PRIMA

PRACTICAL PROGRAMMING SERIES

SYMANTEC® C++

OBJECT-ORIENTED PROGRAMMING FUNDAMENTALS FOR THE MACINTOSH

COVERS VERSIONS 6 & 7

INCLUDES INTERACTIVE TRAINING SOFTWARE AVAILABLE NOWHERE ELSE!

Includes Macintosh Simulator C++— an Innovative Tutorial Program on Disk!

- **Unleash** the Full Power of Your Macintosh—Even if You Are Not a Programmer
- **Learn** Macintosh Programming Fundamentals Quickly and Easily
- **Master** the Basics of Symantec C++ Programming Language from the Ground Up



VALUABLE MAC SIMULATOR C++ SOFTWARE INCLUDED

PRIMA'S MACINTOSH SERIES

DAN PARKS SYDOW

ISBN 1-55958-633-8




Symantec
C++

Now Available from Prima!

Advanced PageMaker 4.0 for Windows
CompuServe Information Manager for Windows: The Complete Membership Kit & Handbook (with two 3½" disks)
Computers Don't Byte—The Absolute Beginner's Guide to
CorelDRAW! 4 Revealed!
Create Wealth with Quicken
DESQview: Everything You Need to Know
DOS 6.2: Everything You Need to Know
Free Electronic Networks
Harvard Graphics for Windows: The Art of Presentation
Improv for Windows Revealed! (with 3½" disk)
Lotus Notes 3 Revealed!
LotusWorks 3: Everything You Need to Know
Making Movies with Your PC
Microsoft Office In Concert, Professional Edition
NetWare 3.x: A Do-It-Yourself Guide
Novell NetWare Lite: Simplified Network Solutions
1-2-3 for Windows: The Visual Learning Guide
PageMaker 5 for the Mac: Everything You Need to Know
Quattro Pro 4: Everything You Need to Know
QuickTime: Making Movies with Your Macintosh
The Software Developer's Complete Legal Companion (with 3½" disk)
SuperPaint 3: Everything You Need to Know
Think THINK C (with two 3½" disks)
WinFax PRO: The Visual Learning Guide
Word for Windows 6: The Visual Learning Guide
WordPerfect 5.1 for Windows Desktop Publishing By Example
WordPerfect 6 for Windows: How Do I...?

Upcoming Books

CorelDRAW! 5 Revealed!
Excel 5 for the Mac: The Visual Learning Guide
Word 6 for the Mac: The Visual Learning Guide

How to Order:

Individual orders and quantity discounts are available from the publisher, Prima Publishing, P.O. Box 1260BK, Rocklin, CA 95677-1260; phone: (916) 632-4400. On your letterhead include information concerning the intended use of the books and the number of books you wish to purchase. Turn to the back of the book for more information.



Object-Oriented Programming Fundamentals for the Macintosh

Dan Parks Sydow

Prima Publishing
P.O. Box 1260BK
Rocklin, CA 95677-1260
(916) 632-4400

Prima Computer Books is an imprint of Prima Publishing, Rocklin, California 95677

© 1994 by Dan Parks Sydow. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without written permission from Prima Publishing, except for the inclusion of quotations in a review.

Executive Editor: Roger Stewart
Managing Editor: Neweleen A. Trebnik
Acquisitions Editor: Sherri Morningstar
Project Editor: Steven Martin
Copy Editor: Betsy Ahl
Indexer: Lynn Brown
Technical Reviewer: Peter Ferrante
Design and Production: Susan Glinert, BookMakers
Cover Design: Page Design, Inc.

If you have problems installing or running Symantec C++, notify the Symantec Corporation. Prima Publishing cannot provide software support.

Information contained in this book has been obtained by Prima Publishing from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Prima Publishing, or others, the publisher does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions of the results obtained from use of such information

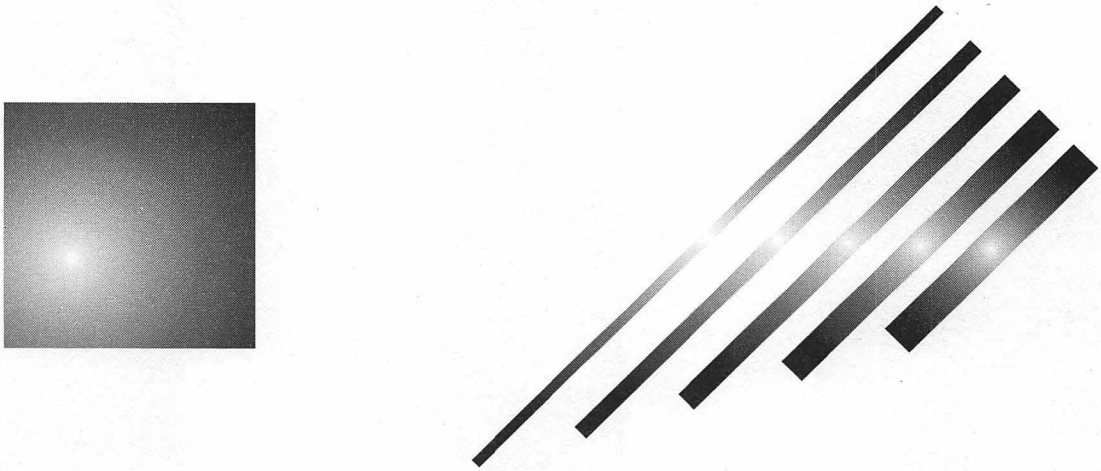
ISBN: 1-55958-633-8

Library of Congress Catalog Card Number: 94-066733

Printed in the United States of America

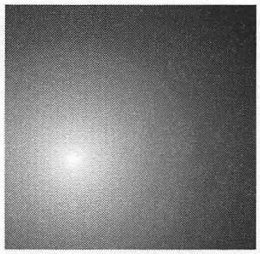
94 95 96 97 BC 10 9 8 7 6 5 4 3 2 1

To my wife, Nadine



Contents at a Glance

	Acknowledgments	xiii
	Introduction	xv
1	Using the Symantec C++ Compiler	1
2	Introduction to C++ and OOP	31
3	The C Language: The Basis of C++	51
4	Additions to C... Means C++	77
5	Classes and Objects	109
6	Derived Classes	159
7	Abstract Classes	199
8	Dynamic Binding	221
9	Objects and the User	265
10	Windows as Objects	309
11	A Complete Example	345
A	Getting and Using QuickTime	387
B	Menu Handling	403
	Index	417





Contents

Acknowledgments xiii

Introduction xv

Whom This Book and Software Are For xvi
What You Need xvi
Using the Book xvii
Running the Simulator C++ Software xvii
Running the Example Programs xviii
How to Use This Book and Software xviii
Using the Symantec C++ Examples xviii
Using the Simulator C++ Software xviii
Features of the Simulator Software xx
Installing the Simulator C++ Software xx
Using the Simulator Software xxiv
Running the Simulator C++ Software xxiv
Pages and the Control Panel xxiv
Simulator Pages xxvii

Chapter 1 Using the Symantec C++ Compiler 1

Creating a New Project 1
Creating a Project Using Symantec C++ 6.0 3
Creating a Project Using Symantec C++ 7.0 5

Adding Files to a Project 9
Adding Libraries to a C++ Project 9
Adding a Source Code File to a C++ Project 12
Segmenting a Project 15
C++ Code: Writing It and Running It 17
Using the Symantec Debugger 19
Debugger Basics 20
Debugging a Program 23
Turning Code into an Application 26
Using the Included Projects 28
About That Code... 28
Chapter Summary 29

Chapter 2 Introduction to C++ and OOP 31

C++ and Object-Oriented Programming 31
C, C++, and OOP 32
From C to C++ to OOP 33
Procedural and Object Programming 34
Procedural Programming 34
Object-Oriented Programming 35
The Advantages of Object-Oriented Programming 38
Classes 42
The Class—the Pattern of Objects 42
Creating Multiple Objects 43
Object-Oriented Programming and C 45
Everything Need Not Be an Object 45
C++ Uses C 46
Chapter Summary 49

Chapter 3 The C Language: The Basis of C++ 51

Basic Data Types 51
Integral Numbers 52
Floating-Point Numbers 53
Characters and Strings 53
Preprocessor Directives 54
The #define Directive 55
The #include Directive 55
Operators 56
Looping Statements 56
The while Loop 57
The do-while Loop 57
The for Loop 60
Branching Statements 60
The if Branch 60
The if-else Branch 62
The else-if Branch 63
The switch Branch 65
Structures 67
Defining and Declaring a struct 67

Accessing struct Members 68
The struct and class Data Types 71
Chapter Summary 75

Chapter 4 Additions to C... Means C++ 77

The Very Basics 77
Functions 78
Comments 79
Function Overloading 79
Functions with a Different Number of Arguments 79
Functions with Different Argument Types 83
Why Create Functions with the Same Name? 84
Allocating Memory in C 85
Pointer Review 85
Using Pointers 86
Pointers and struct Variables 89
Using the Symantec Debugger 92
Using the Debugger to Verify a Proper Memory Allocation 92
Using the Debugger to Examine Bad Memory Allocation 94
Allocating Memory in C++ 99
The Scope Resolution Operator 105
Chapter Summary 107

Chapter 5 Classes and Objects 109

Declaring a Class 109
Defining Member Functions 112
Writing the Header of a Member Function 112
Writing the Body of a Member Function 115
Working with Objects 117
Declaring an Object 117
Objects and Member Functions 121
Invoking a Member Function 122
Objects and Member Functions—Round Two 127
Deleting an Object 129
Multiple Objects 130
Accessing Data Members 134
Data Access via Member Functions 134
Using the private and public Keywords to Limit Access 134
Accessing Data without Using Member Functions 135
The this Operator 139
Constructors and Destructors 143
Constructors 143
Destructors 149
Chapter Summary 156

Chapter 6 Derived Classes 159

Multiple Classes 159
Derived Classes 162
Why Create Derived Classes? 162

The Base Class 166
The Derived Class 167
Working with Derived Class Objects 169
Creating Derived Objects 170
Using Derived Objects 171
Derived Objects and Data Member Access 173
An Example Using Derived Objects 178
A Second Example Using Derived Objects 180
Overriding Member Functions 185
Why Override Member Functions? 185
Overriding a Function 187
An Overriding Example 191
Chapter Summary 197

Chapter 7 Abstract Classes 199

Why Abstract Classes? 199
Creating an Abstract Class 201
The Abstract Class Data Members 201
The Abstract Class Member Functions 202
Creating a Family of Classes 204
The Derived Classes 204
The Member Functions of the Derived Classes 206
An Abstract Example 209
The Class Hierarchy 216
Chapter Summary 219

Chapter 8 Dynamic Binding 221

Returning Objects from Functions 222
A Shape as an Object 222
Returning an Object 227
Function Prototypes and Forward References 229
A Returned Object Example 230
Returned Objects and Derived Classes 233
Rectangles and Derived Classes 233
Dynamic Binding 239
A Dynamic Binding Example 243
The Rectangle Class—Still an Abstract Class? 249
Passing Objects to Functions 249
A Rectangle Object as a Parameter 249
The PassedRectangle Example Program 254
Returned Objects and the Animal Class 257
Chapter Summary 263

Chapter 9 Objects and the User 265

Using an Alert to Create a New Object 265
Alert Resources 267
Using the Alert to Select an Object Type 268
An Alert Example 269

Updating an Object 274
The Need to Redraw a Window's Contents 274
Update Events and Redrawing a Window's Contents 276
Testing the Object Update 279
Updating an Object—an Example 284
Using a Dialog Box to Create a New Object 289
Dialog Box Resources 289
Handling Radio Button Items in a Dialog Box 291
Handling Edit Text Items in a Dialog Box 293
Creating a New Object Using a Dialog Box 294
A Dialog Box Example Program 299
Chapter Summary 308

Chapter 10 Windows as Objects 309

Window Basics 309
Opening a Window 310
Window Data Types 310
Windows and Events 313
A Multiple-Windows Example Program 317
Representing Windows as Objects 325
The Window Class 325
Windows and the Constructor Function 327
Verifying That Objects Are Distinguishable 333
Windows and Events 335
An Example of Windows as Objects 336
Chapter Summary 343

Chapter 11 A Complete Example 345

DerivedWindows: A Complete OOP Example 345
What the Program Does 346
The DerivedWindows Resources 350
The DerivedWindow Classes 353
The WindowClass Member Functions 357
The PictWindow Member Functions 359
The PetWindow Member Functions 361
Updating Object Windows 362
Menus and Objects 363
Objects and User Input 365
The DerivedWindows Source Code 369
What's Next? 385

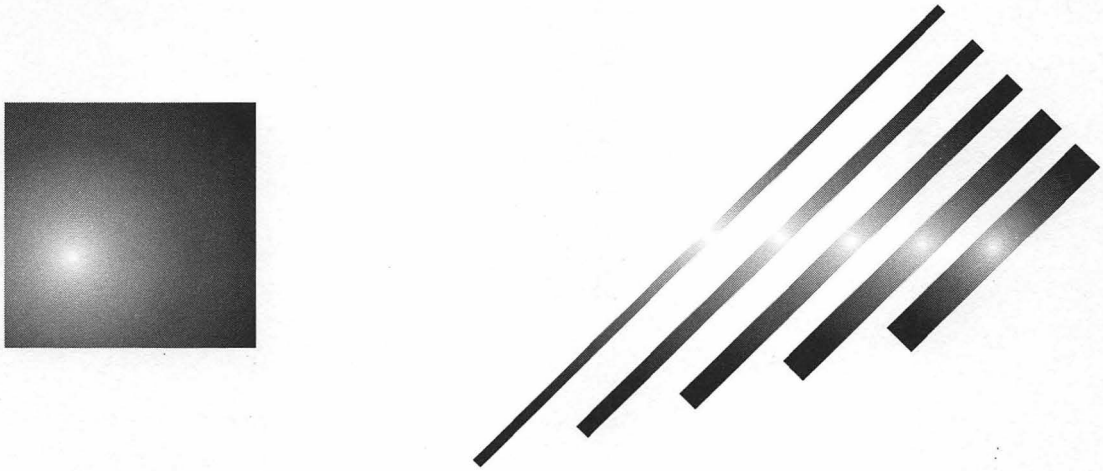
Appendix A Getting and Using QuickTime 387

Getting QuickTime 387
Downloading from America Online 388
Downloading from CompuServe 390
Downloading from GEnie 395
Installing and Using QuickTime 401

Appendix B Menu Handling 403

Menu Resources 403
Menu Code 405
Menu Example 408

Index 417



Acknowledgments

I'd like to take this opportunity to thank the several people who helped make this book a reality:

Steven Martin, Prima Publishing Project Editor, for having a sense of humor while keeping things moving through the production cycle.

