return( DISP_E_TYPEMISMATCH );

   // Finally call our method with the correct parameters
   SetExpression( pDispParams->rgvarg[1].bstrVal,
                  pDispParams->rgvarg[0].iVal );

   return( S_OK );
```

Calling the SetExpression method is the most complicated case. It does not return a value, but it takes two parameters. The DISPPARAMS structure is allocated and passed by the caller. It contains an array of parameters, each stored in a VARIANTARG structure. We first check to make sure that we have two parameters. If we don't, we return DISP_E_PARAMCOUNT. Next, we make sure that the caller has not passed any named arguments. We don't support them in our simple example. We then coerce the parameters into types that our internal SetExpression method supports.

VariantChangeType takes a variant type and coerces it into a type that you specify. In our first case, we need the actual Expression string as a binary string (BSTR). Automation controllers can pass this value in a number of ways. VariantChangeType does all the work for you. For example, here are some ways that Visual Basic might call SetExpression:

```
Dim objExp As Object
Dim v As Variant
Dim str As String

Set objExp = CreateObject("Chap6.Expression.1")
v = "100 * 500"
str = "100 * 600"
objExp.SetExpression str, 1
objExp.SetExpression (str), 1
objExp.SetExpression v, 1
' txtExpression is a standard Windows entry field
objExp.SetExpression txtExpression, 1
```

Each one of the preceding calls to SetExpression passes a different variant type for the BSTR parameter. The first one passes a BSTR by reference (ByRef). The second one passes it as we would typically expect, just a simple BSTR. The third call passes a variant by reference, and the fourth call passes a pointer to the IDispatch interface of Visual Basic's standard ActiveX Edit control. Thankfully, VariantChangeType handles the complexity of converting all these different types into the one we would like: a straight BSTR.

You might wonder how VariantChangeType can convert an IDispatch pointer into a BSTR. An Automation server can specify a default Value property that provides the value most appropriate for the server. Servers need not specify this default, but many of them do. VariantChangeType will attempt to obtain the type that you're trying to convert to by calling the Value property through the passed-in IDispatch. In the preceding example, the standard Visual Basic Edit control has a value property that returns the string within the edit field. This is exactly what we need, so everything works.

We've finished the hard work. All that remains is to change the return types of our previously exposed Expression class methods. Because they are no longer directly exposed, we don't need to treat them as COM interface functions. We only need remove the STDMETHOD macros from the declarations in **EXPSVR.H** and remove the definitions in **EXPSVR.CPP**. Here's an example from the CPP file.

```
//STDMETHODIMP_(BOOL) Expression::Validate()
BOOL Expression::Validate()
//STDMETHODIMP_(long) Expression::Evaluate()
long Expression::Evaluate()
```

Once we've made all the changes, we rebuild the project. We now have an `Expression` component that can be used from a variety of development environments, one of which is Visual Basic. Let's test our component with Visual Basic.

However, before we move on and test our new IDispatch-based server, we need to register the component. We're not using MFC, so we have to use a .REG file like we did with the Chapter 4 server. The CD-ROM contains the WIN32.REG file. Modify the InprocServer32 entry to include the path to the SERVER.DLL on your local machine. Then use Explorer or REGEDIT.EXE to add the entries to the Registry. Here's how it looks on my machine.

```
HKEY_CLASSES_ROOT\CLSID\{a988bd40-9f1a-11ce-8b9f-10005afb7d30}
      \InprocServer32 = d:\examples\chap6\Server\Debug\server.dll
```

Using Visual Basic as an Automation Controller

Visual Basic is easy to use. If you've never developed an application with Visual Basic, you should give it a try. Many developers dismiss it because it has "Basic" in its name, but our focus as developers should be on providing solutions to problems and not on a philosophical debate about which language is best. Visual Basic is the de facto Automation controller, and in this section we'll demonstrate why. Visual Basic eases the task of harnessing the functionality of ActiveX servers.


NOTE

If you don't have Visual Basic, don't worry. Visual C++ comes with an OLE/ActiveX test program that provides all the functionality we need to test our component. The file **DISPTEST.EXE** is in the \MSDEV\BIN directory. To run it, just type **start DISPTEST** from the command line.

Start Visual Basic and build a form that looks like the screen in Figure 6.3. It's easy—just drag and drop an entry field and two buttons from the tool palette. Name the entry field **txtExpression**, the **Validate** button **cmdValidate**, and the **Evaluate** button **cmdEvaluate**. To set these values, set focus to the control and then modify the value of the `Name` property in the Property window. You should also set the default `Text` property to a null string.

Figure 6.3 Visual Basic form.

Visual Basic has many built-in data types, most of which map directly to the standard Automation types. The one we're interested in is `Object`, which encapsulates an `IDispatch` pointer and so allows you to call the various methods on that `IDispatch`. The first step is to define an object. We do this in the form's globals section, which is the area outside any function or procedure.

```
Dim objExp As Object
```

At application startup, one of the first things to happen is the loading of the form, so we create an instance of the `Expression` object and assign it to the `objExp` pointer. The Visual Basic command `CreateObject` internally calls COM's `CoCreateInstance` method to create an instance of the COM object and then requests the default `IDispatch` for that object. Visual Basic stores this pointer in the `objExp` variable.

```
Sub Form_Load ()
    Set objExp = CreateObject("Chap6.Expression.1")
End Sub

Sub Form_Unload (Cancel As Integer)
    Set objExp = Nothing
End Sub
```

Visual Basic's version of `QueryInterface::Release` is to set the `objExp` variable to `Nothing`. This frees the connection to the Automation object. You need not do this explicitly (as we've done here), because Visual Basic will do it whenever the object goes out of scope. Still, it's good programming practice.

To use the object, we add the following lines of code. These methods should look familiar; we added them to our Automation wrapper earlier in this chapter. To add code for the button click event in Visual Basic, double-click on the button. Add the following code for each button.

```
Sub cmdEvaluate_Click ()
    Dim szExp As String
    szExp = txtExpression.Text
    objExp.SetExpression szExp, True
    txtExpression = objExp.Evaluate()
End Sub

Sub cmdValidate_Click ()
    Dim szExp As String
    szExp = txtExpression.Text
    objExp.SetExpression szExp, True
    If objExp.Validate() = False Then
        MsgBox ("Invalid Expression, try again")
    End If
End Sub
```

The syntax for calling object methods in Visual Basic is `object.Method parameters` if there isn't a return value, or `retValue = object.Method( parameters )` if there is a return value. As you can see, it's easy to use our Automation wrapped component. We've used Visual Basic to build an application around our C++ `Expression` class with just 13 lines of code. Press **F5** and run the application. You can step through each line of code using the **F8** key.

A Non-MFC Automation Controller

We've taken a look at what it takes to implement an Automation server, and we then tested it by using Visual Basic as the Automation controller. Next, let's take a quick look at what is required to implement the controller (or client) side in C++. We'll do this initially without the help of MFC's Automation support, but later we'll use MFC. I don't want to spend very much time in this section, but I think it's important for a solid understanding of what Automation is all about.

The accompanying CD-ROM contains the source for this simple client application in the \Examples\Chap6\Client directory. The example is built using a simple dialog-based Visual C++ application. It is similar to the examples in Chapter 3 through Chapter 5 except that it is dialog-based and does not use MFC's document-view architecture. To follow along and build the application as we go, first build a Visual C++ application with the following options:

1. Name the application **Client**.
2. Build an **MFC AppWizard (EXE)** project.
3. In AppWizard Step 1, choose a **Dialog based** application.
4. In AppWizard Step 2, accept the defaults of **About box** and **3D controls** and check the **OLE automation** support button. This action adds the OLE include files and the call to `AfxOleInit` for us. We know what these options do, so we'll let AppWizard do it for us.
5. In AppWizard Step 3, choose to use the MFC library as a **Shared DLL**.
6. In AppWizard Step 4, change the names of the **CClientDlg** class to **DIALOG.H** and **DIALOG.CPP**.
7. Click the **Finish** button. You should have a screen similar to Figure 6.4.

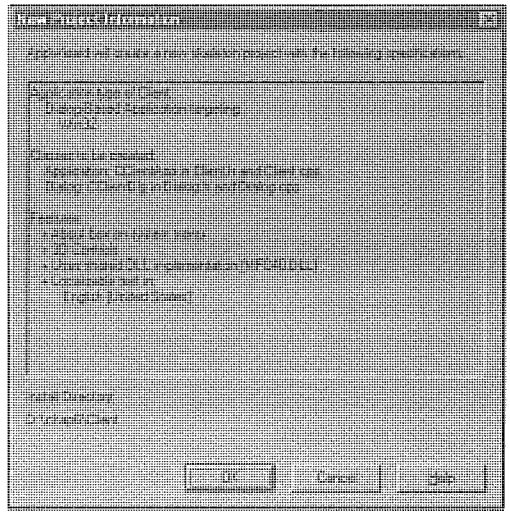


Figure 6.4 Non-MFC client application generation.

If you forget to add **OLE Automation** support to the application, you will get an error in the `CoCreateInstance` call. The `HRESULT` will be something like `0x800401F0`, which by itself doesn't mean much. When you get a `FAILED` status on a COM-based function call, you can look up a textual rendition of the error in the **WINERROR.H** include file. The `0x800401F0` error maps to "CoInitialize has not been called," a much clearer error message.

Just as we've done in previous versions of the CLIENT application, we'll add two buttons and an entry field to the dialog resource. The button IDs are `IDC_EVALUATE` and `IDC_VALIDATE`, and the entry field has an ID of `IDC_EXPRESSION`. Using ClassWizard, we then add, for both buttons, handlers for the `BN_CLICKED` event.

Once we've done all that, we start adding the code. There a few additions to **DIALOG.H**:

```
//
// Dialog.h : header file
//

class CClientDlg : public CDialog
{
...

// Implementation
protected:
   HICON        m_hIcon;
```

```
    LPDISPATCH  m_pDispatch;
    HRESULT     GetDispID( const char* szName, DISPID& dispid );
    BOOL        SetExpression( CString& strExpression );
    BOOL        Evaluate( long& lValue );
    BOOL        Validate( BOOL& bValid );
...
};
```

The `m_pDispatch` member will hold the `IDispatch` to the `Expression` component, and the other methods help implement the client-side `IDispatch` functionality. When the dialog box is created, the `OnInitDialog` method is called. This is a good place to create an instance of the `Expression` component, much as we did in Chapters 4 and 5. We use `CoCreateInstance` to create an instance of the `Expression` class, but this time we ask for the `IDispatch` interface instead of `IExpression`. The modifications below are to **DIALOG.CPP**:

```
//
// Dialog.cpp
//
...
// Use MFC's Unicode conversion functions
#include <afxpriv.h>
...
CClientDlg::CClientDlg(CWnd* pParent /*=NULL*/)
     : CDialog(CClientDlg::IDD, pParent)
{
    ...
    m_pDispatch = 0;
}

BOOL CClientDlg::DestroyWindow()
{
    // Release the dispatch interface
    if ( m_pDispatch )
       m_pDispatch->Release();

   return CDialog::DestroyWindow();
}

BOOL CClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    ...
```

```
    // Create the dispatch to our expression component
    // It must be a Unicode string
    USES_CONVERSION
    LPCOLESTR lpOleStr = T2COLE( "Chap6.Expression.1" );
    CLSID Clsid;
    HRESULT hr = ::CLSIDFromProgID( lpOleStr, &Clsid );
    if( FAILED( hr ))
    {
        AfxMessageBox( "CLSIDFromProgID failed\n" );
        return TRUE;
    }
    hr = CoCreateInstance( Clsid,
                           0,
                           CLSCTX_INPROC_SERVER,
                           IID_IDispatch,
                           (LPVOID FAR *) &m_pDispatch );
    if (FAILED( hr ))
    {
        AfxMessageBox( "CoCreateInstance failed" );
        return TRUE;
    }

    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

We added a handler for the `DestroyWindow` message so that we can release the dispatch when the dialog box is closed. Most of the preceding code should be familiar by now. All the work is done in `OnInitDialog`. First, we convert our ProgID to Unicode and then retrieve the associated CLSID. We then call `CoCreateInstance` and ask for the `IDispatch` of the component. If all goes well, we have all we need to call the component's methods directly.

Notice that we do not have any compile-time definition of the `Expression` component. We did not include **EXPSVR.H**. Instead, we will dynamically query for the DISPIDs of the methods and invoke them dynamically using the `Invoke` method of `IDispatch`. Here's a helper function to return a DISPID of the provided method name:

```
HRESULT CClientDlg::GetDispID( const char* szName, DISPID& dispid )
{
    HRESULT hr;
    // Get a Unicode version of the string
    USES_CONVERSION
    LPOLESTR lpOleStr = T2OLE( szName );
```

```
    hr = m_pDispatch->GetIDsOfNames( IID_NULL,
                                     &lpOleStr,
                                     1,
                                     LOCALE_SYSTEM_DEFAULT,
                                     &dispid );

    if (FAILED( hr ))
    {
        char szTemp[128];
        sprintf( szTemp, "GetIDsOfNames for '%s' failed, HRESULT is %x", szName, hr
                 );
        AfxMessageBox( szTemp );
        return hr;
    }

    return S_OK;
}
```

This is straightforward. We pass a Unicode version of the member name (such as `SetExpression`) and a pointer to a DISPID variable. We call the `GetIDsOfNames` implemented by the server, and, if everything works, we get back the DISPID of the member.

We need the DISPID because we use the `Invoke` method to access the functionality in the `Expression` component, and `Invoke` requires a DISPID. We could have obtained the DISPID from the type information of the server, but we didn't implement type information. We'll cover this later. For now, the client application must call `GetIDsOfNames` to get the DISPID. Here's how we call `SetExpression` in the server. I've added a `SetExpression` method to our dialog class:

```
BOOL CClientDlg::SetExpression( CString& strExpression )
{
    // OK, here we go. Get the DISPID of the SetExpression method
    DISPID dispid;
    if ( FAILED( GetDispID( "SetExpression", dispid )))
        return TRUE;

    // Now that we have the DISPID, call the method using IDispatch::Invoke

    DISPPARAMS dispparms;
    memset( &dispparms, 0, sizeof( DISPPARAMS ));
    dispparms.cArgs = 2;

    // allocate memory for all VARIANT parameters
    VARIANTARG* pArg = new VARIANTARG[dispparms.cArgs];
    dispparms.rgvarg = pArg;
```

```
    memset(pArg, 0, sizeof(VARIANT) * dispparms.cArgs);

    // The parameters are entered right to left
    dispparms.rgvarg[0].vt = VT_I4;
    dispparms.rgvarg[0].lVal = 1;
    dispparms.rgvarg[1].vt = VT_BSTR;
    dispparms.rgvarg[1].bstrVal = strExpression.AllocSysString();

    // Invoke the Start method in the local server
    HRESULT hr = m_pDispatch->Invoke( dispid,
                                IID_NULL,
                                LOCALE_SYSTEM_DEFAULT,
                                DISPATCH_METHOD,
                                &dispparms,
                                0,          // No return value
                                0,          // No exception support
                                NULL );

    // Free up our variantargs
    delete [] pArg;

    if ( FAILED( hr ))
    {
        char szTemp[128];
        sprintf( szTemp, "Unable to Invoke 'SetExpression'. HR is %x", hr );
        AfxMessageBox( szTemp );
        return TRUE;
    }

    return FALSE;
}
```

First, we get the DISPID of `SetExpression`. `SetExpression` takes two parameters and returns `void`. Once we have the DISPID, we build the parameter list for the call. The `DISPPARAMS` structure houses the parameters that we will pass via the `Invoke` method. Here's what it looks like:

```
typedef struct FARSTRUCT tagDISPPARAMS
{
    VARIANTARG FAR* rgvarg;             // Array of arguments.
      DISPID FAR* rgdispidNamedArgs;       // Dispatch IDs of named arguments.
      unsigned int cArgs;                        // Number of arguments.
    unsigned int cNamedArgs;                 // Number of named arguments.
} DISPPARAMS;
```

We set the number of arguments to 2 and allocate storage for two VARIANTARG structures. Next, we clear the structures, and finally we initialize the variants with the parameter data. Arguments are stored right-to-left in the DISPPARAMS structure, so we first pass a long with a value of 1. We then set up a BSTR with the string value by calling CString's AllocSysString member.

Once everything is set up, we call Invoke through our IDispatch pointer. The server then unpacks everything in the DISPPARAM structure and executes the method. We implemented that code earlier, so you should see how it all fits together. The other methods are very similar.

```
BOOL CClientDlg::Evaluate( long& lValue )
{
   DISPID dispid;
   if (FAILED( GetDispID( "Evaluate", dispid )))
      return TRUE;

   // Now that we have the DISPID, call the method using IDispatch::Invoke
   DISPPARAMS dispparms;
   memset( &dispparms, 0, sizeof( DISPPARAMS ));

   // This method returns a value, so we need a VARIANT to store it in

   VARIANTARG vaResult;
   VariantInit( &vaResult );

   HRESULT hr = m_pDispatch->Invoke( dispid,
                                     IID_NULL,
                                     LOCALE_SYSTEM_DEFAULT,
                                     DISPATCH_METHOD,
                                     &dispparms,
                                     &vaResult,
                                     0,
                                     NULL );

   if ( FAILED( hr ))
   {
      char szTemp[128];
      sprintf( szTemp, "Unable to Invoke 'Evaluate'. HR is %x", hr );
      AfxMessageBox( szTemp );
      return TRUE;
   }

   lValue = vaResult.lVal;
   return FALSE;
}
```

```
BOOL CClientDlg::Validate( BOOL& bValid )
{
   DISPID dispid;
   if (FAILED( GetDispID( "Validate", dispid )))
      return TRUE;

   // Now that we have the DISPID, call the method using IDispatch::Invoke
   DISPPARAMS dispparms;
   memset( &dispparms, 0, sizeof( DISPPARAMS ));

   // This method returns a value, so we need a VARIANT to store it in
   VARIANTARG vaResult;
   VariantInit( &vaResult );

   hr = m_pDispatch->Invoke( dispid,
                             IID_NULL,
                             LOCALE_SYSTEM_DEFAULT,
                             DISPATCH_METHOD,
                             &dispparms,
                             &vaResult,
                             0,
                             NULL );

   if ( FAILED( hr ))
   {
      char szTemp[128];
      sprintf( szTemp, "Unable to Invoke 'Validate'. HR is %x", hr );
      AfxMessageBox( szTemp );
      return TRUE;
   }

   bValid = vaResult.lVal;
   return FALSE;
}
```

The code in the `Evaluate` and `Validate` methods is very similar to the earlier `SetExpression`, with one exception: these methods return values and do not take parameters. We pass a blank `DISPPARAMS` structure and provide a `VARIANT` in which the server can store a return value. After `Invoke` returns, we extract the value from the `VARIANT` and pass it to the local caller.

After coding these methods, all that remains is to add the calls to the button handlers.

```
void CClientDlg::OnEvaluate()
{
```

```
    if (! m_pDispatch )
    {
        AfxMessageBox( "There is no dispatch" );
        return;
    }

    // Get the expression from the entry field
    CString strExpression;
    CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
    pWnd->GetWindowText( strExpression );

    if ( SetExpression( strExpression ))
    {
        AfxMessageBox( "Unable to 'SetExpression'" );
        return;
    }

    long lResult;
    if ( Evaluate( lResult ))
    {
        AfxMessageBox( "Unable to 'Evaluate' Expression" );
        return;
    }

    // Set the returned value in the entry field
    char szTemp[128];
    sprintf( szTemp, "%ld", lResult );
    pWnd->SetWindowText( szTemp );

    // Set focus back to the entry field
    GetDlgItem( IDC_EXPRESSION )->SetFocus();
}

void CClientDlg::OnValidate()
{
    if (! m_pDispatch )
    {
        AfxMessageBox( "There is no dispatch" );
        return;
    }

    // Get the expression from the entry field
    CString strExpression;
```

```
    CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
    pWnd->GetWindowText( strExpression );

    if ( SetExpression( strExpression ))
    {
        AfxMessageBox( "Unable to 'SetExpression'" );
        return;
    }

    BOOL bValid;
    if ( Validate( bValid ) )
    {
        AfxMessageBox( "Enable to 'Validate' expression" );
        return;
    }

    if ( !bValid )
        AfxMessageBox( "Invalid Expression, try again" );

    // Set focus back to the entry field

    GetDlgItem(IDC_EXPRESSION)->SetFocus();
}
```

Except for the new technique of calling the methods in the `Expression` component, this code is nearly identical to that used in the earlier CLIENT examples. Now build the application. It should behave just like the examples in Chapters 3, 4, and 5 and the Visual Basic example that we developed earlier in this chapter.

MFC and IDispatch

Next, we'll focus on the MFC facilities that help in the development of component applications that need Automation support. Initially, we'll focus on the server side of `IDispatch`. Later, we'll build an Automation controller similar to the one we built earlier, this time using Visual C++'s shortcut technique. MFC provides the ability to act as an Automation controller but only in a static way. ClassWizard's Automation tab has a **Read Type Library** button that reads a type library from an existing Automation server and provides an MFC wrapper class derived from `COleDispatchDriver`, which wraps each Automation property or method with a C++ method. If the component class adds methods, you must rebuild the `COleDispatchDriver`-derived class in order to use them. Currently, there is no MFC class that

encapsulates the ability to dynamically query and invoke functions (as Visual Basic does) within an Automation server, although it wouldn't be hard to implement one yourself.

MFC implements the `IDispatch` interface as part of the main COM class `CCmdTarget`. `CCmdTarget` maintains a *dispatch map* in much the same way that it maintains a COM interface map and the Windows message map structures. Implementing an Automation component class is easy in MFC. The only requirement is that the class be derived from `CCmdTarget` so that it will contain a hierarchy of dispatch maps. Dispatch maps implement the dispatch table (shown previously in Figure 6.1) and provide the mapping from a DISPID to a specific method of an Automation class.