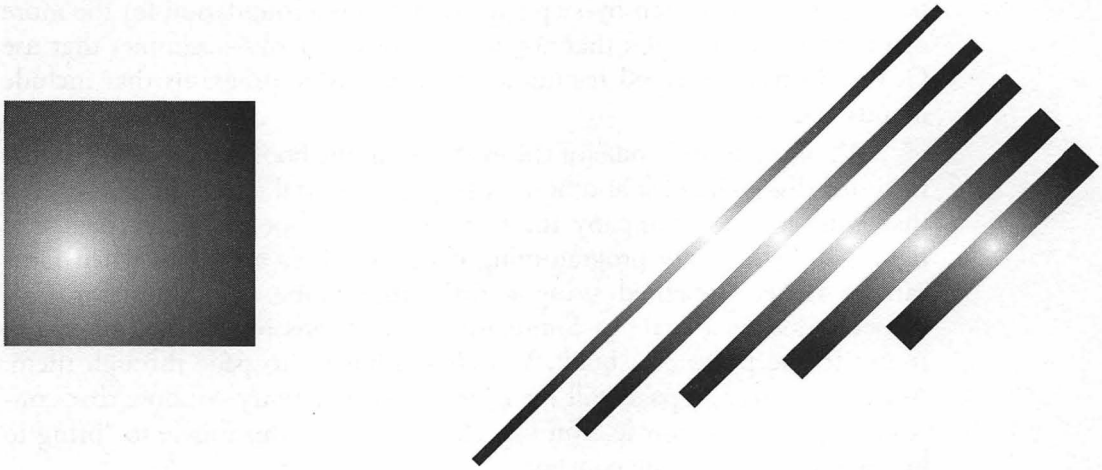
Peter Ferrante, Apple Computer, for a technical edit that resulted in a wealth of helpful comments and suggestions.

Betsy Ahl, SBI Inc., for a copyedit that once again made my grammar look like it ain't all that bad!

Susan Glinert, BookMakers, for a page layout that made the book look as good as it does.

Carole McClendon, Waterside Productions, for making this book happen.

Betty Sydow, for encouragement and for displaying copies of my books on her bookshelf—even if she can't imagine what they're all about! Thanks mom.



Introduction

These days, it's hard to hold a conversation about computer programming without the topics of C++ and object-oriented programming surfacing. You've surely heard such opinions as "C++ is the programming language of the nineties" and "the future of programming is C++." While many will argue the notion that C++ is the *most worthy* recipient of the title "the future of programming," one simple fact stands out—it *will be* the dominant language for at least the next several years.

You are probably aware that C++ is here to stay. Yet you might have resisted the switch from your current language—Pascal or C, perhaps—to C++. If that is indeed the case, you have probably hesitated because you've heard that the C++ language is a complicated one to learn. You've probably heard the same said about object-oriented programming—the set of programming techniques that define how a C++ program is organized. Before you turn this page, I'd like you to cast aside your fears and keep one thought in mind—C++ and object-oriented programming aren't difficult; they're *different.*

This book and the disk that accompanies it exist to eliminate the frustration that many people feel as they attempt to make the transition from a *procedural* language like C to an *object-oriented* language like C++. The book starts with a series of very small and simple examples designed to teach the

basics of C++. This step-by-step approach builds a foundation for the more comprehensive examples that appear later in the book—examples that use C++ and object-oriented techniques to create Mac programs that include menus and windows.

All of the source code for the examples in the book are provided on the included disk. The disk also holds a software tutorial called Simulator C++ that is meant to accompany the text. While the book contains plenty of figures to help clarify programming concepts, there are times when ideas can be better presented using a little animation. The Simulator C++ software does just that. In Simulator C++, screens of information correspond to the pages of a book. You click a button to page through them. And, quite often, a page will open up a supplementary window that contains a QuickTime movie. You can play and replay this movie to "bring to life" a key programming concept.

Whom This Book and Software Are For

Symantec C++: Object-Oriented Programming Fundamentals for the Macintosh was written for C programmers who want to move on and move up to C++ programming. The book is ideal for people who fall into one or more of the following categories:

- People who have programmed the Mac using C and now want to program it using C++.
- People who have the combined Symantec C++/THINK C compiler but haven't taken advantage of the C++ features of it.
- People who know a little about the C++ lanaguage but don't know how to use it to write a Mac program.
- People who want to learn how C++ is used to write object-oriented programs (OOP).

What You Need

The *Symantec C++: Object-Oriented Programming Fundamentals for the Macintosh* package is a comprehensive Macintosh C++ programming guide, but there are a couple of things that will help you to get the maximum benefit from it.

Using the Book

The C++ language is based on C. So it makes sense that you should know the basics of C before trying to tackle C++. If you've programmed the Mac but not in C or you've never written a Mac program, you might want to consider getting a copy of *Think THINK C!*, also published by Prima Publishing. If you have programmed the Mac using C but feel a little rusty, then take a close look at the material in Chapter 3.

Running the Simulator C++ Software

QuickTime is Apple's movie-playing system software extension. Because the Simulator C++ tutorial software included with this book uses QuickTime, you'll need a Macintosh computer capable of running it. QuickTime requires a Mac that has a 68020, 68030, 68040, or PowerPC microprocessor. That shouldn't be a problem; every Mac made in the past several years has one of those chips. Only a few of the older Macintosh models *don't*—the Mac Plus, the Mac SE, the Mac Classic, and the PowerBook 100. Every other model, including all those listed in Table 1, is capable of running QuickTime—and the Simulator C++ software.

If you have one of the Macs listed in Table I–1 and version 6.0.7 or later of the Macintosh operating system—including any version of System 7—you're all set. Just make sure you have a copy of the QuickTime extension in the Extensions folder in your System Folder. If you don't, refer to Appendix A. There you'll find out how to get a free copy of QuickTime and how to install it. You do *not* need a compiler to run the Simulator C++ software.

TABLE I–1 Macintosh models that can run QuickTime and the Simulator C++ software

Any LC	II
SE/30	IIx
Classic II	IIcx
Any Quadra	IIfx
Any Performa	IIvx
Any Duo	IIci
Any Centris	IIsi
Any PowerBook except the 100	IIvi
Any Power Macintosh	

Running the Example Programs

In addition to the Simulator C++ tutorial software, the included disk contains about two dozen example programs. Many are very small and demonstrate just a single C++ concept. Others are much larger and demonstrate how a real-world Macintosh C++ program is written. If you want to examine the code and experiment with the examples, you'll need a copy of the Symantec C++ compiler. The best way to learn to program is to work with examples, so the purchase of a C++ compiler is a very sound investment. All the code in this book and on the disk works with either the Symantec C++ 7.0 compiler or the older Symantec C++ 6.0.

How to Use This Book and Software

The disk that accompanies this book contains example C++ source code and a software program called the Simulator C++. The example code and the Simulator C++ program are separate items that do not depend on one another. Figure I–1 highlights this point.

Using the Symantec C++ Examples

If you own the Symantec C++ compiler, you can use the Think Project Manager that is part of that package to view, edit, compile, and run the example source code that is in the Symantec C++ Examples folder. Chapter 1 provides step-by-step instructions for doing this.

Using the Simulator C++ Software

The Simulator C++ program is a software tutorial that accompanies this book. If object-oriented programming and the C++ language are taught in this book, why also include a software tutorial? Because Prima Publishing believes that every person has a different style and pace of learning. Many people find that some programming concepts are best learned through interactive study—the method that the Simulator C++ software uses.

The Simulator C++ program uses the analogy of a book to teach you how to program your Macintosh using the C++ language. Screens are referred to as pages and can be flipped through much as you would the pages in a book. Pages are grouped into chapters and topics that correspond

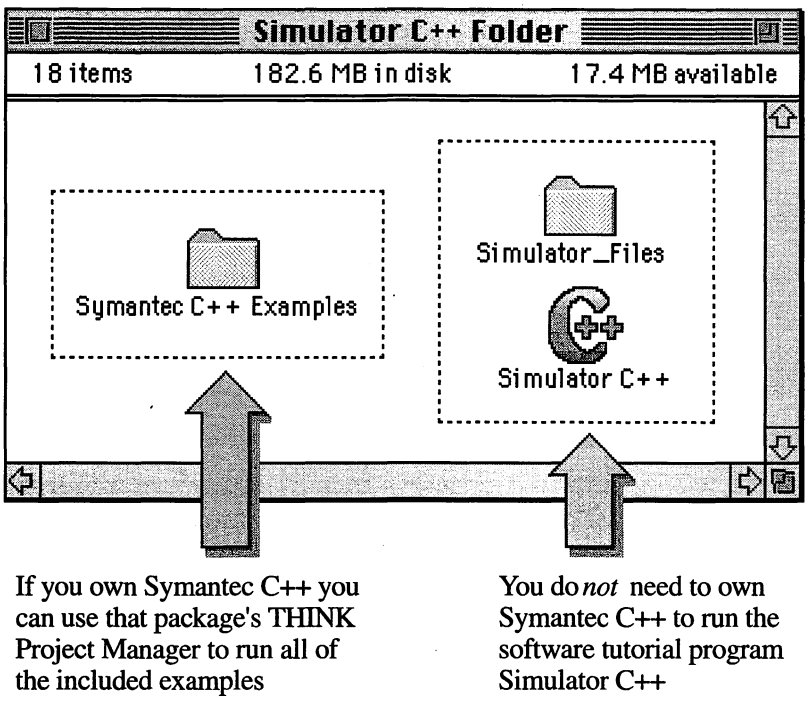


FIGURE I–1 Example programs work with Symantec C++. The Simulator program runs on its own.

to those found in this book. That makes it easy for you to use both the book and the software to study a single topic.

If you're using the software to study a topic and you want to refer to that same topic in the book, just note the chapter and topic names in the software; you'll find a corresponding chapter and topic in the book. To reverse this process and go from the book to the Simulator C++ program, look in the Simulator's pop-up Table of Contents menu to find the corresponding topic.

The Simulator software contains the same programming topics found in the book, but it is not just on-screen duplication of the book's text. Instead, the software covers the material in a way not possible in the printed pages of a book. The tutorial software uses the animation of QuickTime movies to present material in a dynamic way. It also constantly tests your knowledge by asking you questions, and it provides immediate feedback to let you know what areas you need to concentrate on.

The following are a few tips for using the the book and software package. Start by running the Simulator software. Then,

- open the book to the same topic. It offers different wording, a different figure, or a different piece of source code than that found in the Simulator.
- if you want to take notes, mark up the book as you view on-screen pages in the Simulator.
- if you're away from your Macintosh, take the book with you!

Features of the Simulator Software

The Simulator software uses a friendly, interactive approach to teaching you the many concepts needed to write Macintosh programs using the C++ language. Among the features of the software are the following:

- **A pop-up Table of Contents**, which lets you move to any topic at any time.
- **Highlight words**, which can optionally be clicked on to get supplemental information about a topic. These boldfaced words provide an additional layer of learning. When you encounter a word in boldface type, just click on it to open a window that contains more background information.
- **Movie pages**, which provide on-screen animation to bring to life difficult concepts that just can't be clearly explained on the static pages of a book. Each movie page holds a QuickTime movie that can played as often as you like.
- **Question pages**, which constantly test your knowledge and provide helpful feedback.
- A **Status page** at the end of each chapter topic to let you know how well you've answered the questions posed in the current topic. From the Status page you'll be returned to skipped and missed questions so that you can consider a topic mastered before leaving it.

Installing the Simulator C++ Software

The Simulator software is easy to install. Follow the steps provided here to get your Simulator C++ program up and running.

The Simulator software comes in a single compressed file on a single 1.4MB disk, which you will find in a pocket at the back of this book. The many individual files that make up the Simulator tutorial, as well as the Symantec C++ source code examples, have all been compressed into one file to save disk space. This file is *self-extracting.* This means that you do not have to own any special program to decompress the files back to their original sizes.

You'll want to copy the single compressed file to your hard drive before decompressing it. This serves two purposes: it makes the decompression run smoothly, and it allows you to work with a copy of the file, thus preserving the original file and disk for backup.

You can copy the single compressed file directly to the hard drive of your Macintosh; you don't have to create any new folders. After copying the file, you'll have a file titled Simulator.sea on your hard drive, as shown in Figure I–2. Of course, because the other programs and folders on your hard disk are different from mine, your hard disk folder won't look exactly like the one pictured.

The next step is to decompress the file. Decompressing the file extracts all the original files and programs that are currently combined in the one compressed file. Double-click on the Simulator.sea file. You'll be presented with a dialog box like the one pictured in Figure I–3. This dialog box asks you to specify where you want the extracted files to be placed. The Simulator.sea file has been created in such a way that the extraction process will

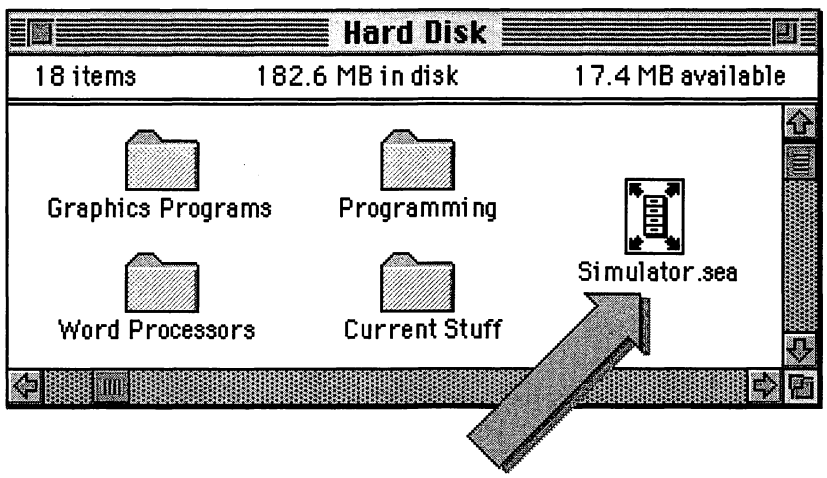


FIGURE I–2 The compressed file on your hard drive

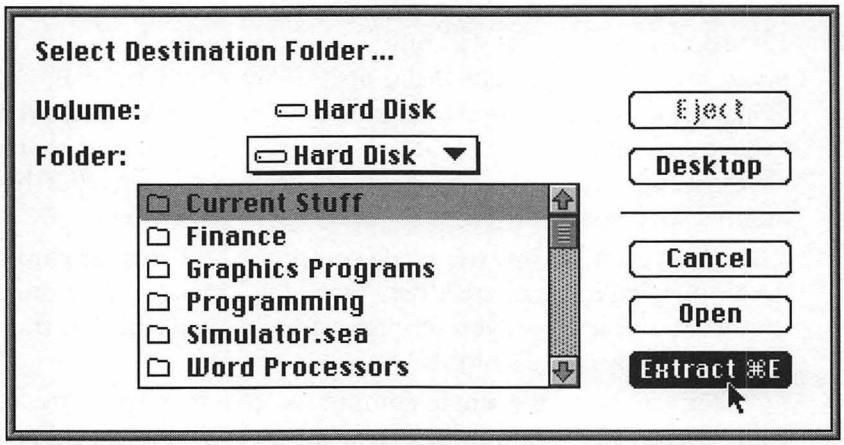


FIGURE I–3 The Extraction dialog box

automatically create the necessary folders. That means you need specify nothing at this dialog box. It doesn't matter what folder or filename happens to be highlighted in the Extraction dialog box. Simply click on the Extract button, as shown in Figure I–3. Again, your list of folders and files will not match ours.