We will discuss the MFC `IDispatch` implementation further as we develop the examples.

An Example MFC-Based Automation Server

I know you must be getting tired of our `Expression` class example, but I promise this is the second-to-last time you will see it. In Chapter 10, we'll finish the `Expression` example by developing an expression evaluation ActiveX control. For now, we're focusing on how COM and ActiveX work instead of actually using it. In the rest of the chapters we'll develop some neat examples.

What we'll do first with our example is to wrap the `Expression` class with an Automation wrapper. This is quick and easy, and it lets us access the `Expression` functionality from Visual Basic, C++, and other languages that support Automation. All our previous examples have been in-process servers, but we'll make this one a local server, for two reasons. First, we haven't learned this yet, and we need to understand the differences between the two implementations. Second, we'll use the marshaling capabilities of COM, which allow the interoperation of 16-bit and 32-bit applications.

Start Visual C++ and follow these steps to create the initial project:

1. Name the application **AutoSvr**.
2. Build an **MFC AppWizard (EXE)** project.
3. In AppWizard Step 1, choose a **Multiple documents** application.
4. In AppWizard Step 2 of 6, accept the default of **None** for database support.
5. In AppWizard Step 3 of 6, select **None** for **OLE Container support** and enable **OLE automation** support.
6. In AppWizard Step 4 of 6, take the default features but turn off **Print Preview** support.
7. In AppWizard Step 5 of 6, take the default **Regular DLL using Shared MFC DLL**.
8. In AppWizard Step 6 of 6, derive the view from **CFormView** and change the implementation file's name to **VIEW.H** and **VIEW.CPP**. Also, change the document class implementation files to **DOCUMENT.H** and **DOCUMENT.CPP**.
9. Click the **Finish** button. You should have a screen similar to Figure 6.5.

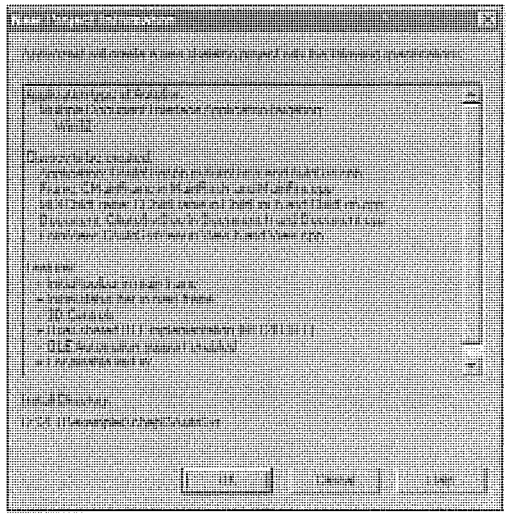


Figure 6.5 AUTOSVR project information dialog box.

Now we copy **EXPRESS.CPP** and **EXPRESS.H** from the Chapter 3 project into the working directory of our AUTOSVR project. After doing this, select **Insert/Files into project** and add **EXPRESS.CPP** to our project.

In Chapter 3, we used the `Expression` class directly in our project to provide algebraic expression capabilities. In Chapters 4 and 5, we used COM and ActiveX to separate the `Expression` class into a component that was accessed through a custom COM interface, and earlier in this chapter we implemented the component using a native `IDispatch` interface. Now we'll wrap the existing C++ class inside an Automation (`IDispatch`-based) wrapper class. This technique will expose the class's capabilities without modifying the implementation at all. The `Expression` class will be just as useful to C++ developers as it will be to developers who use Automation–capable tools (such as Visual Basic). Go ahead and compile, link, and run the application before we add the wrapping.

Wrapping the Expression Class

Start ClassWizard (**Ctrl-W**) and select the **OLE Automation** tab. Click the **Add Class** button to add a new class to our project. In the Class Name entry field, type **OExpression**. The default header and implementation filenames are fine. However, for this example I'm using the 8.3 names of **OEXPRESS.H** and **OEXPRESS.CPP**. In the Base class dropdown listbox, select **CCmdTarget**, the class from which our new class will be derived. As we discussed before, MFC's `CCmdTarget` provides the base COM functionality for all the MFC classes. Our Automation wrapper class will be derived from it.

When we chose to derive from `CCmdTarget`, three additional options were enabled in the OLE Automation frame. The second checkbox, **OLE Automation**, asks whether we want to enable this feature for our class. We do, so check it. When **OLE Automation** is checked, the **Createable by type ID** option is

enabled. This option, when checked, causes ClassWizard to create a CLSID and ProgID for our new Automation class. When your component class is to be directly exposed without going through the main OLE class for the application, you should check this option. If you do, the entry field will allow you to enter the ProgID for your Automation class. We want to allow direct access to the `Expression` class, so we check the **Createable by type ID** box and enter **Chap6.MFCExpression.1** in the entry field. See Figure 6.6.

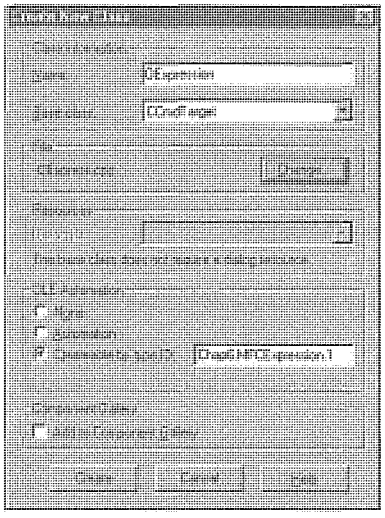


Figure 6.6 ClassWizard Create New Class dialog box.

Click the **Create** button and then press **OK** in the ClassWizard dialog box. This adds two new files to our application: **OEXPRESS.H** and **OEXPRESS.CPP**. Let's take a look at what they do. Here's **OEXPRESS.H**:

```
//
// Oexpress.h : header file
//

class OExpression : public CCmdTarget
{
   DECLARE_DYNCREATE(OExpression)
protected:
   OExpression();      // protected constructor used by dynamic creation

// Overrides
   // ClassWizard generated virtual function overrides
   //{{AFX_VIRTUAL(OExpression)
public:
   virtual void OnFinalRelease();
```

```
    //}}AFX_VIRTUAL
// Implementation
protected:
    virtual ~OExpression();

    // Generated message map functions
    //{{AFX_MSG(OExpression)
    // NOTE - the ClassWizard will add and remove member functions here.
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
    DECLARE_OLECREATE(OExpression)

    // Generated OLE dispatch map functions
    //{{AFX_DISPATCH(OExpression)
    // NOTE - the ClassWizard will add and remove member functions here.
    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
};
```

Most of this code should look familiar. In Chapter 5, we explored in detail most of the macros in **OEXPRESS.H**. We have the requisite macros that allow for dynamic creation. There's the class factory macro DECLARE_OLECREATE. The only new thing is the DECLARE_DISPATCH_MAP macro, we'll look at it in more detail in a moment. First, we'll add some Automation properties and methods:

```
// oexpress.cpp : implementation file
//
...

IMPLEMENT_DYNCREATE(OExpression, CCmdTarget)

BEGIN_DISPATCH_MAP(OExpression, CCmdTarget)
    //{{AFX_DISPATCH_MAP(OExpression)
        // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

IMPLEMENT_OLECREATE(OExpression, "Chap6.Expression.1", 0x9b027266, 0xb3e4, 0x11ce, 0xb6, 0xe, 0xf2,
0xbd, 0x12, 0x2b, 0xbc, 0x9)

OExpression::OExpression()
{
    EnableAutomation();

    // To keep the application running as long as an ActiveX Automation
```

```
    //    object is active, the constructor calls AfxOleLockApp.

    AfxOleLockApp();
}

OExpression::~OExpression()
{
    // To terminate the application when all objects created with
    //      with ActiveX Automation, the destructor calls AfxOleUnlockApp.

    AfxOleUnlockApp();
}

void OExpression::OnFinalRelease()
{
    // When the last reference for an automation object is released
    //    OnFinalRelease is called.  This implementation deletes the
    //    object.  Add additional cleanup required for your object before
    //    deleting it from memory.

    delete this;
}
```

Except for the call to `EnableAutomation` in the constructor, all the code in the implementation file, **OEXPRESS.CPP**, should also be familiar. All we need to do now is to add the methods and properties from the `Expression` class. We'll use (guess what?) ClassWizard to do this. Start ClassWizard and select the **OLE Automation** tab. Make sure that you have the **OExpression** class selected as well.

Using the **Automation** tab of ClassWizard allows you to add and remove Automation methods and properties. For the `Expression` class, we'll add four methods. Click the **Add Method** button, type **GetExpression** in the External Name field, and choose a return type of **BSTR**. Then press **OK**. This adds the `GetExpression` method to the `OExpression` class. We need to do this for each of the four methods of `Expression`. The `SetExpression` method takes two parameters. You add parameters by clicking in the Parameter list listbox and typing the parameter name in the left-hand side, and the parameter type on the right. Figure 6.7 shows the `SetExpression` method being added.

The two remaining methods—`Validate` and `Evaluate`—take no parameters. When you're finished adding all four methods, the **OLE Automation** tab should look like Figure 6.8.

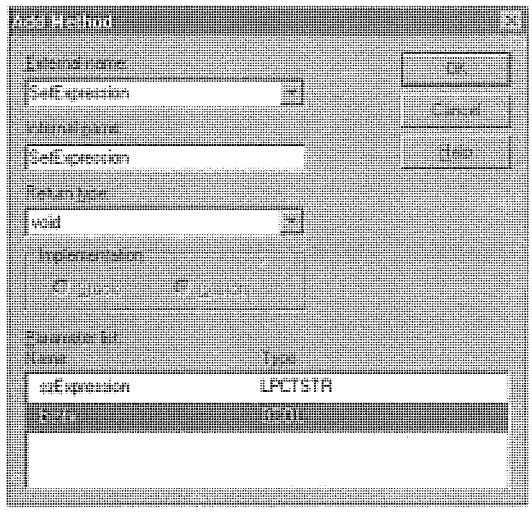


Figure 6.7 Adding the `SetExpression` method.

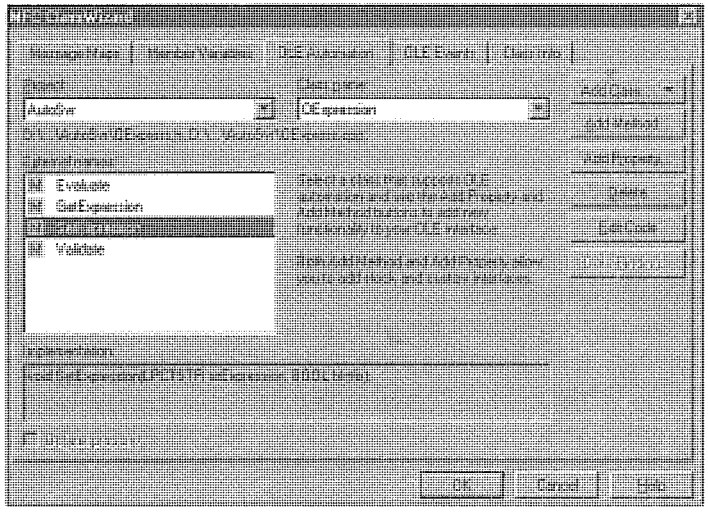


Figure 6.8 Adding Automation methods with ClassWizard.

ClassWizard has added four new methods to the `OExpression` class. It is now our responsibility to make them do something. All we are doing is wrapping the `Expression` class, so we include its declarations in **OEXPRESS.CPP**:

```
//
// oexpress.cpp : implementation file
//

#include "stdafx.h"
#include "autosvr.h"

#include "express.h"

#include "oexpress.h"
```

We also include a member of the `Expression` class within `OExpression` so that each instance of the `Expression` component has its own instance of `Expression`. Add the following to **OEXPRESS.H**:

```
class OExpression : public CCmdTarget
{
     DECLARE_DYNCREATE(OExpression)
protected:
     OExpression();           // protected constructor used by dynamic creation

// Attributes
public:
   Expression  m_Expression;

// Operations
public:
...
};
```

Now we need to access the `m_Expression` instance in the exposed Automation methods. This is easy—it's a one-to-one mapping of the methods and their parameters.

```
/////////////////////////////////////////////////////////////////////////////
// OExpression message handlers

BSTR OExpression::GetExpression()
{
   // TODO: Add your dispatch handler code here

   CString s = m_Expression.GetExpression();
   return s.AllocSysString();
}

void OExpression::SetExpression(LPCTSTR szExpression, BOOL bInfix)

{
   // TODO: Add your dispatch handler code here
```

```
    m_Expression.SetExpression( szExpression, bInfix );
}

BOOL OExpression::Validate()
{
    // TODO: Add your dispatch handler code here

    return m_Expression.Validate();
}

long OExpression::Evaluate()
{
    // TODO: Add your dispatch handler code here

    return m_Expression.Evaluate();
}
```

We're finished. That's all there is to wrapping an existing C++ class with Automation. We designed the class from the beginning to be compatible with Automation controllers by not passing structures, references, or pointers and using only native or Automation-supported data types. The specification for Automation supports passing certain types by reference, but currently there aren't any controllers that provide this functionality. I imagine, though, that this feature will soon be available.

Compile, link, and run the application. It doesn't do much, but running the application causes MFC to automatically update the Windows Registry with the `OExpression` component information. Any Automation controller can now access the `Expression` component.

MFC's Dispatch Macros

Now we'll take a look at how MFC built the dispatch table for the four methods that we added to `OExpression`. The following code was added by ClassWizard as we added the methods for the `OExpression` class:

```
// From oexpress.h
class OExpression : public CCmdTarget
{
...
    // Generated OLE dispatch map functions
    //{{AFX_DISPATCH(OExpression)
    afx_msg BSTR GetExpression();
    afx_msg void SetExpression(LPCTSTR szExpression, BOOL bInfix);
    afx_msg BOOL Validate();
    afx_msg long Evaluate();
    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
};
```

The preceding code is used by ClassWizard to declare, and keep track of, the internal methods used to implement the Automation properties and methods. The DECLARE_DISPATCH_MAP macro declares a few static member variables in the class to store and retrieve the dispatch map and its entries. Once the member functions are declared in the **.H** file, they are placed in the dispatch map in our **.CPP** file (also done for us by ClassWizard):

```
// From oexpress.cpp
..
BEGIN_DISPATCH_MAP(OExpression, CCmdTarget)
   //{{AFX_DISPATCH_MAP(OExpression)
   DISP_FUNCTION(OExpression, "GetExpression", GetExpression, VT_BSTR, VTS_NONE)
   DISP_FUNCTION(OExpression, "SetExpression", SetExpression, VT_EMPTY, VTS_BSTR
                 VTS_BOOL)
   DISP_FUNCTION(OExpression, "Validate", Validate, VT_BOOL, VTS_NONE)
   DISP_FUNCTION(OExpression, "Evaluate", Evaluate, VT_I4, VTS_NONE)
   //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()
...
```

The DISP_FUNCTION macro and its companions, DISP_PROPERTY and DISP_PROPERTY_EX, define the Automation property or method name, the implementing function or member variable, the return type, and the parameters for the entry in the dispatch map.

```
#define DISP_FUNCTION(theClass, szExternalName, pfnMember, vtRetVal, vtsParams) \
   { _T(szExternalName), DISPID_UNKNOWN, vtsParams, vtRetVal, \
     (AFX_PMSG)(void (theClass::*)(void))pfnMember, (AFX_PMSG)0, 0 }, \
#define DISP_PROPERTY(theClass, szExternalName, memberName, vtPropType) \
   { _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, (AFX_PMSG)0, (AFX_PMSG)0, \
     offsetof(theClass, memberName) }, \
#define DISP_PROPERTY_EX(theClass, szExternalName, pfnGet, pfnSet, vtPropType) \
   { _T(szExternalName), DISPID_UNKNOWN, NULL, vtPropType, \
     (AFX_PMSG)(void (theClass::*)(void))pfnGet, \
     (AFX_PMSG)(void (theClass::*)(void))pfnSet, 0 }, \
```

ClassWizard does all this for us. At first the idea of ClassWizard mucking around in your code can be a little disconcerting, but after you get used to it you'll think it's great.

Let's take a look at how MFC uses the dispatch map to process requests from a controller application. When CCmdTarget::EnableAutomation is called in the constructor of the OExpression class, EnableAutomation creates an instance of the MFC class COleDispatchImp, which contains the implementation of the IDispatch interface. EnableAutomation copies the Vtable of COleDispatchImp to

m_xDispatch, an internal variable within CCmdTarget. The following code, from CCmdTarget, illustrates this:

```
// enable this object for ActiveX Automation, called from derived class ctor
void CCmdTarget::EnableAutomation()
{
   ASSERT(GetDispatchMap() != NULL);    // must have DECLARE_DISPATCH_MAP

   // construct an COleDispatchImpl instance just to get to the vtable
   COleDispatchImpl dispatch;

   // vtable pointer should be already set to same or NULL
   ASSERT(m_xDispatch.m_vtbl == NULL||
      *(DWORD*)&dispatch == m_xDispatch.m_vtbl);

   // sizeof(COleDispatchImpl) should be just a DWORD (vtable pointer)
   ASSERT(sizeof(m_xDispatch) == sizeof(COleDispatchImpl));
   // copy the vtable (and other data) to make sure it is initialized
   m_xDispatch.m_vtbl = *(DWORD*)&dispatch;
   *(COleDispatchImpl*)&m_xDispatch = dispatch;
}
```

Once this is set up, a client application's QueryInterface call for IDispatch will return a pointer to the m_xDispatch.m_vtbl member. The client now has an IDispatch pointer that contains the methods: Invoke, GetIDsOfNames, GetTypeInfoCount, and GetTypeInfo. All these methods are implemented in the COleDispatchImp class. For an example, let's follow the sequence where the client initially calls the SetExpression Automation method. MFC doesn't provide an implementation of the GetTypeInfoCount and GetTypeInfo methods, so the client should call GetIDsOfNames to convert the string "SetExpression" to the appropriate DISPID. The following code shows the nonimplementation of the GetTypeInfo functions and the implementation of GetIDsOfNames:

```
STDMETHODIMP COleDispatchImpl::GetTypeInfoCount(UINT* pctinfo)
{
   *pctinfo = 0;
   return E_NOTIMPL;
}

STDMETHODIMP COleDispatchImpl::GetTypeInfo(UINT /*itinfo*/, LCID /*lcid*/,
   ITypeInfo** pptinfo)
{
   METHOD_PROLOGUE_EX(CCmdTarget, Dispatch)
   ASSERT_VALID(pThis);

   *pptinfo = NULL;
   return E_NOTIMPL;
}
```

```
STDMETHODIMP COleDispatchImpl::GetIDsOfNames( REFIID riid, LPTSTR* rgszNames, UINT cNames, LCID
/*lcid*/, DISPID* rgdispid )
{
   METHOD_PROLOGUE_EX(CCmdTarget, Dispatch)
   SCODE sc = S_OK;

   // fill in the member name
   const AFX_DISPMAP* pDerivMap = pThis->GetDispatchMap();
   rgdispid[0] = pThis->MemberIDFromName(pDerivMap, rgszNames[0]);
   if (rgdispid[0] == DISPID_UNKNOWN)

      sc = DISP_E_UNKNOWNNAME;

   // argument names are always DISPID_UNKNOWN (for this implementation)
   for (UINT nIndex = 1; nIndex < cNames; nIndex++)
      rgdispid[nIndex] = DISPID_UNKNOWN;

   return sc;
}
```

I've highlighted the preceding code that finds the DISPID for the given method or property name. `CCmdTarget::MemberIDFromName` does the work. Here it is:

```
MEMBERID PASCAL CCmdTarget::MemberIDFromName(
     const AFX_DISPMAP* pDispMap, LPCTSTR lpszName)
{
   // search all maps and their base maps
   UINT nInherit = 0;
   while (pDispMap != NULL)
   {
      // search all entries in this map
      const AFX_DISPMAP_ENTRY* pEntry = pDispMap->lpEntries;
      // How many entries in the dispatch map
      UINT nEntryCount = GetEntryCount(pDispMap);
      for (UINT nIndex = 0; nIndex < nEntryCount; nIndex++)
      {
         if (pEntry->vt != VT_MFCVALUE &&
            lstrcmpi(pEntry->lpszName, lpszName) == 0)
         {
            if (pEntry->lDispID == DISPID_UNKNOWN)
            {
               // the MEMBERID is combination of nIndex & nInherit
               return MAKELONG(nIndex+1, nInherit);
            }
```

```
            // the MEMBERID is specified as the lDispID
            return pEntry->lDispID;
        }
        ++pEntry;
        }
     pDispMap = pDispMap->pBaseMap;
    ++nInherit;
    }

   return DISPID_UNKNOWN;  // name not found
}
```

The preceding code spins through the dispatch map, maps the external name to the internal DISPID, and returns it to the client application (the Automation controller). The implementation of dispatch maps is similar to the implementation of MFC's COM interface maps that we covered in Chapter 5. You may have noticed the code that uses the `nInherit` flag. You can easily derive new classes from existing Automation-enabled MFC classes and then override or leave intact the Automation properties and methods from the base class. This is explained in *MFC Tech Note 39*.

When the client wants to invoke an Automation method or manipulate an Automation property, it calls the `COleDispatchImp::Invoke` method with the DISPID (which it obtained through `GetIDsOfNames`) and any required parameters. MFC maps the DISPID to a specific method (or property) and calls it on behalf of the client application. The code for `Invoke` is rather long, so I haven't included it here. Most of the MFC implementation of Automation is in **OLEDISP1.CPP** in the standard MFC source path \MSDEV\MFC\SRC.

Local Server Differences

In Chapters 4 and 5 we housed our components in DLLs because they used custom COM interfaces and therefore could not be marshaled across process boundaries. Now that we are using a standard COM interface, this is no longer necessary. Most of the differences between local and in-process servers occur in the `InitInstance` method of the derived `CWinApp` class.