NOTE

If you already have the Simulator C software from Prima Publishing's *Think THINK C* book on your hard drive, don't be alarmed. This new Simulator C++ software package will not delete or overwrite any of the folders or files associated with that software.

After clicking on the Extract button, you have nothing to do but sit back and watch; the extraction processes runs on its own. You'll see a dialog box—pictured in Figure I–4—that marks the progress of the extraction.

When the extraction is complete, the progress dialog box will disappear and you will be returned to the Mac desktop. On your hard drive, you'll find one new folder, titled Simulator C++ Folder. Within this folder are two more folders and the Simulator C++ program, as shown in Figure I–5.

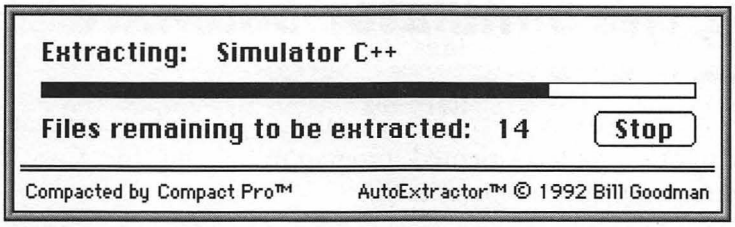


FIGURE I–4 The extraction progress indicator

The last step is to move the Simulator.sea file into your trash can. You've extracted all the files you need from this one compressed file; get rid of it and free up the disk space it occupies. If you later determine that something went wrong during the extraction process, you still have the original self-extracting file on your floppy disk, so you can repeat the process.

IMPORTANT

Keep the Simulator_Files folder and the Simulator C++ program in the same folder, as shown in Figure I–5. As the Simulator program runs, it will look for files contained in the Simulator_Files folder. It assumes that the Simulator_Files folder is right nearby.

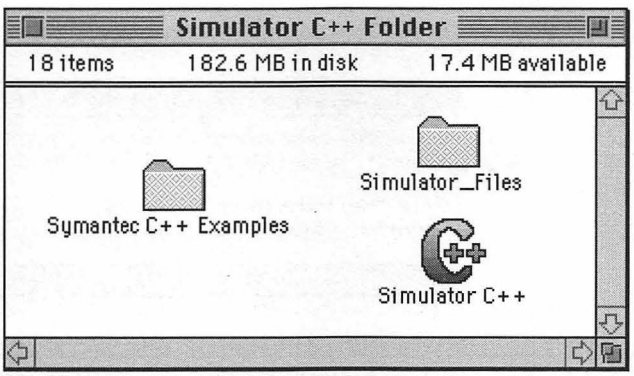


FIGURE I–5 The contents of the Simulator C++ Folder

Using the Simulator Software

The Simulator software has several easy-to-use features that help you learn Macintosh object-oriented programming and the C++ language. This section provides a look at each.

Running the Simulator C++ Software

To run the Simulator C++ program, just double-click on the Simulator C++ program icon.

Pages and the Control Panel

The Simulator software uses the analogy of a book in that screens are thought of as pages. Moving from one screen of information to another is like turning the pages of a book. As you work with the Simulator software, you'll always see two windows on the screen—the page window and the control panel window. Figure I–6 shows a typical page. Figure I–7 shows the control panel.

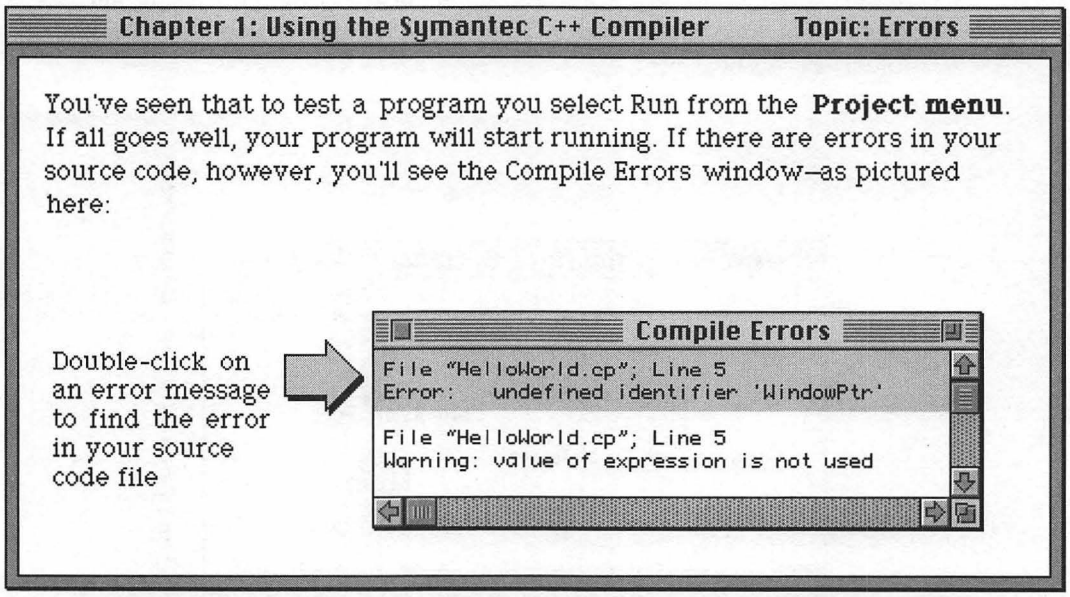


FIGURE I–6 A typical Simulator C++ page

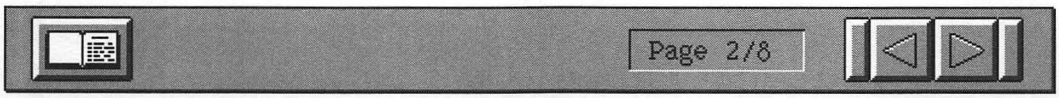


FIGURE I–7 The control panel

NOTE

The control panel pictured in Figure I–7 is a window opened by the Simulator software and is not related to the Control Panels folder that comes with your Macintosh system.

Pages are turned by clicking on the Next page or Previous page icon found on the Simulator program's control panel window. Figure I–8 shows these icons.

If you want to go back to a particular page—perhaps one that appeared much earlier in a topic—use the control panel's Back 5 icon. Clicking once on this icon moves you back five pages. Similarly, the Forward 5 icon jumps you ahead five pages with each click. Figure I–9 shows these icons.

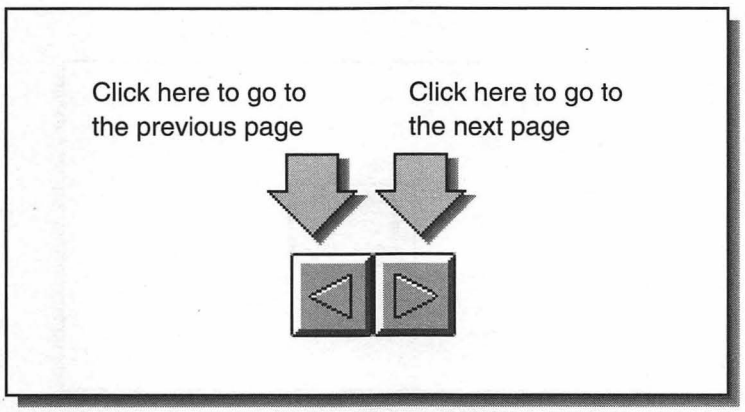


FIGURE I–8 The control panel's Previous page and Next page icons

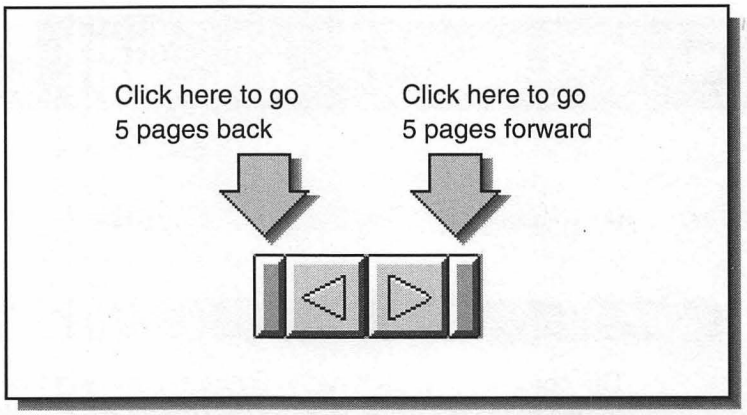


FIGURE I–9 The control panel's Back 5 pages and Forward 5 pages icons

The Simulator has one chapter for each chapter of the book. To move from one chapter to another, click on the control panel's Table of Contents icon. This icon is shown in Figure I–10.

Clicking on the Table of Contents icon displays a hierarchical menu that lists every chapter. Move the mouse to highlight any chapter title; then move the mouse to the right to display a submenu of chapter topics. Release the mouse button on a topic to start that topic. Figure I–11 shows an example.

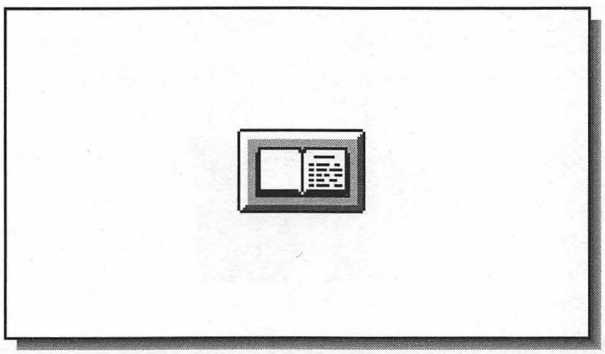


FIGURE I–10 The control panel's Table of Contents icon

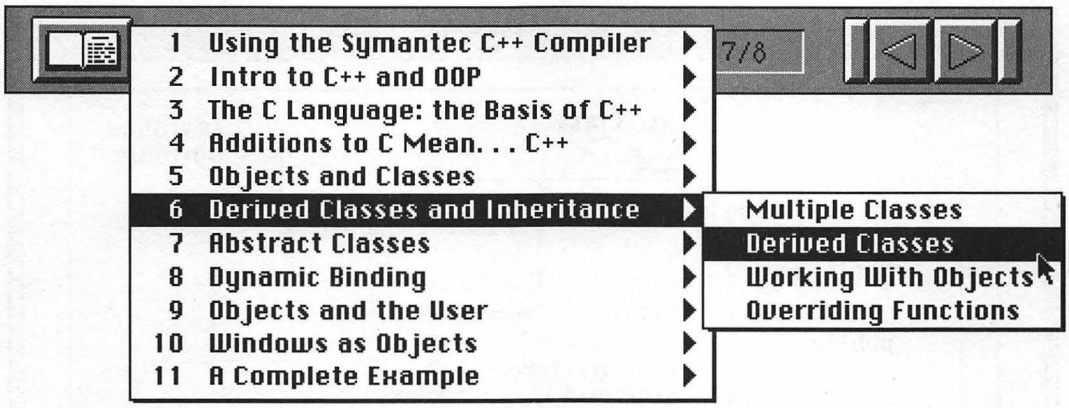


FIGURE I–11 Selecting a topic from Chapter 6 by clicking on the control panel's Table of Contents icon

When you've read each page and answered each question in a topic, that topic is said to be complete. When that happens, a check mark will appear beside the topic name. Once a topic has been marked, it will remain so—even between runnings of the Simulator program. This allows you to keep track of which topics you've completed. Figure I–12 shows that two topics from Chapter 6 have been completed.

Simulator Pages

Each page, or screen, of the Simulator may contain text, graphics, or both. Additionally, there are four special types of pages that you will occasionally come across.

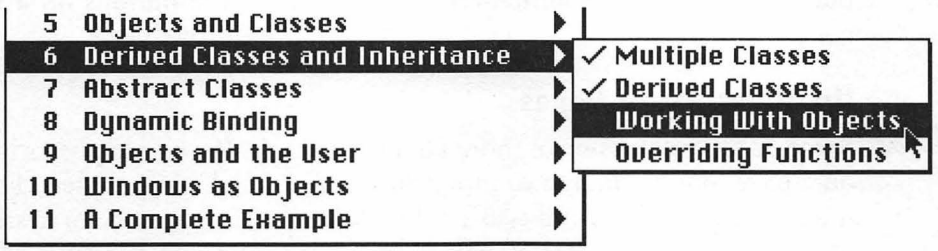


FIGURE I–12 A check mark appears beside a completed topic

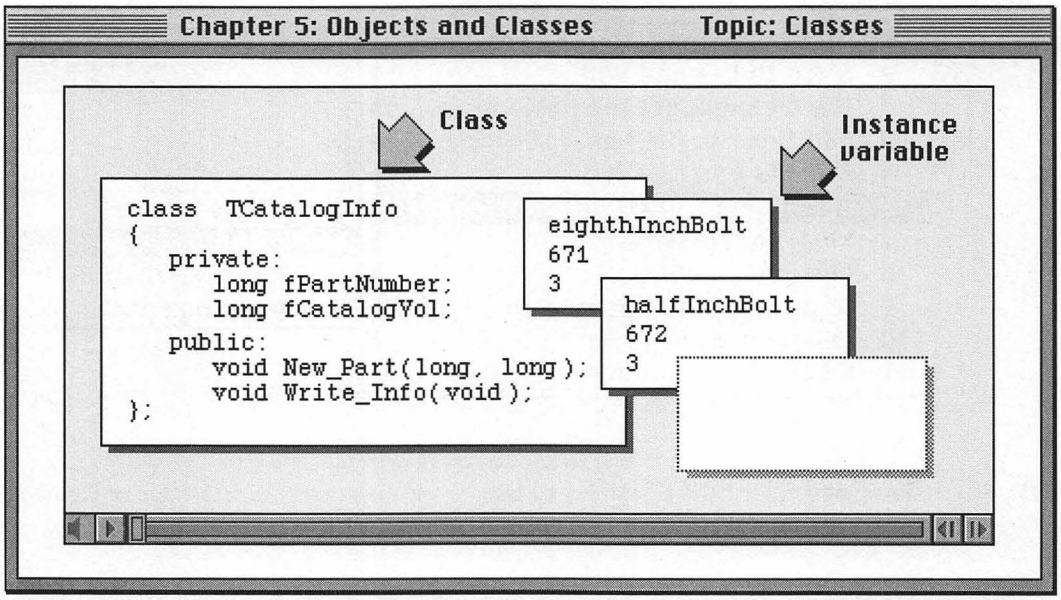


FIGURE I–13 A typical Movie page

Movie Pages

Some pages have QuickTime movies on them. Why movies? Because some topics can best be explained through the use of animation. Figure I–13 shows a typical Movie page.

You'll recognize a Movie page by the movie controller that appears on the page. The movie controller is Apple's standardized way of allowing a user to play a movie. By clicking the Play button on the controller, you can play a movie as many times as you want. If the movie goes too fast for your liking, you can use one of the Step buttons to step through it slowly. A few movies include sound. To change the volume, use the controller's Speaker button. Figure I–14 summarizes the purposes of the buttons on a movie controller.

Highlight Word Pages

Some pages contain one or more Highlight words. If a word appears in bold on a page, use the mouse to move the cursor over the word; then click the mouse button. The word will change color if you have a color system or change to an italic style if you have a black-and-white system. A new window that contains more information about the word will open. Figure I–15 shows *Project menu* as a Highlight word.