When we built our application with AppWizard and answered **Yes, please** to the question concerning Automation, AppWizard generated quite a bit of Automation specific code for us. Our application now supports ActiveX Automation. AppWizard generated a unique CLSID for our application as well as a ProgID based on the project name. The component class within our application is the `CDocument`-derived class, so the generated ProgID is `AutoSvr.Document`. It's easy to change this ID; all you need to do is to modify the sixth substring of the `IDR_AUTOSVR_TYPE` string in the string table of the **.RC** file. (It's easy to identify. It's the only one with a period in it.) Following are selected lines from the `InitInstance` method in **AUTOSVR.CPP**.

```
// This identifier was generated to be statistically unique for your app.
// You may change it if you prefer to choose a specific identifier.
```

```
static const CLSID BASED_CODE clsid =
{ 0x77fc5ac3, 0xb494, 0x11ce, { 0xb6, 0xe, 0xfd, 0x5d, 0x8a, 0xfc, 0x39, 0x75 } };
```

This is the AppWizard-generated CLSID for our `CDocument` class. You will find it in the Registry associated with our application's ProgID. `CDocument` is the only (default) application class that derives from `CCmdTarget`, so it is the only class that can expose Automation methods and properties. As you will see in our next example, this is where you will typically add methods and properties to allow your application to be automated by an external Automation controller.

```
////////////////////////////////////////////////////////////////////////////
// CAutosvrApp initialization

BOOL CAutosvrApp::InitInstance()
{
   // Initialize ActiveX libraries
   if (!AfxOleInit())
   {
      AfxMessageBox(IDP_OLE_INIT_FAILED);
      return FALSE;
   }
```

Initialize the ActiveX libraries, just as we did in Chapter 5:

```
   // Connect the COleTemplateServer to the document template.
   //  The COleTemplateServer creates new documents on behalf
   //  of requesting OLE containers by using information
   //  specified in the document template.
   m_server.ConnectTemplate(clsid, pDocTemplate, FALSE);
```

This is a little more complicated. You can ignore the comment about "requesting OLE containers." The member variable, `m_server`, is an instance of `COleTemplateServer` and is declared within the `CAutosvrApp` class. `COleTemplateServer` is derived from `COleObjectFactory` and provides a class factory for the `CDocument`-derived class. The `ConnectTemplate` method attaches the server to the document template class. The third parameter, `bMultiInstance`, is an important one. It indicates whether an application instance should be invoked for every new document that is created. For MDI applications, this flag is set to `FALSE` to indicate that the application can support multiple client connections within a single EXE. For SDI applications, this flag is set to `TRUE`, which indicates that a new EXE will be launched for every client that creates an instance of the `CDocument` class.

```
   // Register all COM server factories as running.  This enables the
   //  COM libraries to create objects from other applications.
   COleTemplateServer::RegisterAll();
   // Note: MDI applications register all server objects without regard
   //  to the /Embedding or /Automation on the command line.
```

Here's where all the class factories for our local server are registered with COM. This call is the same as our previous chapter example of `COleObjectFactory::RegisterAll`. As the comment indicates, if this is an MDI application, the `RegisterAll` method is called regardless of the `/Automation` flag on the command line. This is because the application supports access by multiple controllers within the same EXE. In an SDI application, the `RegisterAll` method is called only if the EXE is started with the `/Automation` flag. The `/Automation` flag tells the EXE that it was launched on behalf of a controller, as opposed to being launched by an application user.

```
// Parse the command line to see if launched as ActiveX server
if (RunEmbedded() || RunAutomated())
{
   // Application was run with /Embedding or /Automation.  Don't show the
   //  main window in this case.
   return TRUE;
}
```

When COM starts an executable on behalf of a requesting client (or controller) via the `CoGetClassObject` function, COM uses the `LocalServer32` entry in the Registry. The entry should have the `/Automation` command-line option tacked on the end of the pathname. This arrangement enables the application to start without showing its main window. When you're using an Automation server, it isn't always necessary to see what the server is doing, so the default local server behavior is to not show its main window when started by an Automation controller.

```
// When a server application is launched stand-alone, it is a good idea
//  to update the system registry in case it has been damaged.
m_server.UpdateRegistry(OAT_DISPATCH_OBJECT);
COleObjectFactory::UpdateRegistryAll();

...
}
```

The preceding code updates the Registry with all the information that would normally be included in the **.REG** file for a server. When you add component classes to an MFC application, as we did with the `OExpression` class, this information will automatically be added to the Registry the next time the application is run without the `/Automation` switch. Let's see whether all this works by quickly building an application to test the `OExpression` component.

Type Information

In our discussions so far, the controller (or client) application has required certain information about the component before the application can access the component's functionality. In our examples, we've hard coded the names of the members in the server and have had to know the number of parameters and their types. In a component-based environment, this is hardly a good situation. A client application shouldn't

have to know the specifics of a component's services at compile time. A better approach would be for the server application to somehow advertise its functionality through some well-specified technique. In this way, generic clients could query a server programmatically for all the information needed to use a certain piece of functionality. COM provides such a mechanism, and it's called *type information*.

A component's type information describes, in a binary standard way, the capabilities of the server. The component's Automation methods and property names, their return values, and the number and types of their parameters can all be described using type information.

The type information for a component is described using Microsoft's Object Description Language. Visual C++ automatically maintains a file, **PROJNAME.ODL**, with the definition of all your control's methods, properties, and events. An exhaustive discussion of the structure of an ODL file is beyond our scope, but I've included a partial listing of the ODL file for the AUTOSVR example that we just developed. Whenever we added a method using ClassWizard's **OLE Automation** tab, ClassWizard wrote a line to the project's ODL file. Table 6.5 contains some of the important keywords used in an ODL.

```
//
// AutoSvr.odl : type library source for AutoSvr.exe
//
// This file will be processed by the Make Type Library (mktyplib) tool to
// produce the type library (AutoSvr.tlb).

[ uuid(A7699974-0060-11D0-A61D-E8F97D000000), version(1.0) ]
library AutoSvr
{
   importlib("stdole32.tlb");

   ...

   //  Primary dispatch interface for OExpression

   [ uuid(32CE8D20-0129-11D0-A61F-6C5374000000) ]

   dispinterface IOExpression
   {
      properties:
         // NOTE - ClassWizard will maintain property information here.
         //    Use extreme caution when editing this section.
         //{{AFX_ODL_PROP(OExpression)
         //}}AFX_ODL_PROP

      methods:
         // NOTE - ClassWizard will maintain method information here.
         //    Use extreme caution when editing this section.
         //{{AFX_ODL_METHOD(OExpression)
```

```
        [id(1)] void SetExpression(BSTR szExpression, boolean bInfix);
        [id(2)] BSTR GetExpression();
        [id(3)] long Evaluate();
        [id(4)] boolean Validate();
        //}}AFX_ODL_METHOD

};

   //  Class information for OExpression

   [ uuid(32CE8D21-0129-11D0-A61F-6C5374000000) ]
   coclass MFCExpression
   {
      [default] dispinterface IOExpression;
   };

   //{{AFX_APPEND_ODL}}
};
```

Table 6.5 ODL Keywords

Keyword	Purpose
`interface`	Standard COM-based, Vtable interface definition.
`dispinterface`	Defines an `IDispatch` interface. This includes the DISPIDs, properties, and methods for the interface.
`coclass`	Describes an implementation of a component object class.
`default`	Indicates that the interface is the default `IDispatch` interface. This is the one that is returned when a client queries for `IDispatch`.
`bindable`	The property supports being bound from the client and supports notification via the `IPropertyNotifySink::OnChange` method.
`requestedit`	The property supports the `OnRequestEdit` notification of `IPropertyNotifySink`.
`hidden`	When used as an attribute of a property, it indicates that the property should not be displayed by type library browsers.
`library`	Begins the definition of a type library. This gives its name and GUID. The GUID is listed in the Registry and provides a way for tools and components to locate the library's TLB file.
`source`	Indicates that the `IDispatch` interface is an outgoing one. This means that the `IDispatch` is not implemented by the control but instead is called by the control. This keyword is used for the control's event set.
`id(num)`	Defines the DISPID for the associated property or method.
`version`	Version of the library, interface, or class implementation.
`uuid( uuidval )`	The GUID that identifies the library or interface.

As you can see, the ODL describes the Automation methods that our `Expression` component exposes. The DISPID, name, and parameter types are all described using ODL keywords.

The ODL file must be compiled into a binary form for other OLE components to use. Visual C++ contains a utility, MKTYPLIB, that compiles the ODL file and produces a binary file with an extension of **TLB**. This file can be distributed as is, or it can be appended to a DLL or EXE file as a resource. Visual C++ automatically invokes the MKTYPLIB utility whenever the ODL file changes. It then binds the **.TLB** file into the resources of your server's housing file (either a DLL or EXE).

Once the type information is in a binary form, it can easily be queried by OLE tools and other components. The OLEVIEW utility can display the type information of a component. Just double-click on the `IDispatch` interface to display the details of the interface.

A client application can obtain and iterate over a server's type information in one of several ways. As we've seen, the `IDispatch` interface provides two methods—`GetTypeInfoCount` and `GetTypeInfo`—that provide access to the type information. However, this approach requires the client to instantiate a component before scanning a server's capabilities. This requirement can be expensive if the only reason for instantiating the component is to query type information. A client application (such as a component browser) that is interested only in a component's capabilities and does not want to use the component can scan the type information by accessing it directly in the housing. The **TypeLib** Registry entry provides the location of a component's type information.

Dual Interfaces

We now have the pieces to understand the concept of a *dual interface*. Dual interfaces are implemented by a server component and give the client application two different ways to access its functionality. We studied custom COM interfaces in Chapter 4, and in this chapter we took a look at `IDispatch`, a very useful ActiveX interface. A dual interface combines a custom interface with the standard `IDispatch` interface. This technique allows the client to choose which interface it wants to use.

Figure 6.9 depicts what the `Expression` component would look like with a dual interface. It is a combination of our custom interface (`IExpression`) and the `IDispatch` that we implemented earlier in this chapter. The `Expression` methods are now exposed directly through our Vtable and through the `IDispatch`.

Why should we expose two interfaces that provide basically the same functionality? The primary reason is performance. If the server has an in-process (DLL) implementation, then no marshaling is required. The client can directly bind to the custom interface methods and make very efficient calls. The performance with this method is nearly identical to that of direct C or C++ function bindings: very fast. However, if the client requires a local-server implementation of the server because of 32-bit-to-16-bit interoperation or another cross-process requirement, the client can use the `IDispatch` implementation. This method is slower because of marshaling requirements and the added time required to use the `Invoke` method and possibly `GetIDsOfNames`.