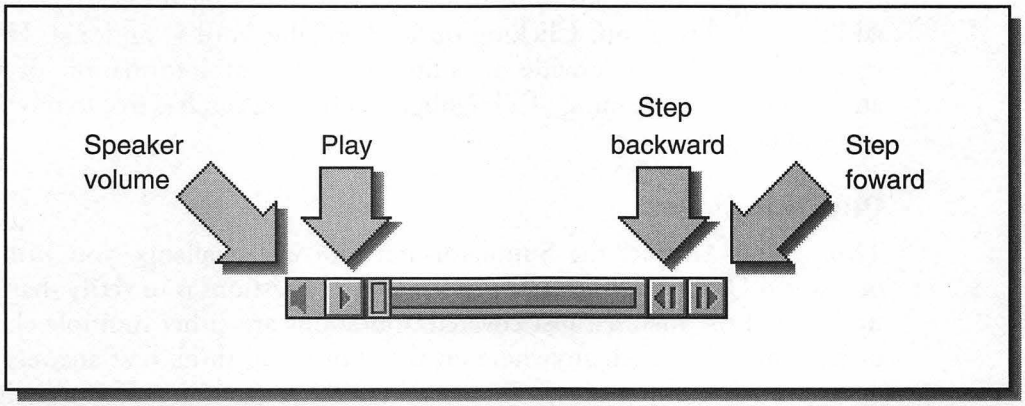


FIGURE I–14 The buttons on a movie controller

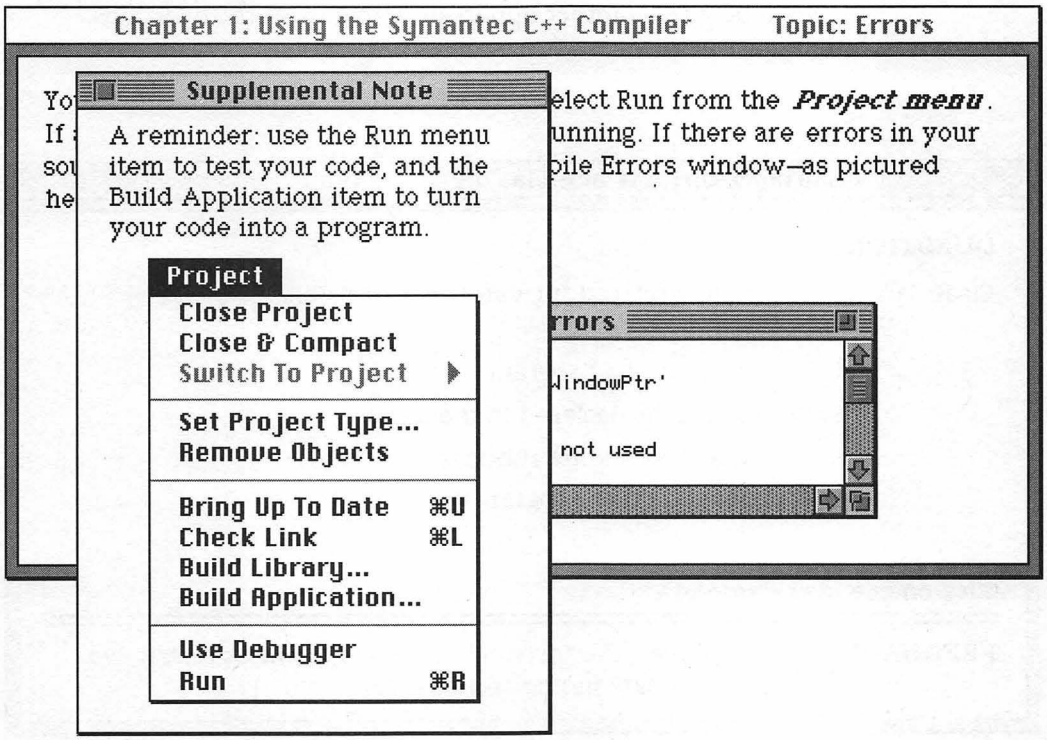


FIGURE I–15 The Highlight words *Project menu* appear on a page. The Supplemental Note opens when you click anywhere on those words.

Highlight words are a form of *hypertext;* clicking on one gives you additional information. Clicking on a Highlight word is optional. Highlight words exist to provide an additional layer of information. If you already know the meaning of a Highlight word, you can feel free to move on without clicking on it.

Question Pages

Throughout a topic, the Simulator software will challenge you with an occasional Question page. The purpose of the questions is to verify that you understand the material just covered. Questions are either multiple choice or true/false. Just click anywhere on the choice you think best answers the question. A check mark will be placed beside your choice. If you're right, you'll be congratulated. If you're wrong, you will receive helpful feedback. Figure I–16 shows a typical Question page.

You're allowed two chances to answer any one question. All subsequent choices will be ignored. Once you get to the last page of a topic, the Status page, you'll be returned to all missed and skipped questions for that topic. There you'll be given another opportunity to answer the questions.

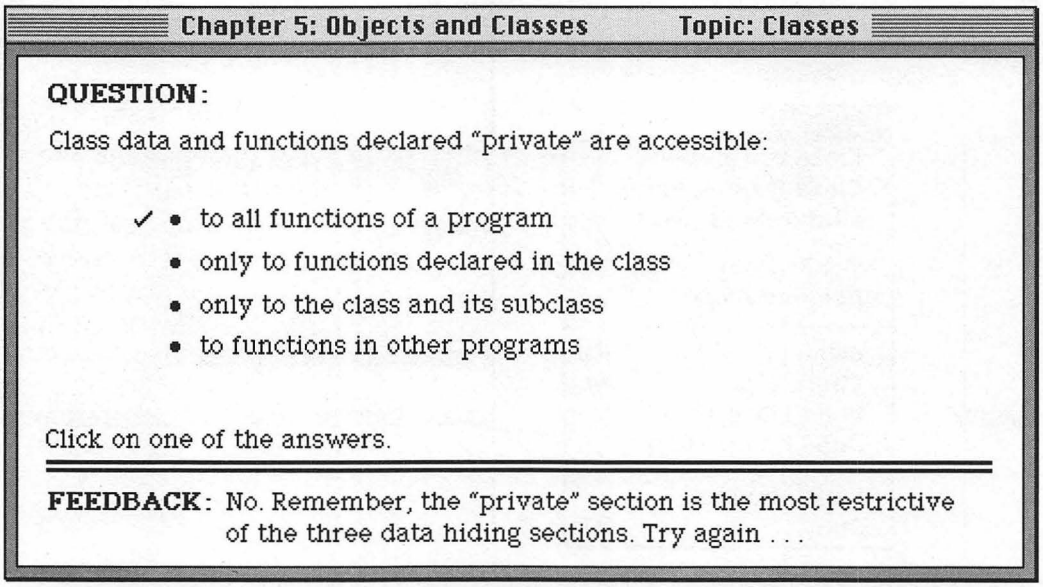


FIGURE I–16 A typical Question page with feedback

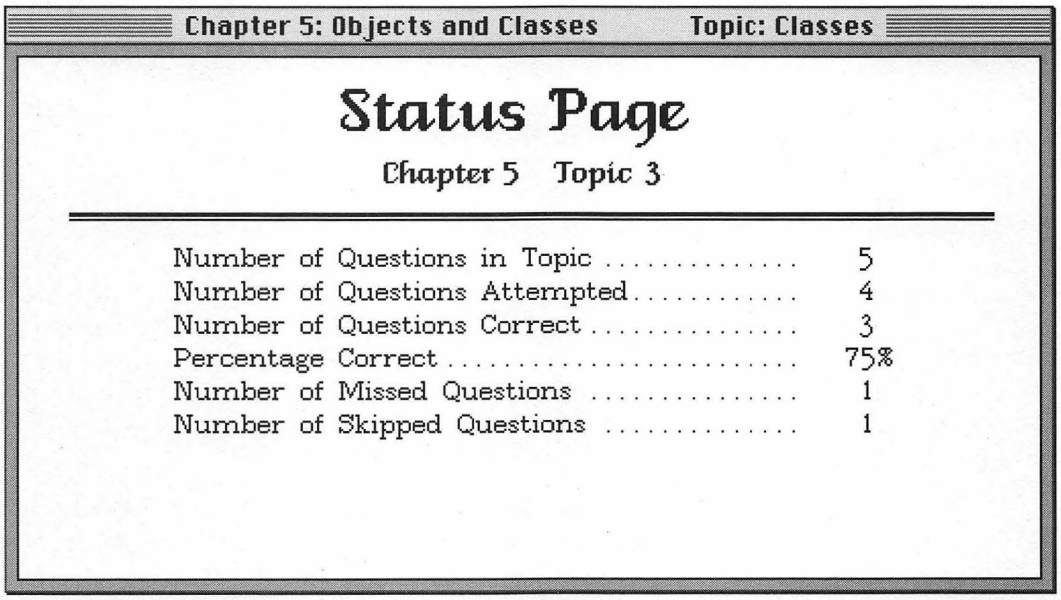


FIGURE I–17 A Status page at the end of a topic

Status Pages

At the end of every topic is a Status page. This page shows you how well you've mastered the topic. Figure I–17 shows a Status page. After giving you a few seconds to look over your score for the topic questions, the Simulator will take you back to the first missed or skipped question of the topic. There you'll be given two more chances to answer the question. If you still get it wrong, the Simulator will help you out by checking the correct answer. The purpose of a Question page is not to frustrate you, but rather to provide a review that points out your strengths and weaknesses. After finishing all missed and skipped questions, you'll be returned to the Status page. Then it's time to select a new topic from the Table of Contents icon found on the Simulator's control panel window.



Chapter 1

Using the Symantec C++ Compiler

In the world of Symantec C++, a program begins life as a project. A project organizes all of the source code that is to become a program. In this chapter, you'll see how to create a new project and how to work with a project—that is, how to add files to the project and how to compile those files. You'll also see how to turn a project into an application—a stand-alone Macintosh program.

To demonstrate how to work with a project, you'll walk through the creation of a very simple C++ application called HelloWorld. In this chapter, you'll cover each step of the process of creating a Macintosh program using the Symantec C++ compiler—from creating a new HelloWorld project to turning your work into the HelloWorld application.

Creating a New Project

The *THINK Project Manager* is the C++ compiler, source code editor, and project file organizer, all rolled into one *environment.* When you edit a

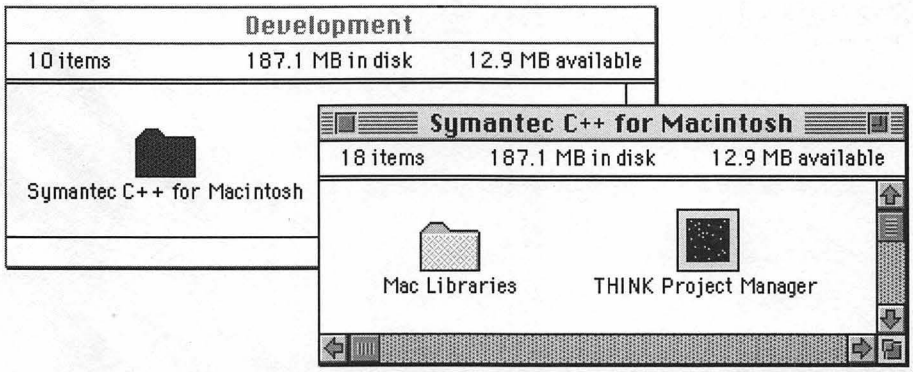


FIGURE 1–1 The THINK Project Manager icon, found in the Symantec C++ for Macintosh folder

source code file, you're using the THINK Project Manager. When you compile that same source code file, you're again using the THINK Project Manager. And the same applies when it comes time to turn your code into a Macintosh application. So while you will be using Symantec's C++ compiler, you'll be doing so from within the environment of the THINK Project Manager. That's why we'll talk so much more about the THINK Project Manager than the Symantec C++ compiler.

The THINK Project Manager can be found in the Symantec C++ for Macintosh folder, which is in the Development folder. The program's icon is shown in Figure 1–1.

NOTE

When you originally installed the Symantec software, the installer created the Development folder. If you've since renamed it, you might want to consider changing its name back to Development so that your folders match those shown in the figures in this chapter.

The THINK Project Manager uses a single project file to store the source code files, compiled code, and libraries of code that are used to create a program. The next two sections deal with the creation of a project file. If you're using version 7.0 of Symantec's C++ compiler, skip the following section and go to the next one—Creating a Project Using Symantec C++ 7.0. If you're using Symantec C++ 6.0, read on.

NOTE

If you've worked with Symantec's THINK C compiler, much of this chapter will look very familiar—so familiar, in fact, that you may be tempted to skip it. If you haven't worked with the Symantec C++ compiler, please don't give in to that temptation. There are a few topics that you'll want to make sure you take note of, such as which libraries are added to a typical Symantec C++ project.

Creating a Project Using Symantec C++ 6.0

Before starting the THINK Project Manager program, create a folder to hold the soon-to-be created project. Because this chapter will walk through the creation of a simple program named HelloWorld, I've named my folder HelloWorld ƒ. The ƒ character stands for *folder* and is created by pressing Option-f. Make sure the folder is in the Development folder, as shown in Figure 1–2. Keeping a project folder in the Development folder but not inside the Symantec C++ for Macintosh folder makes it easy for the THINK Project Manager to find the files it needs.

Next, open the Symantec C++ for Macintosh folder and double-click on the THINK Project Manager. You'll be presented with a dialog box like the one shown in Figure 1–3. Click on the New button.

Next you'll see a dialog box like the one pictured in Figure 1–4. Use the pop-up menu at the top of the dialog box to work your way into the Development folder. Then double-click on the HelloWorld ƒ folder to move inside it. Now type in the name of the project. The project name is generally the name you'll give the program, followed by the π symbol. Create the π

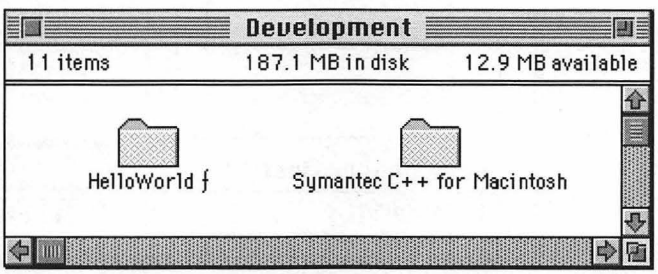


FIGURE 1–2 Keep your project folders in the Development folder.

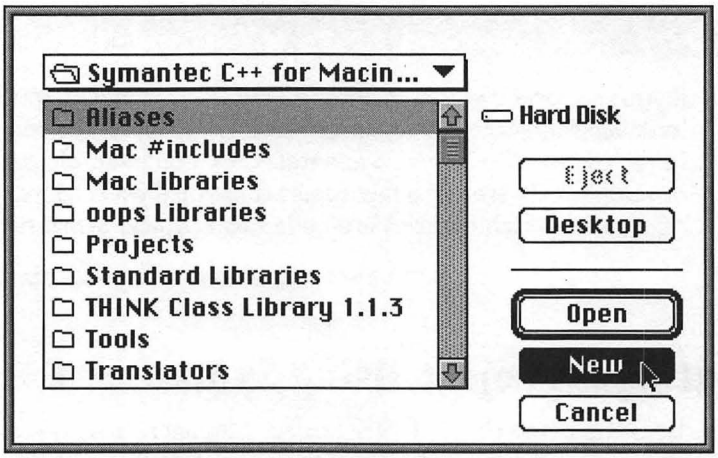


FIGURE 1–3 Creating a new project

character by pressing Option-P. After typing in the project name, click on the Create button.

The THINK Project Manager will open a new, empty project window, as shown in Figure 1–5.

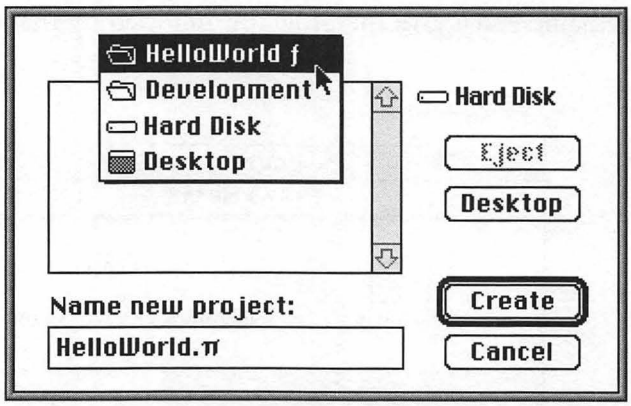


FIGURE 1–4 Creating a new project in the HelloWorld *f* folder

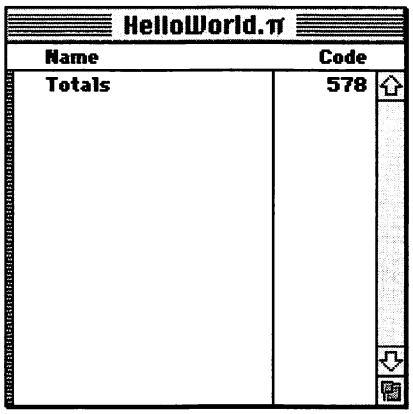


FIGURE 1–5 The empty HelloWorld project window

Now you're all set to add the necessary files to the project. Skip the next section, which discusses creating a project using the new version of Symantec's C++ compiler. Go directly to the section titled Adding Files to a Project.

Creating a Project Using Symantec C++ 7.0

Opening the Symantec C++ for Macintosh folder and double-clicking on the THINK Project Manager icon is the first step in creating a project using Symantec C++ 7.0. The first thing you'll see is the dialog box pictured in Figure 1–6. Click on the New button.

After you click on the New button, the dialog box will be dismissed and will be replaced by the one shown in Figure 1–7. Version 7.0 lets you create several different types of projects; you can see some of them in the list in the dialog box shown in Figure 1–7. In this book, you'll always be starting with an empty project and adding files to it. Click on Empty Project in the scrollable list; then click on the Create button. Be sure to leave the Create Folder check box as it is—checked.

After clicking on the Create button, still another dialog box appears, as shown in Figure 1–8. The THINK Project Manager will create a new folder in which the project file will be stored. Use the pop-up menu at the top of the dialog box to move into the Development folder, which is where you want the new folder to be placed.

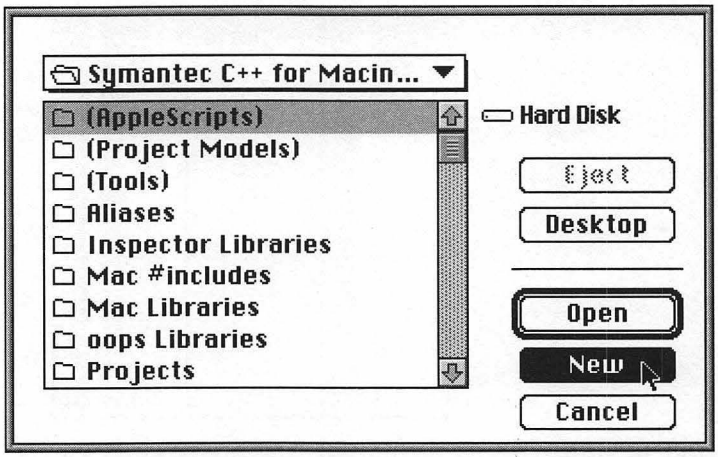


FIGURE 1–6 Creating a new project file

Now type in the name of project. The convention for naming a project is to give the project the name you'll give the program, followed by the π symbol (Option-P). After typing in the project name, click on the Save button, as shown in Figure 1–9.

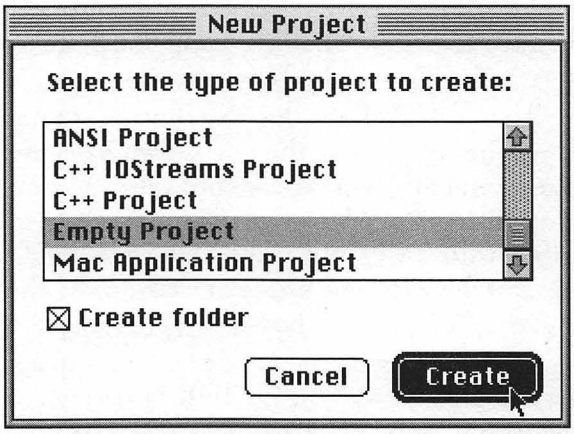


FIGURE 1–7 Selecting the Empty Project as the type of new project to create

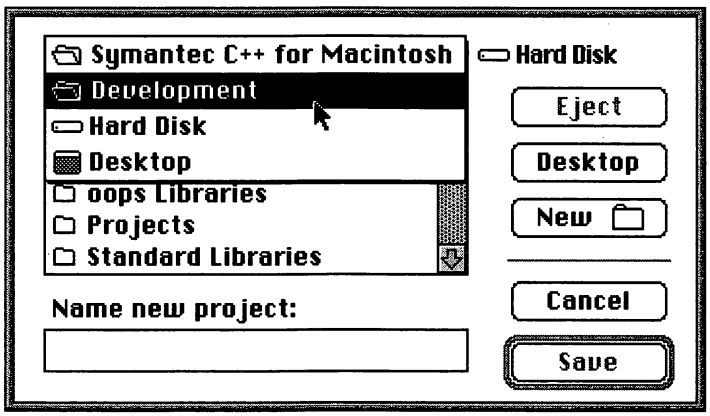


FIGURE 1–8 Moving into the Development folder in preparation for creating the new project

When you click on the Save button, a new, empty project window, like the one shown in Figure 1–10, will open.

Before adding any files to the new project, you might want to take a look at your hard drive's main folder—the one that opens when you double-click on the hard drive icon. When you installed your Symantec compiler, a

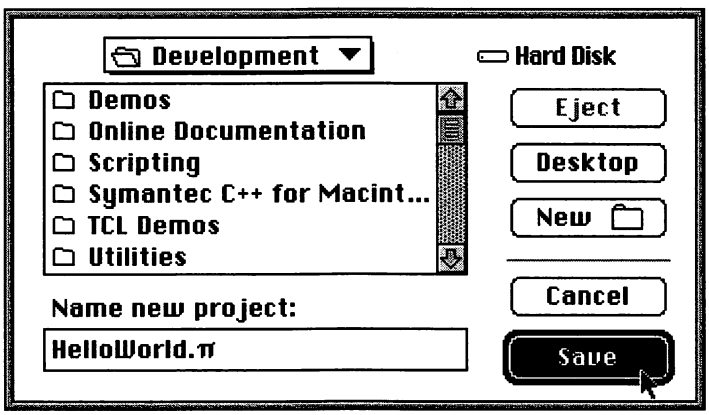


FIGURE 1–9 Naming and saving a new project

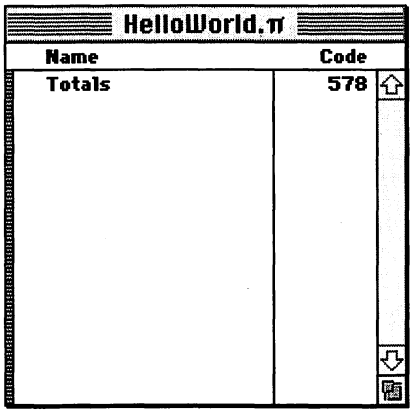


FIGURE 1–10 The empty HelloWorld project window

folder named Development was placed in that folder. In the Development folder you'll see a new folder titled HelloWorld ƒ, as shown in Figure 1–11. That's the folder that the THINK Project Manager created. When you create a new project, the THINK Project Manager creates a new folder and gives it the same name you have given the project. But instead of the π symbol, the folder has the ƒ symbol after its name.

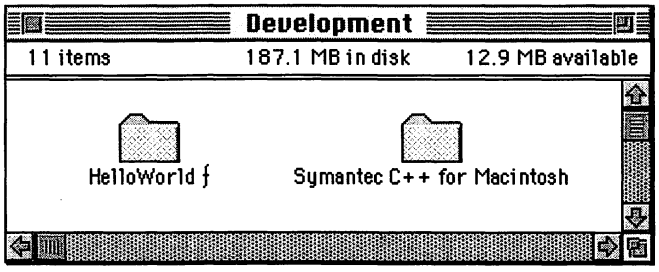


FIGURE 1–11 The new HelloWorld ƒ folder created by the THINK Project Manager

Adding Files to a Project

A project file is the THINK Project Manager's way of organizing all the code that will be used for one program. The THINK Project Manager allows you to easily add the files you'll need to create a Macintosh program.

Adding Libraries to a C++ Project

The THINK Project Manager organizes not just the source code *you* write but certain functions written by *others*—namely, Apple and Symantec. This source code—already compiled for you—appears in *libraries.* A library consists of precompiled code that is ready to use as is. For a Symantec C++ project, you'll always want to add the same three libraries to your project—MacTraps, ANSI++, and CPlusLib. The MacTraps library lets your programs access Toolbox functions such as GetNewWindow(). The ANSI++ library contains the precompiled code for standard ANSI functions, and the CPlusLib library contains routines used by the Symantec C++ compiler to create and delete objects. You'll learn about objects and object-oriented programming later in this book.

Now that you know what you need to add...go ahead, add it. Select Add Files from the Source menu, as shown in Figure 1–12.

When you select Add Files, you'll see the dialog box pictured in Figure 1–13. This figure also shows the associated pop-up menu. Use this menu to move into the Symantec C++ for Macintosh folder. From the pop-up menu in Figure 1–13, you can see the path you need to traverse to get to the Mac Libraries folder. Once there, double-click on the Mac Libraries folder; that's where the MacTraps library file can be found. Figure 1–13 shows where the Mac Libraries folder is located.

The Mac Libraries folder is in the Symantec C++ for Macintosh folder, which is, in turn, in the Development folder. Once in the proper location, click on the MacTraps name in the list and then click on the Add button, as is being done in Figure 1–14.

Clicking on the Add button moves the selected file to the list at the bottom of the dialog box, as shown in Figure 1–15.

The Add Files dialog box allows you to add more than one file at a time. So before dismissing the dialog box, add the other two libraries that you'll want in a C++ project—ANSI++ and CPlusLib. Both of these libraries are located in the Standard Libraries folder. Use the pop-up menu in the dialog box to move back to the Symantec C++ for Macintosh folder.

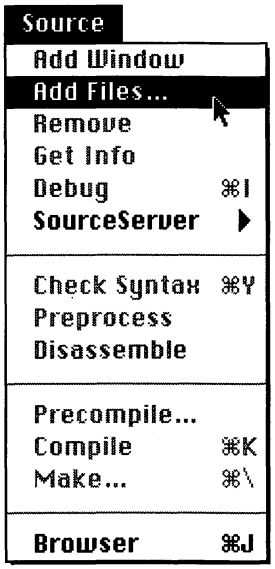


FIGURE 1–12 Selecting the Add Files menu item from the Source menu

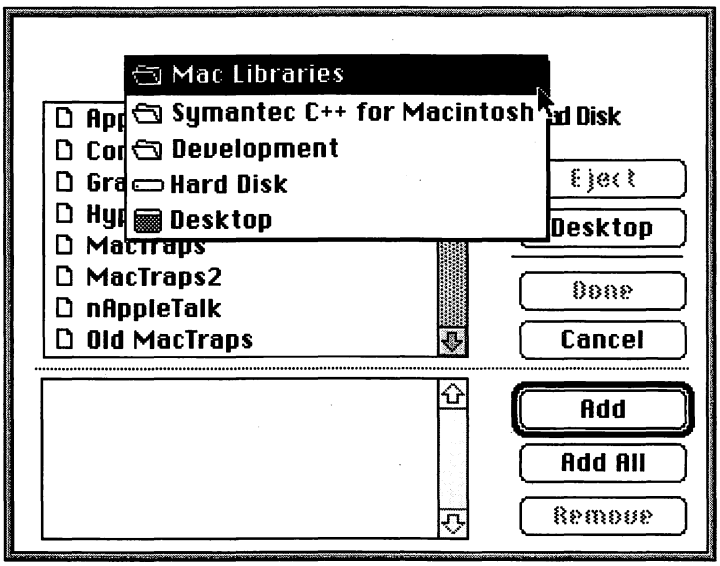


FIGURE 1–13 Maneuvering into the Mac Libraries folder

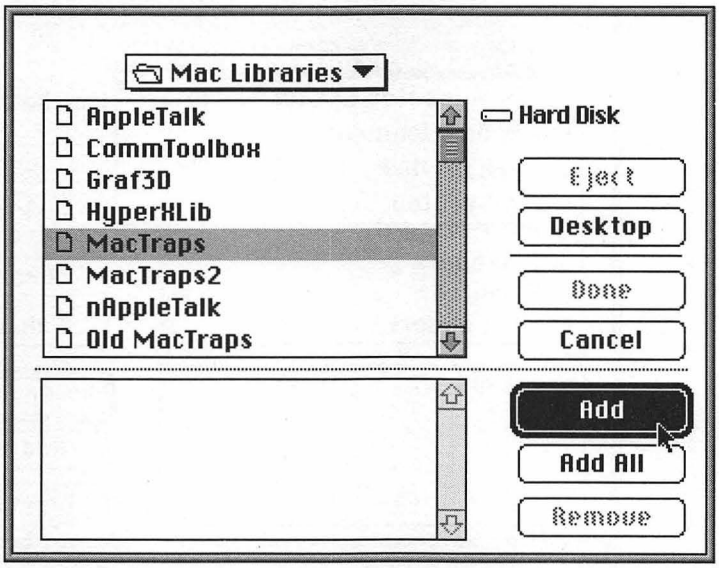


FIGURE 1–14 Adding the MacTraps library to the project

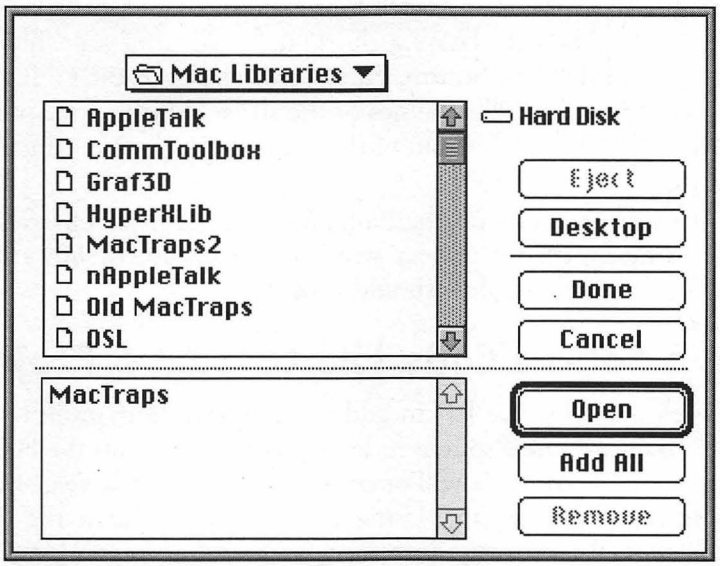


FIGURE 1–15 The added file appears in the list at the bottom of the Add Files dialog box.