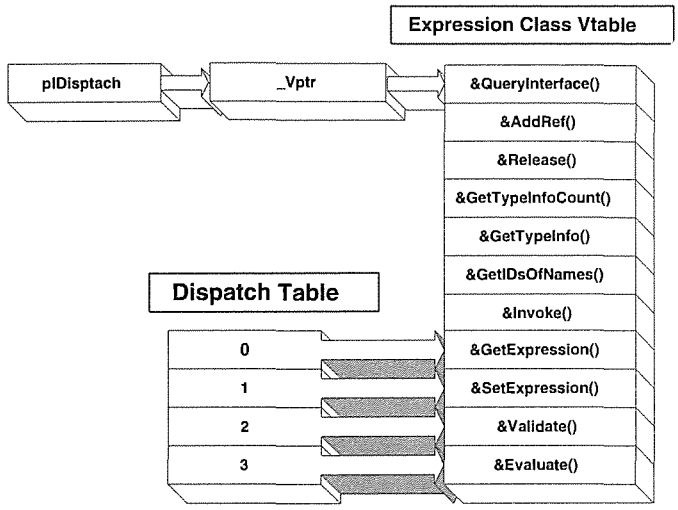


Figure 6.9 `Expression` class with a dual interface.

I say "possibly" because there are really three ways a client can bind to a dual interface server: late binding, ID binding, and early binding. Each one has specific type checking and performance characteristics. Here's a look at each one.

Late Binding

The examples that we've developed so far in this example have used late binding. We have ignored any type information provided by the server and have dynamically, at run time, determined the DISPID of the method to call; we have called `Invoke` through the `IDispatch` interface. Basically, everything occurred at run time. This is the most expensive technique and provides virtually no type checking. Any type checking is performed at run time by the server. Although this technique is the slowest, it is also the most flexible because everything is determined at run time. If the server interface changes, the client need not be recompiled to take advantage of the changes.

ID Binding

ID binding is very similar to late binding. The only difference is that the call to `GetIDsOfNames` is not made at run time. This approach saves a significant amount of time and dramatically improves performance. This technique requires that the server provide type information. At compile time, the controller application reads the type library and retrieves the DISPIDs of the called members. It can then statically store these values for use at run time.

ID binding also provides better compile-time type checking, because the compiler can determine the parameter types from the server's type library. However, if the members are rearranged in the server or if additional members are added, you must recompile to access the new functionality. Our next Visual C++ example will illustrate ID binding.

Early Binding

Early binding also requires that the server provide type information. It is the most efficient and the least flexible of the three techniques. As always, there is a trade-off. Early binding provides good type checking, because the controller uses the type information to verify the parameters as it builds them, which is done at compile time and not at run time. The binding is directly through the Vtable, so no DISPIDs or calls to `Invoke` are required. If the server is implemented in a DLL, the speed of early binding is the same as that of a direct DLL-type function call. However, as with ID binding, early binding requires a rebuild whenever the server component's interface is changed.

A Visual C++ Automation Controller

Actually, we've already built an Automation controller using Visual C++, but we did it without using MFC. Visual C++'s ClassWizard makes it easy to access the methods and properties of an Automation server. MFC uses ID binding in its implementation. Let's build a quick application to demonstrate how to access the functionality of the MFC-based server that we built in the previous example.

First, start Visual C++ and use AppWizard to create an application with the following options:

1. Name the application **VCClient**.
2. Build an **MFC AppWizard (EXE)** project.
3. In AppWizard Step 1, choose a **Single document** application.
4. In AppWizard Step 2 of 6, accept the default of **None** for database support.
5. In AppWizard Step 3 of 6, select **None** for **OLE Container support**, and be sure to enable **OLE automation** support.
6. In AppWizard Step 4 of 6, take the default features, but turn off **Print Preview** support.
7. In AppWizard Step 5 of 6, take the default Regular DLL using Shared MFC DLL.
8. In AppWizard Step 6 of 6, derive the view from **CFormView** and change the implementation file's name to **VIEW.H** and **VIEW.CPP**. Also, change the document class implementation files to **DOCUMENT.H** and **DOCUMENT.CPP**.
9. Click the **Finish** button.

Just as we've done in previous versions of the CLIENT application, we add two buttons and an entry field to the dialog resource (`IDD_VCCLIENT_FORM`). The button IDs are `IDC_EVALUATE` and `IDC_VALIDATE`, and the entry field has an ID of `IDC_EXPRESSION`. Using ClassWizard, we add, for both buttons, handlers for the `BN_CLICKED` event.

Next, we'll use ClassWizard to create a wrapper class for our Automation-enabled `Expression` component that we added to the AUTOSVR example. Start ClassWizard and follow these steps:

1. Go to the **OLE Automation** tab.
2. Click the **Add Class** button and choose **From an OLE TypLib**.
3. From the Import from TypLib file dialog box, select the **AUTOSVR.TLB** file from the \DEBUG directory of the AUTOSVR project.
4. In the Confirm Classes dialog box shown in Figure 6.10, change the names of the header and implementation files to **SVR.H** and **SVR.CPP**, respectively. This will remove any confusion with the AUTOSVR files from the previous project. Click **OK** to add the files to our project.

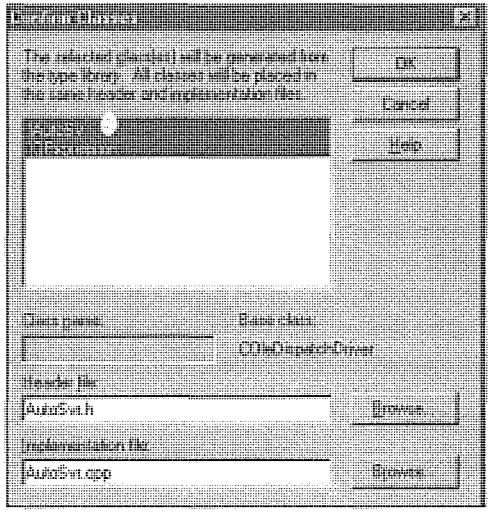


Figure 6.10 Confirm Classes dialog box.

What did all that do? Visual C++ opened the type library file of our AUTOSVR local-server application and created two classes that will make it easy to access any functionality housed in the **AUTOSVR.EXE** application. ClassWizard created two classes: `IAutoSvr` and `IOExpression`. Every AppWizard-based application gets a default Automation-enabled document class, and that is what the `IAutoSvr` class refers to. We didn't add any functionality to that class, but we will in the next section. For now, we'll focus on the `IOExpression` class. It provides an `IDispatch`-based interface to our `Expression` component within the AUTOSVR application. Here is the definition from **SVR.H**:

```
//
// svr.h
// Machine generated IDispatch wrapper class(es) created with ClassWizard
//
```

```
...

////////////////////////////
// IOExpression wrapper class

class IOExpression : public COleDispatchDriver
{
public:
   IOExpression() {}            // Calls COleDispatchDriver default constructor
   IOExpression(LPDISPATCH pDispatch) : COleDispatchDriver(pDispatch) {}
   IOExpression(const IOExpression& dispatchSrc) : COleDispatchDriver(dispatchSrc) {}

// Attributes
public:

// Operations
public:

   void SetExpression(LPCTSTR szExpression, BOOL bInfix);
   CString GetExpression();
   long Evaluate();
   BOOL Validate();
};
```

Here are those methods that we're familiar with. ClassWizard read the type library and created prototypes for the `OExpression` methods in the AUTOSVR application. Nearly all the work is done by MFC's `COleDispatchDriver` class, but before we look at it, here is the code from the **SVR.CPP** file:

```
//
// svr.cpp
// Machine generated IDispatch wrapper class(es) created with ClassWizard
//

#include "stdafx.h"
#include "svr.h"

...

void IOExpression::SetExpression(LPCTSTR szExpression, BOOL bInfix)
{
   static BYTE parms[] =
      VTS_BSTR VTS_BOOL;
   InvokeHelper(0x1, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
       szExpression, bInfix);
}
```

```
CString IOExpression::GetExpression()
{
   CString result;
   InvokeHelper(0x2, DISPATCH_METHOD, VT_BSTR, (void*)&result, NULL);
   return result;
}

long IOExpression::Evaluate()
{
   long result;
   InvokeHelper(0x3, DISPATCH_METHOD, VT_I4, (void*)&result, NULL);
   return result;
}

BOOL IOExpression::Validate()
{
   BOOL result;
   InvokeHelper(0x4, DISPATCH_METHOD, VT_BOOL, (void*)&result, NULL);
   return result;
}
```

This code is similar to the code that we developed at the beginning of this chapter in the CLIENT example. MFC, however, has a set of routines that make it easy to call Automation methods with various parameter types. MFC's implementation of the Automation client uses the ID binding technique that we just discussed. The first parameter of the `InvokeHelper` function is the DISPID of the method or property. This value is hard coded and is determined when the Automation wrapper class is generated. This technique eliminates the need for `COleDispatchDriver` to call `GetIDsOfNames`, but if the server's implementation is changed—say, by the addition of other methods—the whole class must be regenerated and the application recompiled and relinked. Remember, that's one of the drawbacks of the ID binding technique. Let's take a quick look at the `COleDispatchDriver` class.

COleDispatchDriver

MFC's `COleDispatchDriver` class provides a basic implementation of the client side of an Automation interface. It basically encapsulates the functionality that we developed in the non-MFC CLIENT example at the beginning of this chapter. Table 6.6 lists some useful members of the `COleDispatchDriver` class. We'll use some of them as we finish the VCCLIENT examples.

Table 6.6 Important COleDispatchDriver Members

Parameter	Purpose
m_lpDispatch	The IDispatch pointer that the instance is using.
CreateDispatch(LPCSTR ProgID)	Creates an instance of an automation object and attaches it to the wrapper class instance.
AttachDispatch(LPDISPATCH)	Attaches an existing IDispatch pointer to the wrapper class. Use this method if you have a dispatch pointer to an existing object.
ReleaseDispatch(void)	Releases any attached dispatch pointer.
InvokeHelper(...)	A help method used to call IDispatch::Invoke. This makes it a little easier to call members in the automation server component.
SetProperty(DISPID, VT, value)	Sets a property of the provided type and value.
GetProperty(DISPID, VT, value*)	Retrieves a property value of the described DISPID and type.

The IOExpression wrapper class uses the InvokeHelper method to call the Expression component's methods. This code was generated automatically for us; all we need to do is to create an instance of the wrapper class and use either the CreateDispatch or AttachDispatch method to create or attach an instance of an Expression component. We'll use CreateDispatch. The AttachDispatch method "attaches" an existing IDispatch pointer to the wrapper class, which assigns the pointer to the internal m_pDispatch member.

We will use the component in our view class, so make the following modifications to **VIEW.H**:

```
//
// View.h : interface of the CVCClientView class

#include "svr.h"

class CVCClientView : public CFormView
{
...

// Implementation
protected:
   IOExpression   m_IOExpression;
...

};
```

We include the wrapper class definition (**SVR.H**) and add an instance of the class as a member of our view class. When the view is created, we need to create an instance of the Expression component housed in the AUTOSVR application. Later, when we shut down our application, we will release our connection to the component. We can handle these steps in the view's constructor and destructor.

```
CVCClientView::CVCClientView()
     : CFormView(CVCClientView::IDD)
{
   //{{AFX_DATA_INIT(CVCClientView)
      // NOTE: the ClassWizard will add member initialization here
   //}}AFX_DATA_INIT
   m_IOExpression.CreateDispatch( "Chap6.MFCExpression.1" );
}

CVCClientView::~CVCClientView()
{
   m_IOExpression.ReleaseDispatch();
}
```

We create an instance using `CreateDispatch` and pass the ProgID of our component. `CreateDispatch` uses `CoCreateInstance` and queries for an `IDispatch` pointer, just as we did in the non-MFC CLIENT example. When we're finished, we call `ReleaseDispatch`, which releases the interface.