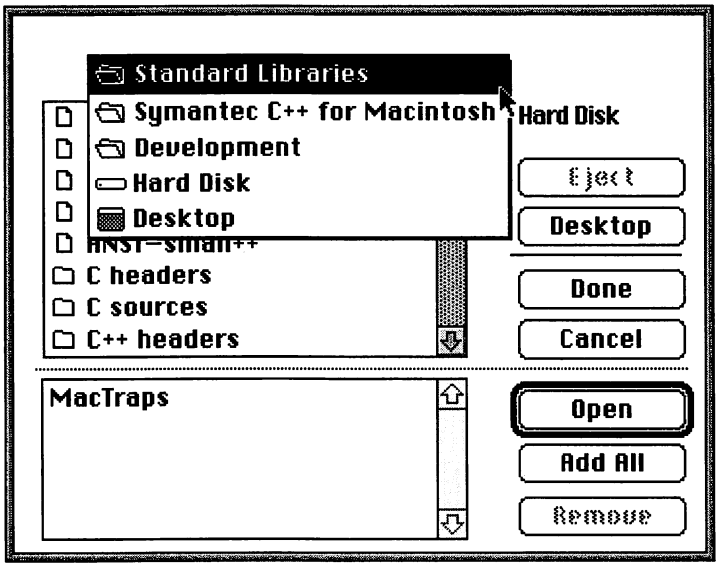


FIGURE 1–16 Moving into the Standard Libraries folder

Once there, double-click on the Standard Libraries folder. Figure 1–16 shows the path that was traversed to get into the Standard Libraries folder.

Once in the Standard Libraries folder, double-click on the ANSI++ filename in the list. That's a shortcut for clicking on a filename and then clicking on the Add button. Next, scroll down to the CPlusLib library and double-click on it. The names of the three libraries your project needs will now be listed at the bottom of the dialog box. Click on the Done button, as shown in Figure 1–17.

After you click on the Done button, the three libraries will appear in the formerly empty project window. Figure 1–18 shows how your HelloWorld project window should now look.

Adding a Source Code File to a C++ Project

There's only one file left to add to complete your project—a source code file. To create a new source code file, select New from the File menu. A new, empty source code file will open. Name the file right away by selecting Save As from the File menu. Using the pop-up menu in the dialog box that appears, make your way into the Development folder. Once there, double-click on the HelloWorld f folder. Now you're ready to name and save the new file. Type **HelloWorld.cp** and click on the Save button. Figure 1–19

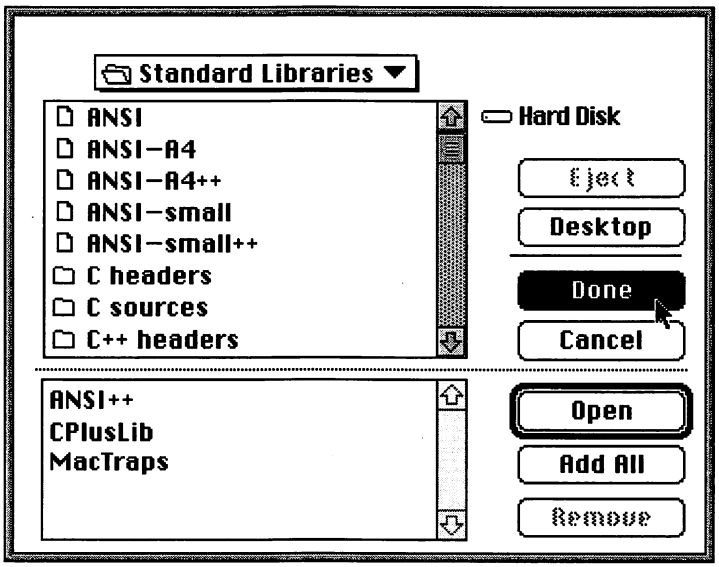


FIGURE 1–17 The Done button completes the addition of the listed files to the project.

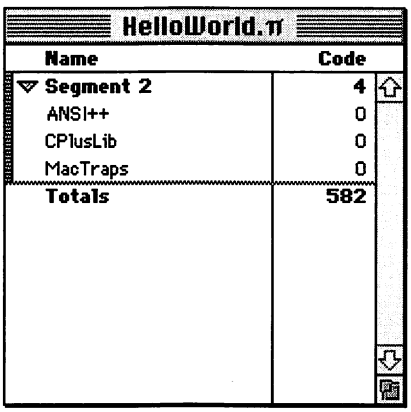


FIGURE 1–18 The project window after the addition of the three library files

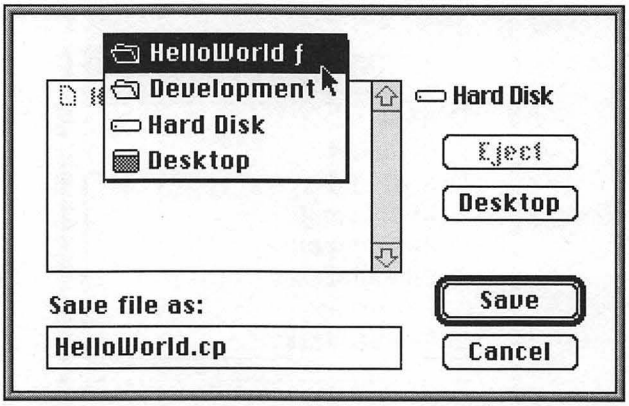


FIGURE 1–19 Naming a source code file

shows the path to your HelloWorld f folder and the name that the new file is being given.

The THINK Project Manager recognizes a file as a C++ source code file if it ends with the extension .cp or .cpp. For this example and for all others in this book, .cp will be used. Be sure to give all of your source code files this extension.

Your source code file, shown in Figure 1–20, now has the name HelloWorld.cp in its title bar. Even though it's still empty, you'll want to add it to the project; simply creating a new file doesn't automatically place it in the

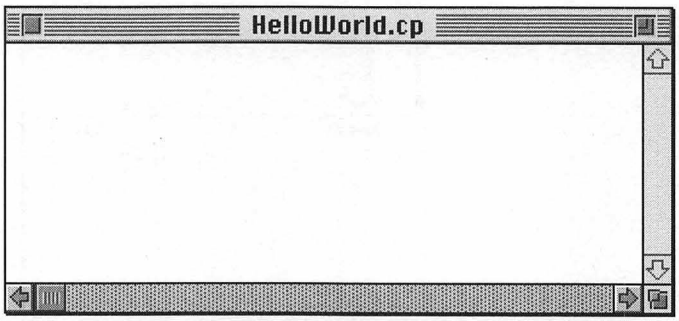


FIGURE 1–20 The new, empty source code file with the name in its title bar

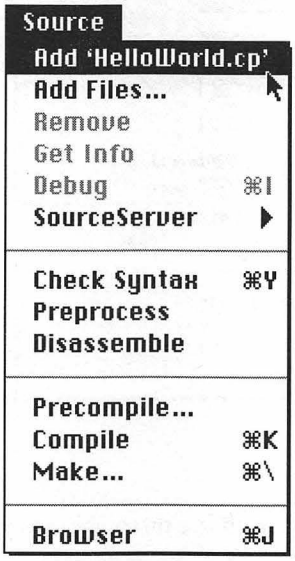


FIGURE 1–21 Adding the new source code file to the HelloWorld project

project. To do this, select the very first menu item in the Source menu—Add 'HelloWorld.cp', which is shown in Figure 1–21. This menu item will appear dim after the file has been added to the project. Once you create and save a new file, however, the menu item is enabled and takes on the name of the file. This menu item is simply a shortcut to adding the source file to your project—it saves you the work of using the Add Files menu option.

Segmenting a Project

Figure 1–22 shows how your HelloWorld project window now looks. Notice the name—Segment 2—next to the inverted triangle. To help the Mac operating system work with large programs, Macintosh programs are divided into segments, each of which can be no larger than 32K. Since the ANSI++ library is close to 30K all by itself, it should be obvious that this project requires more than one segment. To move a file into a new segment, click on its name. With the mouse button still pressed, drag the file below the word Totals in the project window. Then release the mouse button. In Figure 1–23 the ANSI++ library is being moved into its own segment. On

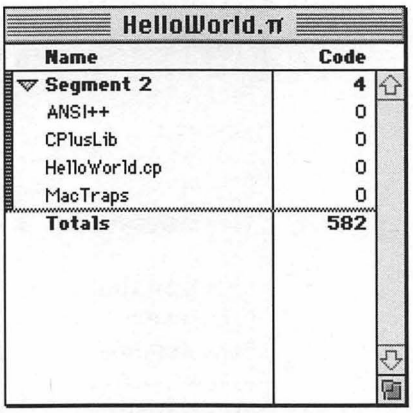


FIGURE 1–22 The project window with the three libraries and one source code file in it

the left, you see the file being dragged to the bottom of the window. On the right, you see the new segment that is created by this action.

Next, drag the CPlusLib library until the mouse is over the ANSI++ library name; then release the mouse button. That places the CPlusLib library in the same segment as the ANSI++ library. The final segmentation of the project is shown in Figure 1–24.

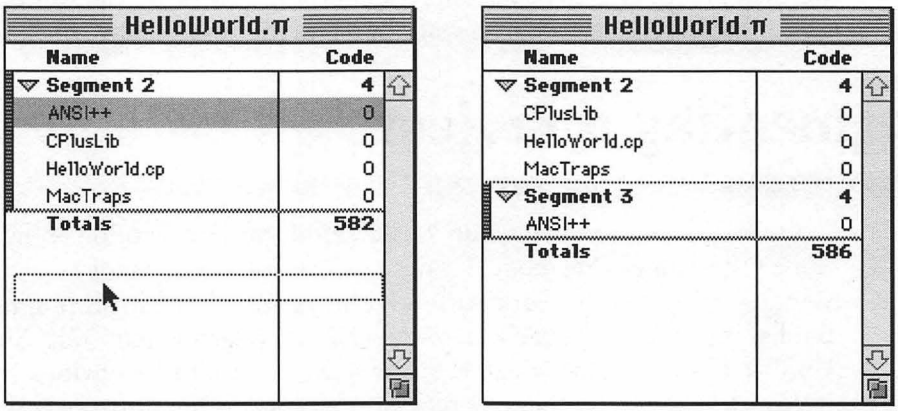


FIGURE 1–23 Moving the ANSI++ library into a new segment

C++ Code: Writing It and Running It

With the preliminaries out of the way, it's time to write a simple C++ program. If you've closed the empty HelloWorld.cp source code file, open it now. You can do that by double-clicking on its name in the project window. When the file is open, type in the short program that is shown in Figure 1–25. Type carefully and compare your file to the one pictured here.

After typing the code, select Save from the File menu. To test it, select Run from the Project menu, as shown in Figure 1–26.

Choosing Run presents you with the dialog box shown in Figure 1–27. Here the THINK Project Manager wants you to verify that it is okay for it to go ahead and compile your source code and to make the added libraries a part of your project by loading them. Click on the Update button to let the THINK Project Manager do its thing.

The simple program you typed in opens a window and writes the words "Hello, World!" to it. If all went well, you'll see a window like the one pictured in Figure 1–28. If you see a different window—one that lists an error message—refer to Appendix B at the end of this book. That appendix lists common mistakes and ways to correct them.

To end the HelloWorld program, click the mouse button. That closes the window and returns you to the THINK Project Manager environment. Before you ran your program, the right side of the project window showed a

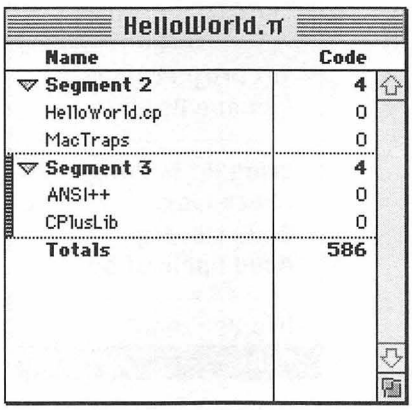


FIGURE 1–24 The segmented HelloWorld project

HelloWorld.cp

```
void  main( void )
{
   WindowPtr  the_window;
   Rect       window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect, "\pNew Window", true,
                           noGrowDocProc, (WindowPtr) -1L, true, 0 );
   SetPort( the_window );

   MoveTo( 20, 30 );
   DrawString( "\pHello, World!" );

   while ( !Button() )
      ;
}
```

FIGURE 1–25 The HelloWorld C++ source code

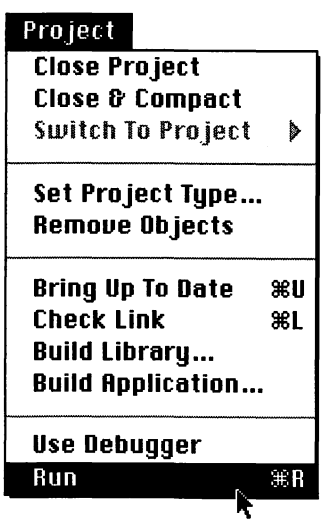


FIGURE 1–26 Selecting Run from the Project menu to give the code a test run

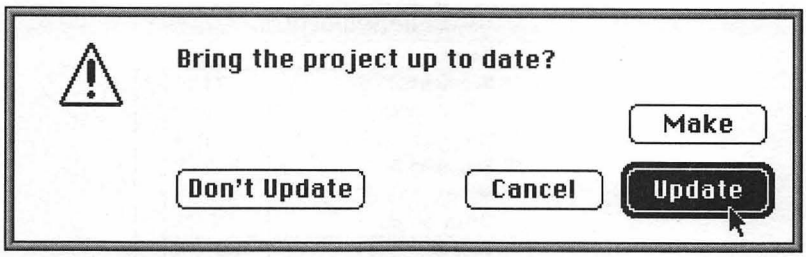


FIGURE 1–27 Telling the THINK Project Manager to update the project

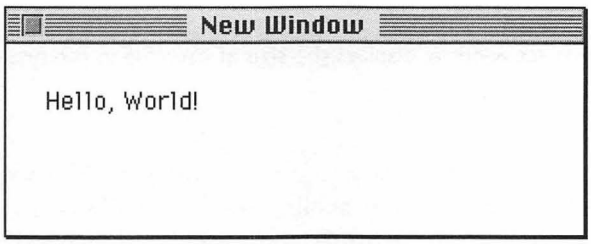


FIGURE 1–28 The result of running the code in the HelloWorld project

column with several numbers—many of them zeros—in it. Take a close look at the project window, shown in Figure 1–29, now. Notice that each filename now has a new number to the right of it. That number represents the size of the compiled code from that file. The size of a source code file isn't known until it has been compiled, and the size of a library isn't known until it has been loaded. Selecting Run from the Project menu accomplishes these tasks.