Now that we have a dispatch to our `Expression` component, we can use all the wrapper methods in the `IOExpression` class. First, though, we need to add handlers for the `IDC_VALIDATE` and `IDC_EVALUATE` buttons, just as we've done several times before. After you've done that, add the following code to the handlers:

```
void CVCClientView::OnEvaluate()
{
   // Get the expression from the entry field
   CString strExpression;
   CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
   pWnd->GetWindowText( strExpression );

   m_IOExpression.SetExpression( strExpression, TRUE );

   long lResult = m_IOExpression.Evaluate();

   // Set the returned value in the entry field
   char szTemp[128];
   sprintf( szTemp, "%ld", lResult );
   pWnd->SetWindowText( szTemp );

   // Set focus back to the entry field
   GetDlgItem( IDC_EXPRESSION )->SetFocus();
}

void CVCClientView::OnValidate()
{
```

```
    // Get the expression from the entry field

    CString strExpression;
    CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
    pWnd->GetWindowText( strExpression );

    m_IOExpression.SetExpression( strExpression, TRUE );

    if (! m_IOExpression.Validate() )
        AfxMessageBox( "Invalid Expression, try again" );

    // Set focus back to the entry field
    GetDlgItem(IDC_EXPRESSION)->SetFocus();
}
```

The code is nearly identical to what we've coded before; the only difference is the way that we're accessing the `Expression` component. We're now using MFC, and the `Expression` component resides in the AUTOSVR application.

When we call `CreateDispatch` in the view's constructor, COM starts the AUTOSVR executable. Because we're just using the services of AUTOSVR, we don't actually see the application. It's not in the task list, but if you use a utility such as PVIEW you'll see that it is running. When we shut down and release the `IDispatch` pointer, AUTOSVR shuts down as well.

Automating an MFC Application

At the beginning of the chapter, I mentioned that there are two uses of Automation that are important to the reuse of software. We've explored the first approach: using Automation to wrap existing code and exposing it for other applications and languages to use. The second technique is to turn an existing application into a component. That's what we'll do next. We'll allow our AUTOSVR application to be driven externally. In doing so, we will implement many of the Automation properties and methods that Microsoft recommends be exposed when an application provides Automation support. We'll also allow the controlling application to use our `Expression` functionality, although a little differently from the techniques we've discussed previously.

The first thing we need to do is to update the `CFormView` dialog box so that it has the `IDC_EXPRESSION` entry field and the `IDC_VALIDATE` and `IDC_EVALUATE` buttons that we've used before. I won't lead you through that again. Just edit the `IDD_AUTOSVR_FORM` dialog box and make it look like the one in Figure 6.11.

We're not going to use the `IExpression` method to implement the `Expression` class functionality; we've already exposed it for other applications to use. We will use the `Expression` class internally, just as we would any C++ object. Our focus now is to allow an external client to drive the behavior of our application.

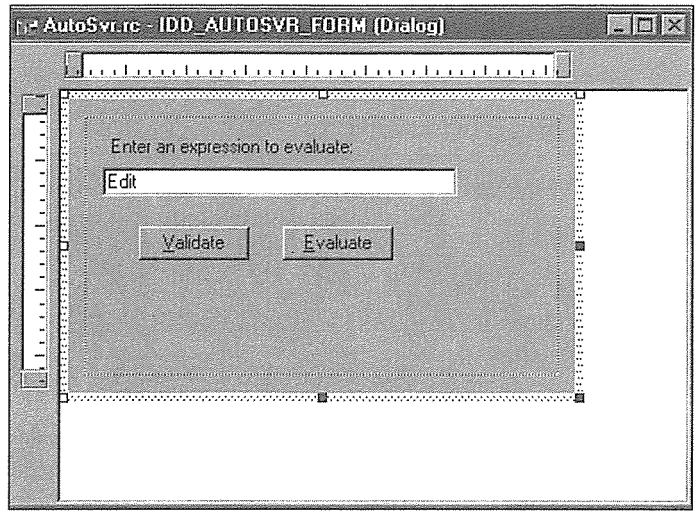


Figure 6.11 Building the AUTOSVR form dialog box.

When automating or allowing an external application to drive your application, you must expose properties and methods that are similar to ones that you use internally. The MFC document/view architecture makes it easy to allow your application to be automated, but only through the `CDocument`-derived class. Of the standard application classes provided by AppWizard in our project, only `CDocument` and its derivatives are derived from `CCmdTarget`. (Actually, the `CWnd` class is derived from `CCmdTarget`. Any true window classes in your application can support Automation, but they're not directly exposed by ClassWizard.) As you may recall, only classes derived from `CCmdTarget` have COM support within MFC. So we expose all our application's methods and properties through the `CDocument`-derived class, which in our application is `CAutosvrDoc`.

AUTOSVR is an MDI application and so can support multiple Automation clients simultaneously. Our first step is to allow an Automation controller to set the expression in our entry field and act as though it is pressing the **Validate** and **Evaluate** command buttons. To do this, we need to look at the document/view architecture again. Within an MFC MDI application, whenever a new document is created, a new frame/document/view tuple is created. This arrangement makes it easy to manage each MDI child window or view. There is one document instance and one or more view instances associated with each MDI child. So we need to add an instance of the `Expression` class to the document class. Why not the view class? It could also go there, but the `Expression` class contains the data of our application and so belongs within the document. As you'll see, this technique makes things fairly easy to implement. Add the following code to **DOCUMENT.H**:

```
// document.h : interface of the CAutosvrDoc class
//
/////////////////////////////////////////////////////////////////////////////
```

```
class Expression;

class CAutosvrDoc : public CDocument
{
protected: // create from serialization only
   CAutosvrDoc();
   DECLARE_DYNCREATE(CAutosvrDoc)

// Attributes
protected:
Expression* m_pExpression;

public:
Expression* GetExp() const { return m_pExpression; }

...
```

Here we've added a pointer to the `Expression` class and have included a forward declaration for the `Expression` class. This arrangement reduces the dependencies on the **EXPRESS.H** file. Most of the MFC application classes include **DOCUMENT.H** but don't need access to the definition of the `Expression` class. Because we've forward declared the class and have implemented it using a pointer, only **DOCUMENT.CPP** and **VIEW.CPP** need to include **EXPRESS.H**. We've also added a `GetExp` function that returns the expression pointer. This pointer will be useful when we need to access the `Expression` instance from within the view class. Here's the instantiation and destruction code for **DOCUMENT.CPP**:

```
// document.cpp : implementation of the CAutosvrDoc class
//

#include "stdafx.h"
#include "autosvr.h"
#include "express.h"
#include "document.h"

...
/////////////////////////////////////////////////////////////////////////////
// CAutosvrDoc construction/destruction

CAutosvrDoc::CAutosvrDoc()
{
   // TODO: add one-time construction code here

   EnableAutomation();
   AfxOleLockApp();

   m_pExpression = new Expression;
}
```

```
CAutosvrDoc::~CAutosvrDoc()
{
   AfxOleUnlockApp();
   delete m_pExpression;
}
```

Now go into ClassWizard and add the two functions for the `BN_CLICKED` event. Take the defaults `OnValidate` and `OnEvaluate`. Add the following code to **VIEW.CPP**:

```
// view.cpp : implementation of the CAutosvrView class
//

#include "stdafx.h"
#include "autosvr.h"
#include "document.h"
#include "express.h"
#include "view.h"
...
void CAutosvrView::OnEvaluate()
{
     // TODO: Add your control notification handler code here
   CString strExpression;
   char szTemp[128];

   // Get the expression from the entry field
   CWnd* pWnd = GetDlgItem(IDC_EXPRESSION);
   pWnd->GetWindowText( strExpression );

   TRACE1( "OnEvaluate: Expression is %s\n", strExpression );

   CAutosvrDoc* pDoc = GetDocument();
   pDoc->GetExp()->SetExpression( strExpression, TRUE );

   long lResult = pDoc->GetExp()->Evaluate();
   sprintf( szTemp, "%ld", lResult );

   pWnd->SetWindowText( szTemp );

   // Set focus back to the entry field
   GetDlgItem( IDC_EXPRESSION )->SetFocus();

}

void CAutosvrView::OnValidate()
{
```

```
    // TODO: Add your control notification handler code here
    CString strExpression;

    // Get the expression from the entry field
    CWnd* pWnd = GetDlgItem( IDC_EXPRESSION );
    pWnd->GetWindowText( strExpression );

    TRACE1( "OnValidate: Expression is %s\n", strExpression );

    CAutosvrDoc* pDoc = GetDocument();
    pDoc->GetExp()->SetExpression( strExpression, TRUE );

    if (! pDoc->GetExp()->Validate() )
        AfxMessageBox( "Invalid Expression, try again" );

    // Set focus back to the entry field
    GetDlgItem(IDC_EXPRESSION)->SetFocus();
}
```

This is a little different from our previous implementations. We now must get the document associated with our view in order to access the `Expression` instance. This is how typical MFC development is done. The data is stored in the document and the view must go and get it.

We've done enough that we should probably compile, link, and run. When you do, you should see something like the screen in Figure 6.12. It shows three active documents, each with an instance of the `Expression` class. On application startup, you only will see one MDI child window. You must explicitly create the others either through the **File/New** menu item or by using multiple Automation controllers.

Everything should be working from the perspective of the application user, so now we can allow the process to be automated by an external application. To do this, we expose some methods that correspond to user actions within the application. Start ClassWizard, go to the **OLE Automation** tab, select the `CAutosvrDoc` class, and add an `Expression` property. You should have a screen like that in Figure 6.13.

The External Name field holds the name that the controller will use to manipulate the property (or method). The type will change depending on which implementation method you choose. The **Stock** implementation is used for ActiveX controls, so we will see it shortly. The **Member Variable** implementation adds a member variable to your class and directly exposes it to the controller. If you don't really care what the controller is doing with the property, this may be the method to use. The **Get/Set Methods** implementation provides C++-style `Get` and `Set` methods to wrap around the member variable. This is probably the most typical technique used. The controller must go through the `Get` and `Set` methods to access the internal variable, so the application knows when it changes.

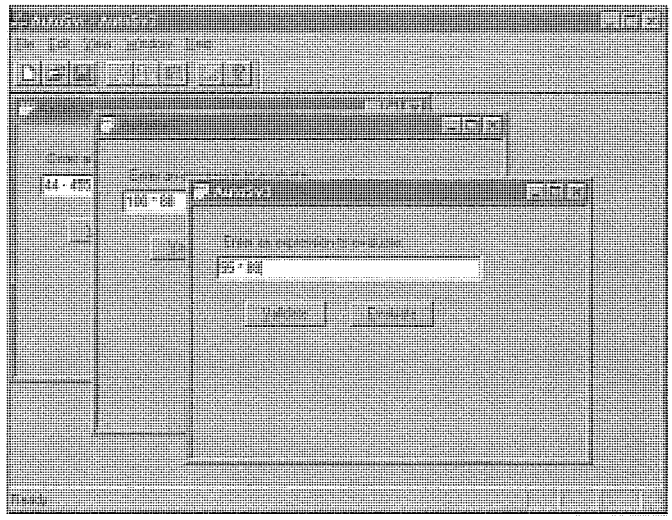


Figure 6.12 AUTOSVR application with three documents.

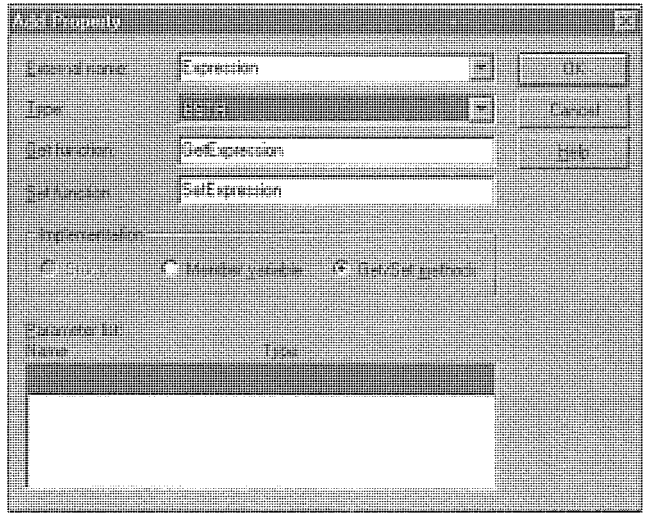


Figure 6.13 ClassWizard Add Property dialog box.

For our example, we'll choose a type of **BSTR** and an implementation of **Get/Set Methods**. If you don't want to allow the controller to set the property, clear out the **Set Function** entry field. This will put in its place a `SetNotSupported` method; if the controller attempts to set the function, it will throw an OLE

exception. For this example, we want to allow the property to be set, so don't clear the `Set` function's name. Click **OK** to add the property. Click **Edit Code** in the ClassWizard dialog box and add the following code:

```
////////////////////////////////////////////////////////////////////////////
// CAutosvrDoc commands

BSTR CAutosvrDoc::GetExpression()
{
   // TODO: Add your property handler here

   // Get the expression and return it to the Controller
   CString s = m_pExpression->GetExpression();
   return s.AllocSysString();
}

void CAutosvrDoc::SetExpression(LPCTSTR lpszNewValue)
{
   // TODO: Add your property handler here

   // Update the Expression instance with the new
   // expression, default the type to infix
   m_pExpression->SetExpression( lpszNewValue, TRUE );

   // Update the dialog form associated with the view
   // by sending it a hint as to what changed
   UpdateAllViews( NULL, VIEW_HINT_SETEXPRESSION );
}
```

The last section of code, which calls `UpdateAllViews`, needs some explanation. Now that we're using MFC's document/view architecture—in which the data is stored in the document class and the view is responsible for the display of this data—we must inform the views whenever any document data has changed. `UpdateAllViews` causes the `View::OnUpdate` function to be called for each view associated with the document. When the document calls `UpdateAllViews`, the document can pass a hint that helps the view determine what data has changed. I've set up three different hints that the view uses to determine what to do. These hints need to be defined in **DOCUMENT.H**:

```
// document.h : interface of the CAutosvrDoc class
//

class Expression;

const VIEW_HINT_SETEXPRESSION = 1;
const VIEW_HINT_VALIDATE      = 2;
const VIEW_HINT_EVALUATE      = 3;
```

The hints are used by the `OnUpdate` method of our view class. Here's the code that updates the appropriate view item, depending on the hint provided by the document class. You must override the `OnUpdate` method of `CAutosvrView` using ClassWizard and then add the following code to **VIEW.CPP**:

```
void CAutosvrView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
   // TODO: Add your specialized code here and/or call the base class

   switch( lHint )
   {
      // The Automation controller called the
      // validate function. Simulate a push
      // of the "Validate" button.
      case VIEW_HINT_VALIDATE:
         OnValidate();
         break;

      // The Automation controller called the
      // Evaluate function. Simulate a push
      // of the "Evaluate" button.
      case VIEW_HINT_EVALUATE:
         OnEvaluate();
         break;

      case VIEW_HINT_SETEXPRESSION:
         CWnd* pWnd = GetDlgItem(IDC_EXPRESSION);
         CAutosvrDoc* pDoc = GetDocument();
         pWnd->SetWindowText( pDoc->GetExp()->GetExpression() );
         break;

   }

}
```

As you can see, we either simulate a press of one of the buttons or set the text of the entry field. This technique makes it easy for the Automation class, `CAutosvrDoc`, to update the user's view as a controller invokes these methods.

We need to add Automation methods that allow an external application to simulate the press of our two buttons: **Validate** and **Evaluate**. Use ClassWizard to add two Automation methods—`Validate` and `Evaluate`—that take no parameters and return `void`. Then add the following code:

```
void CAutosvrDoc::Validate()
{
   // TODO: Add your dispatch handler code here
```

```
    UpdateAllViews( NULL, VIEW_HINT_VALIDATE );
}

void CAutosvrDoc::Evaluate()
{
    // TODO: Add your dispatch handler code here
    UpdateAllViews( NULL, VIEW_HINT_EVALUATE );
}
```

There isn't much to do here. We just route the update to the view class with the specific hint, and the view simulates actions of a user. That was too easy. We need to add a little more to the AUTOSVR application.

Standard Application Properties

When we're providing Automation capabilities for a full application, as we're doing here with the AUTOSVR project, Microsoft recommends that certain standard properties and methods be supplied for the external application user. Microsoft also recommends a specific hierarchy for the Automation objects that are contained with the application. At the root is the `Application` object. An application contains a list of document objects, and so on. This hierarchy is consistent with the structure of typical MFC applications. It provides consistency among applications developed by various vendors and makes it easier for developers when they begin using the Automation capabilities of an application; the naming and object hierarchy are consistent. This standard is detailed in the *OLE 2 Programmer's Reference, Volume Two*. Table 6.7 lists some of the recommended `Application` object properties in AUTOSVR to add a few more options when we're driving our application via an Automation controller.

Table 6.7 Application Object Properties

Type/Property	Purpose
`LPDISPATCH Application (Read-only)`	Returns the `IDispatch` for the application object.
`BSTR Name (Read-only)`	Returns the name of the application.
`BSTR FullName (Read-only)`	Returns the complete pathname and filename for the application.
`long Top`	Sets or returns the distance from the top of the display to the top of the main application window.
`long Left`	Sets or returns the distance from the left edge of the display to the left edge of the main application window.
`long Height`	Sets or returns the height of the application's main window.
`long Width`	Sets or returns the width of the application's main window.
`BOOL Visible`	Toggles the visibility of the application windows.

The Chapter 6 AUTOSVR project on the accompanying CD-ROM implements all the standard `Application` properties described in Table 6.7. They're all fairly easy to implement, and you might try them yourself before checking what's on the CD-ROM. As you're developing the properties for the AUTOSVR project, you need a way to easily test the new properties and methods you've added. We need another Visual Basic application.

Driving the Autosvr Example

Now that we have added properties and methods to AUTOSVR, we need a controller to test what we've done. We'll be using the DISPTEST tool provided with Visual C++. If you have a current version of Visual Basic, it will work, too. Figure 6.14 shows the Visual Basic form of our VBDRIVER example, which we will use to control AUTOSVR.

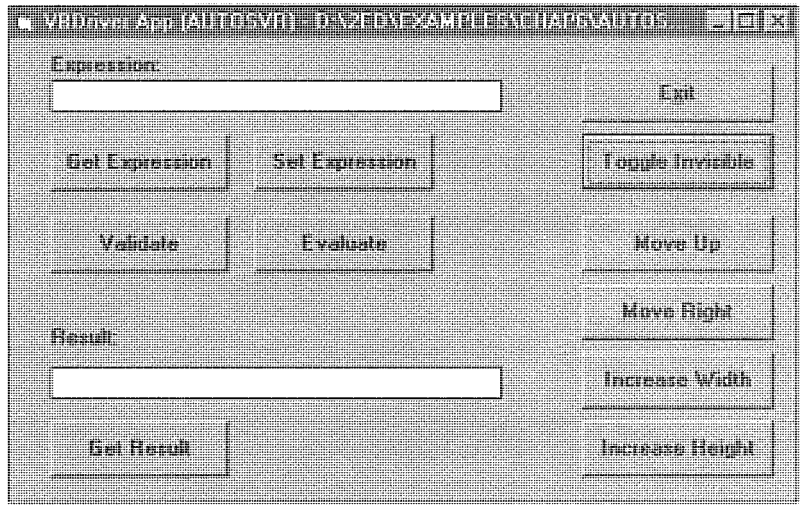


Figure 6.14 Visual Basic driver application.

The form has buttons that map to the various Automation methods and properties that we have added to AUTOSVR. To get things going, we connect to the main application object when we initially start the Visual Basic application.

```
' Global Form declaration
' Declared outside of any form procedure
Dim obj as Object

Sub Form_Load ()
    Dim objApp As object
```

```
    Set obj = CreateObject("Autosvr.Document")
    Set objApp = obj.Application
    Form1.Caption = "VBDriver App (" + objApp.Name + ") - " + objApp.FullName
    Set objApp = Nothing
End Sub
```

Here we are creating an instance of AUTOSVR. The `CreateObject` function will start the application if it isn't running and will create a new document if the application is running. MFC MDI applications allow multiple concurrent clients to access its document component. Whenever a new instance is created, a new document/view/template structure is created to service the controller.

I've added some code to test the various `Application` object properties we added. The `obj.Application` property returns an `LPDISPATCH` that we test by calling the `Name` and `FullName` properties. The rest of the code is fairly simple. Following is some of the code. It would be best to get the source from the accompanying CD-ROM, load Visual Basic (or **DISPTEST.EXE**), and experiment with the properties.

```
' Toggle the visibility of the main application window
' as well as the caption of the command button
Sub cmdVisible_Click ()
    If obj.Visible Then
        obj.Visible = False
        cmdVisible.Caption = "Toggle Visible"
    Else
        obj.Visible = True
        cmdVisible.Caption = "Toggle Invisible"
    End If
End Sub

' Increate the width of the application window
Sub cmdWidth_Click ()
    obj.Width = obj.Width + 5
End Sub
```

Summary

In this chapter, we investigated a powerful feature of ActiveX called Automation. Automation can be used to encapsulate a C++ language class in a binary standard wrapper that can easily be accessed by non-C++ languages. Automation also provides facilities that make it easy to control an application externally by another application or programming tool such as Visual Basic.

The backbone of Automation is the IDispatch interface and its four methods: Invoke, GetIDsOfNames, GetTypeInfoCount, and GetTypeInfo