In Figure 1–29, notice that both segments have a total size of less than 32K—as the THINK Project Manager requires.

Using the Symantec Debugger

Programming errors, or bugs, creep into the programs of every developer. The best way to find the cause of an error—or to just see what's going on as a

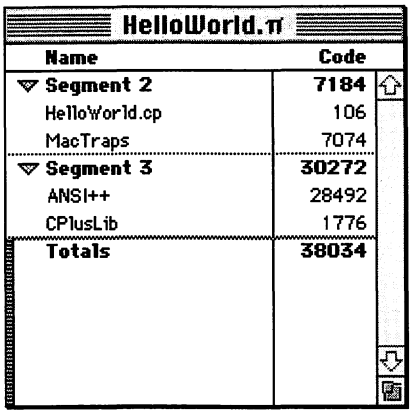


FIGURE 1–29 The project window displays the size of each file in the project.

program executes—is to use a debugger. Many people shy away from debuggers because they assume they need a knowledge of assembly language to understand how memory and variable values are displayed. This is certainly true for some debuggers—but not for the one included with your Symantec C++ package.

Debugger Basics

The Symantec source debugger works behind the scenes; you don't run it from the desktop as you do other software applications. All you do to activate it is select Use Debugger from the Project menu. Figure 1–30 shows that menu item.

Selecting Use Debugger doesn't do anything noticeable—not just yet. The debugger doesn't run until you select Run from the Project menu. Then two windows open—the Source window and the Data window. The Source window displays your project's source code. The title bar of the Source window displays the name of the source code file that's about to execute. The Data window allows you to enter the names of variables you want to monitor as the program executes. The title bar of the Data window displays the word Data. These two windows are shown in Figure 1–31.

To familiarize you with the debugger, I'll walk through a very short program called DebuggerDemo, which is included on the accompanying disk. The source code listing for DebuggerDemo appears below. Figure 1–32 shows the results of running the program.

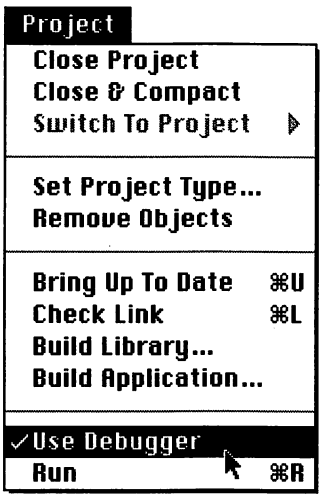


FIGURE 1–30 Turning the debugger on in Symantec C++

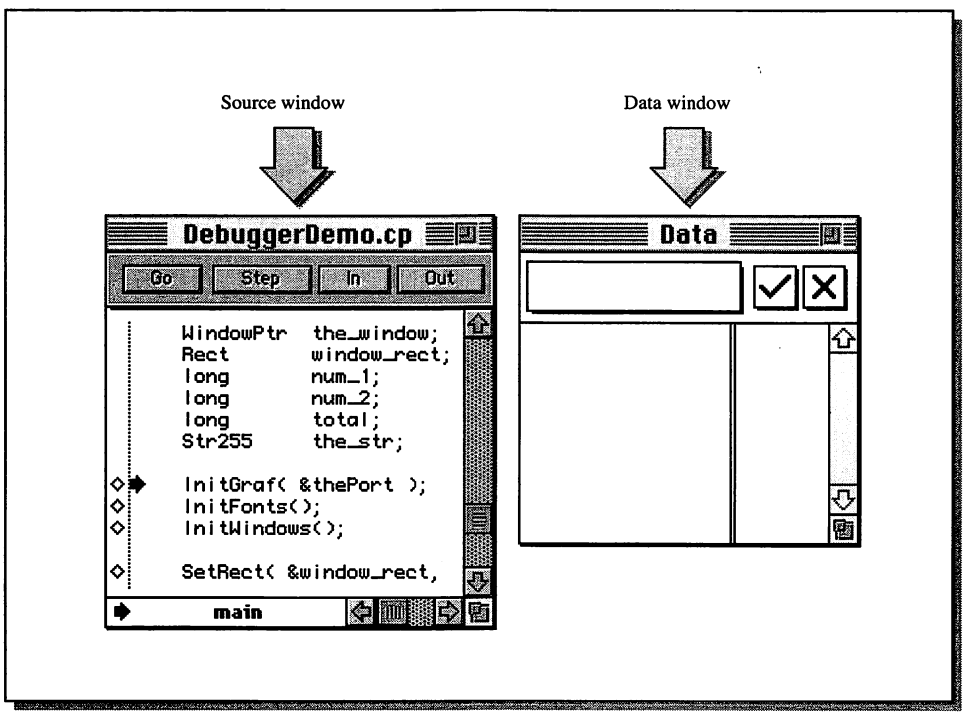


FIGURE 1–31 The Source window and the Data window of the Symantec debugger

```
void main( void )
{
  WindowPtr the_window;
  Rect window_rect;
  long num_1;
  long num_2;
  long total;
  Str255 the_str;

  InitGraf( &thePort );
  InitFonts();
  InitWindows();

  SetRect( &window_rect, 50, 50, 350, 150 );
  the_window = NewWindow( 0L, &window_rect,
                          "\pNew Window", true,
                        noGrowDocProc, (WindowPtr)-1L, true, 0 );
  SetPort( the_window );

  num_1 = 3.2;
  num_2 = 6.8;
  total = num_1 + num_2;

  NumToString( total, the_str );

  MoveTo( 20, 30 );
  DrawString( the_str );

  while ( !Button() )
;
}
```

DebuggerDemo opens a window, assigns values to two variables, adds the values of the variables together, and writes the resulting total to the window. The NumToString() routine is a Toolbox function that accepts a long variable and a Str255 variable as parameters. The function converts the number to a string and places it in the Str255 variable. Since the sum of 3.2 and 6.8 is 10, I was assuming that NumToString() would convert the number 10 to the string 10. Then I'd be all set to write the string to the window using DrawString(). You can see in Figure 1–32 that instead of 10 being written to the window, 9 was. The program contains a very fundamental error, which

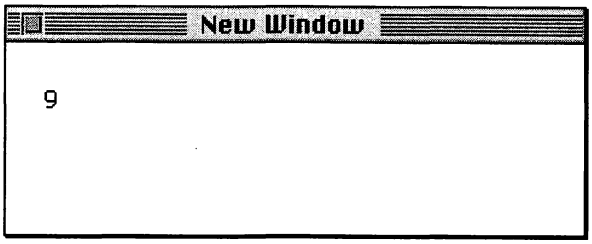


FIGURE 1–32 The results of running the DebuggerDemo program

you may have caught just by looking at the source code. The purpose of this example isn't to stump you, though; it's to allow you to see the debugger in action. So let's assume that neither you nor I know just what went wrong. That means it's time to run the DebuggerDemo program with the Symantec debugger turned on.

Debugging a Program

To run the debugger, make sure that Use Debugger is checked in the Project menu. Then select Run from that same menu. I've done that with DebuggerDemo. You saw how the Source window and the Data window look for DebuggerDemo back in Figure 1–31.

With the debugger running, the first thing you'll do is type in the name of a variable whose value you want to monitor, or follow, as the program runs. Type the variable name in the box at the top of the Data window; then press Enter. Do this for each of the variables you want to watch. In Figure 1–33, I've entered the `num_1` and `num_2` variable names and am about to do the same for the `total` variable.

You can see that in Figure 1–33 the variables `num_1` and `num_2` have values—before they have even been used in assignment statements. At the start of a program a variable contains a "garbage" value, which turns out to be whatever value was last left at the memory location that the variable occupies. This value is sometimes 0, but it can be any number at all.

After entering variable names in the Data window, you'll want to set a *breakpoint* in the Source window. A breakpoint specifies the line of code at which you want the debugger to pause. When you click on the Go button in the debugger Source window, the program will start executing. If you don't have at least one breakpoint set, the program will run from start to finish without pausing. You'll never get a chance to look at the values of the

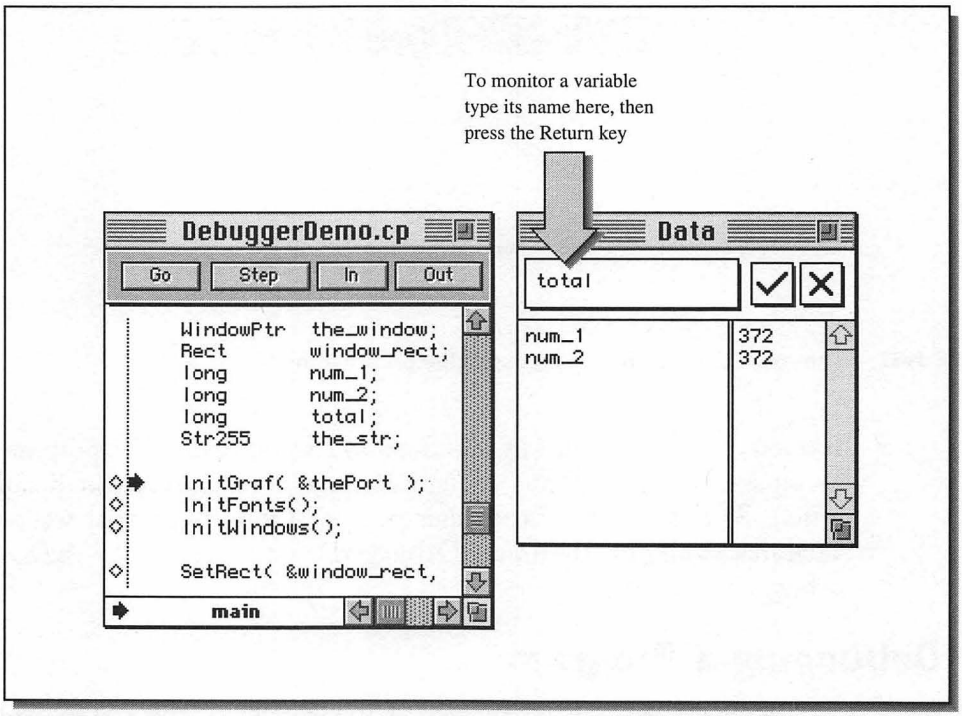


FIGURE 1–33 Entering the names of variables whose values are to be monitored as the program runs

variables—and that's the whole point of debugging. You set a breakpoint by clicking on the small diamond that appears to the left of the line of code at which you want the program to halt. Figure 1–34 shows a breakpoint set in the DebuggerDemo program.

With a breakpoint set, it's time to start your program running by clicking on the Go button in the Source Window. The program will execute up to the breakpoint. In Figure 1–34, you can see a black arrow just to the right of the breakpoint diamond. The arrow indicates where the program has stopped. It's important to note that the line to which the arrow points hasn't yet executed—it is the next line of code that will execute. You can verify this by looking at the Data window in Figure 1–34. The value of `num_1` is still 372; it hasn't yet taken on the value it will be given in the assignment statement.

To execute the single line of code that the black arrow points to, click on the Step button in the Source window. You'll see the arrow move down a line to indicate that the line at which the program was halted has now exe-

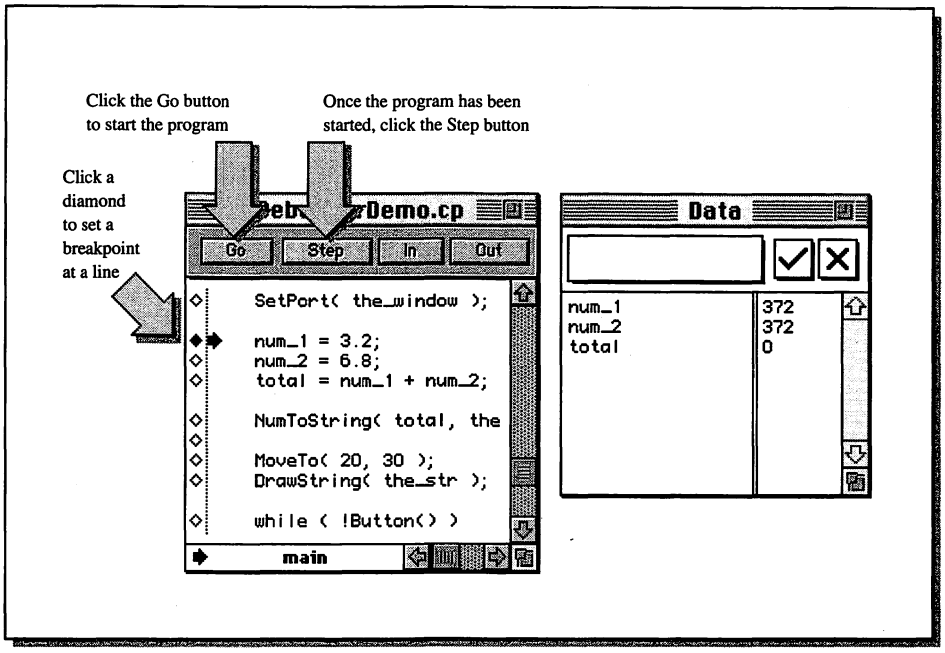


FIGURE 1–34 Setting a breakpoint and using the Go and Step buttons in the debugger

cuted. You can click on the Step button again and again to single-step through as much of the program as you want. In Figure 1–35, you can see that the Step button was clicked once and that the arrow now points to the line following the `num_1` assignment. The Data window shows you that the value of `num_1` has changed from 372 to 3.

The Data window in Figure 1–35 shows the values of the variables after the following line has executed:

```
num_1 = 3.2;
```

Note that the value for `num_1` in the Data window is 3—not 3.2. The variables `num_1`, `num_2`, and `total` were all declared to be of type long. Recall from C that the long is a form of integer. Integers don't hold the decimal, or fractional, portion of a number. Therein lies the reason for `total` having a value of 9 rather than 10. After the program runs, `num_1` has a value of 3 and `num_2` has a value of 6.

You might have been able to detect the bug in the DebuggerDemo program without the use of the Symantec debugger, but now that you're familiar with its use, you'll be able to solve much trickier programming problems.

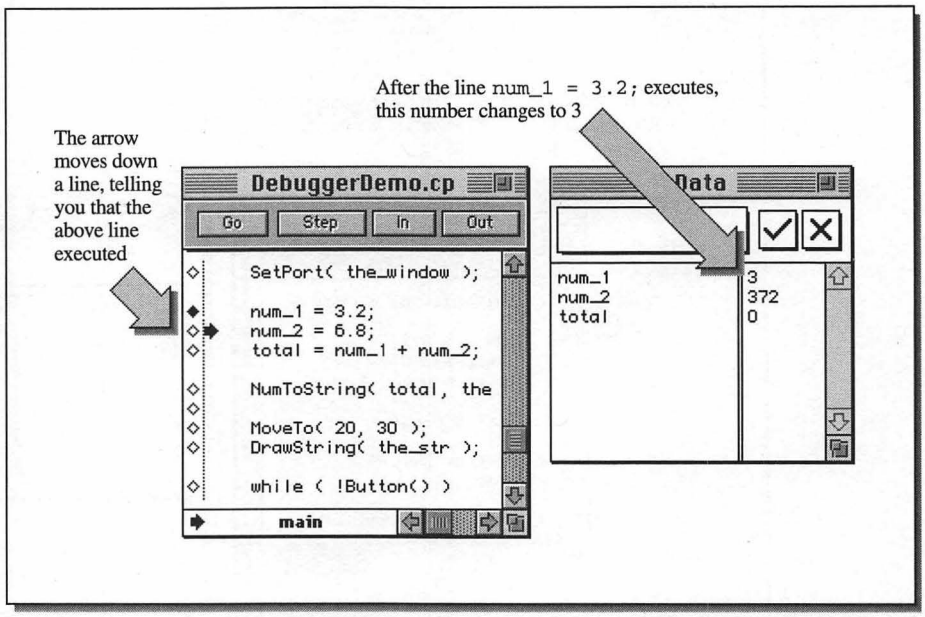


FIGURE 1–35 After the Step button is clicked, the black arrow moves down a line and the values in the Data window are updated.

Turning Code into an Application

The THINK Project Manager makes it easy to turn any project into a stand-alone Macintosh application. If you haven't selected Run from the Project menu to test your project, do that now. When you're satisfied that the code works properly, choose Build Application from the Project menu, as is being done in Figure 1–36.

Before building the application, the THINK Project Manager lets you enter the name you want to give the program. In Figure 1–37, the program is being given the name HelloWorld.

Click on the Save button, and, after just a second or two, you'll have a new Mac program. To verify this, go to the desktop and open the HelloWorld f folder that's in your Development folder. There, along with the project file and the source code file, will be the icon for your new application, as shown in Figure 1–38.

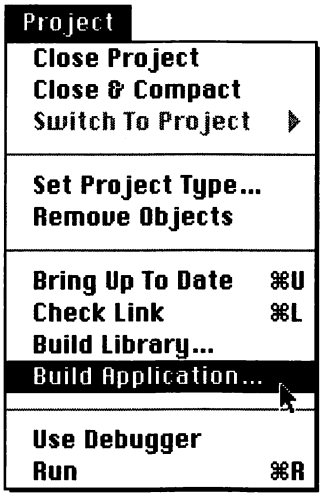


FIGURE 1–36 Turning the code into a stand-alone application

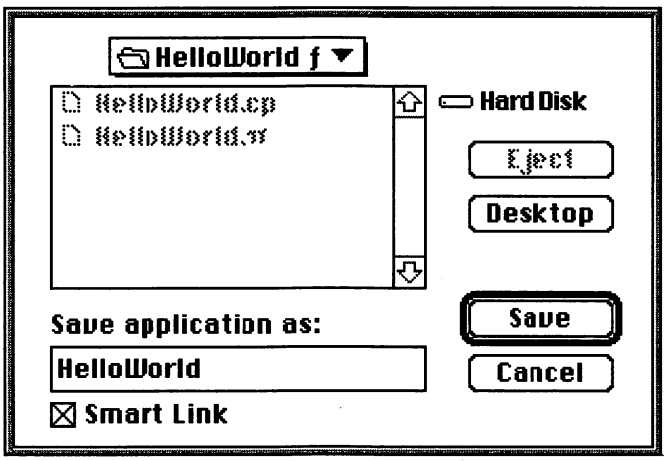


FIGURE 1–37 Giving the application a name

Using the Included Projects

The Symantec C++ project file and source code for each example program in this book can be found on the included disk. If you don't have the THINK Project Manager running, you can open any existing project by simply double-clicking on its icon. If the THINK Project Manager is already running, you must first close the current project. Select Close Project from the Project menu. A dialog box will open. Use the pop-up menu in it to move into the Symantec C++ Examples folder that's located in the Simulator C++ folder. Once there, you can double-click on any folder in the list.

If you have the Symantec C++ compiler, try running the HelloWorld.π project in the (C01)HelloWorld ƒ folder. All the examples in this book work for both version 6.0 and version 7.0 of Symantec C++. If you're using Symantec C++ 6.0 and you double-click on one of the included projects, the project will open. If you're using Symantec C++ 7.0, you'll first see a dialog box like the one shown in Figure 1–39. Click on the Convert button. The dialog box will be dismissed, and the project will open.

About That Code...

If you've written a Macintosh program using the C language, the HelloWorld source code should look very familiar to you. If it does, you should congratulate yourself for remembering C, because that's exactly what the

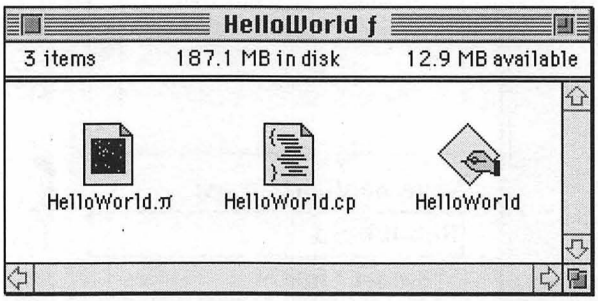


FIGURE 1–38 The icon for the new application will be in the HelloWorld ƒ folder

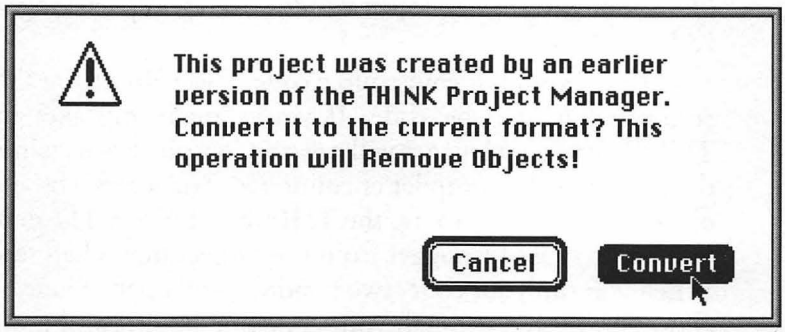


FIGURE 1–39 Opening one of the included example projects using Symantec C++ 7.0

source code is! Remember the definition of the C++ language: it's the C language, plus. In the HelloWorld program, I intentionally chose not to include any of the "plus" features. The purpose of this chapter was to familiarize you with the environment of the Symantec compiler, not to have you master a new language. You have the remainder of the book for that—starting with the very next chapter....

Chapter Summary

Any program that was created using the Symantec C++ compiler started out as a *project file.* A project file holds the names of the source code files and library files needed to compile a single program. When code is compiled, the resulting object code is also stored in the project file. The THINK Project Manager—the combined programming environment that consists of a text editor, compiler, linker, and interface for these components—keeps track of the contents of a project file.

Almost all THINK C++ projects need three libraries in order to successfully compile. You'll use the Add Files menu item from the Source menu to add the MacTraps, ANSI++, and CPlusLib libraries to each of your projects. You'll use this same menu item to add your source code file to the project—after creating the new file by selecting New and Save As from the File menu.

Projects consist of segments—sections of source code that must not exceed 32K. If a single segment goes beyond this size limit, you can move

files about within the project window to regroup existing segments or create additional ones.

After typing in your source code, select Run from the Project menu to compile and test the code. If you've made mistakes in your code, the THINK Project Manager will open a Compile Errors window that describes the problems the compiler encountered. For errors whose source is less than obvious, you'll want to use the THINK debugger. The debugger is activated by selecting Use Debugger from the Project menu before you run the code. When you run your code, two windows will open. These debugger windows allow you to step through your code line-by-line and monitor the values of the variables as the code executes.

When you're satisfied that your code is functional, you'll turn it into a stand-alone application by selecting Build Application from the Project menu. The result will appear on your Macintosh desktop in the form of an executable program with its own icon.



Chapter 2

Introduction to C++ and OOP

While often thought of as one and the same, C++ and object-oriented programming are not the same at all. In this chapter, you'll see the differences and similarities between C and C++, and you'll learn how OOP—object-oriented programming—is used to create programs in the C++ language.

Object-oriented programming uses objects to represent the data and the actions, or operations, that are performed on that data. This chapter will take a long, hard look at objects and the data structure that defines an object—the *class*.

C++ and Object-Oriented Programming

A program written in the C++ language is an object-oriented program, right? Not necessarily. While a C++ program usually does, in fact, work with objects, it doesn't have to.

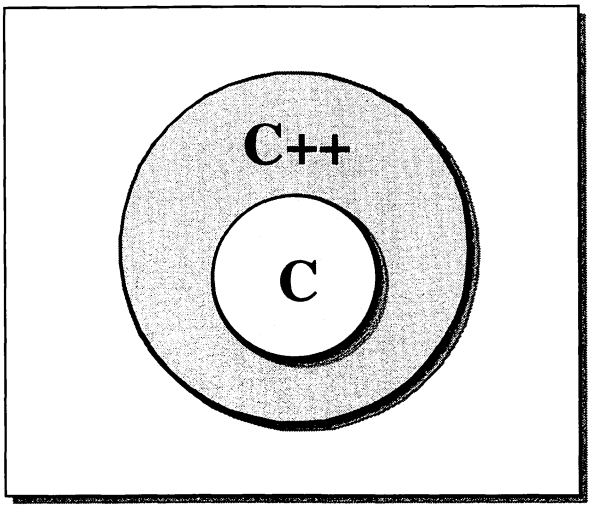


FIGURE 2–1 The C++ language includes all of the C language—and more.

C, C++, and OOP

If you've programmed using C, your skills will not be lost as you make the transition to C++. That's because the C++ language is a *superset* of the C language, which means C++ includes all of the C language and more. Figure 2–1 gives emphasis to this point.

Some of the programming features, or *elements,* that are found in C++ but not in C are trivial, while others are much more important. The features found in C++ but not in C that are considered the most crucial are those that make *objects* possible. Although C++ programs are usually object-oriented, they don't have to be. Consider this very simple C++ program:

```
void  main( void )
{
   int  i;                  /* declare an integer */
   int  j = 0;              // declare and initialize an
                            // integer

   for (i=0; i < 10; i++)   // loop ten times
      j += 5;               // increment j by 5 at each
                            // pass
}
```

In the C language, comments begin with /* and end with */. This method of writing a comment also applies to C++. C++, however, also lets you identify a single-line comment by using a double slash. The above C++ program contains both types of comments.

While the above program is very trivial—and in itself useless—it does serve to demonstrate one point. A C++ program does not *have* to be object-oriented. This example is nothing more than a trivial C program with a few C++ style comments. There's nothing object-like about it.

OOP is not a programming language. It is a programming *methodology*—a way of organizing or structuring a program. You may have noticed programmers using C++ and OOP almost interchangeably. That's because while programs written in C++ don't have to be object-oriented, they almost always are. The elements that were added to the C language to make C++ are, for the most part, elements that allow programs to become object-oriented. If a C++ program doesn't take advantage of those elements, there is little difference between it and a C program.

From C to C++ to OOP

Since C++ is built on C, and OOP is a way of working with the elements of the C++ language, it would make sense to present the material in this book in an order that follows this progression. And so, of course, it does. Here's a summary of the next three chapters.

Chapter 3 is a summary of C. It is not intended to be a complete reference to C. Instead, it highlights the basic elements of the C language—those parts of C that are also basic to C++ and used regularly in C++ programs.

Chapter 4 covers many of the C++ additions to the C language but not the C++ elements that allow object programming. You've already seen one such element—the use of the double slash to denote a comment.

Chapter 5 covers more of the programming elements that are found in C++ but not in C. In particular, this chapter discusses the features of C++ that enable you to write object-oriented programs.

Chapters 3, 4, and 5 present plenty of source code examples to illustrate the programming discussions. This chapter doesn't. Instead, the remainder of this chapter consists of an overview of the ways in which an object-oriented program differs from a procedural one and a general look at the programming elements of C++ that make objects possible.

Procedural and Object Programming

Object-oriented programming isn't new; it has been around for many years. But it is new to most people. In the past, most programs were traditionally and historically written using a *procedural* language, classically speaking.

Procedural Programming

In a procedural, or *structured,* programming language such as C or Pascal, variables are defined to hold data. Routines are then written to operate, or act, on that data. In procedural programming, there is no correlation between data and the routines that can act on that data—until a routine is called. Only when data is passed to a routine—in the form of *parameters*—does it become associated with that routine. Figure 2–2 shows this.

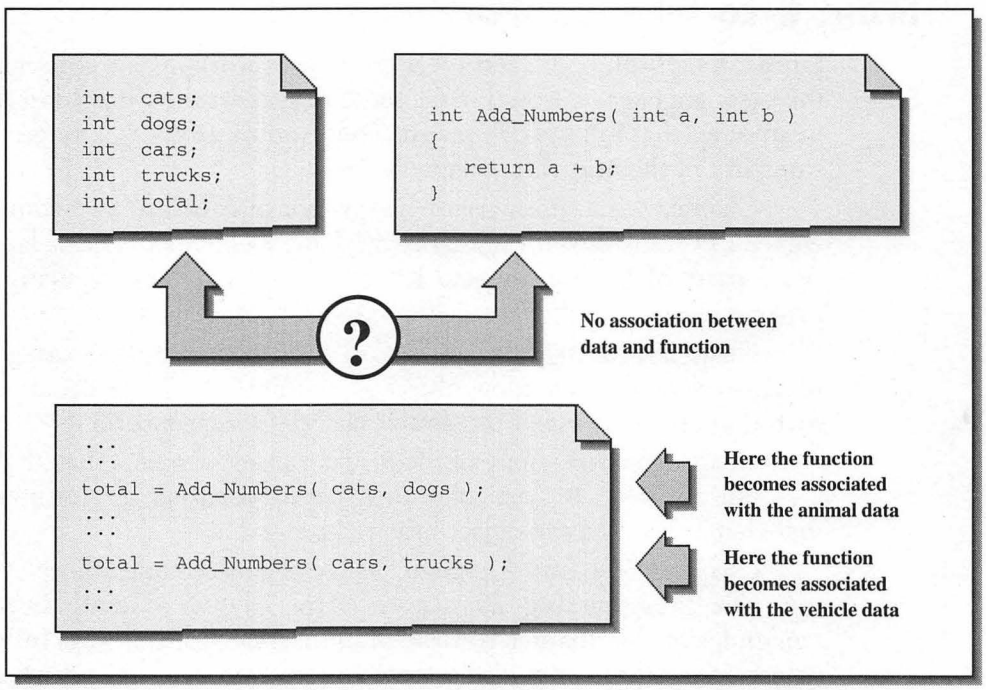


FIGURE 2–2 In a procedural language, data and functions are not clearly related.

NOTE

Routine is a general term for a procedure or function. The Pascal language calls a routine that returns a value a *function* and a routine that doesn't return a value a *procedure*. In C, all routines are called *functions*

A program written in a procedural language almost always relies on at least some of the program's data being stored in global variables. Program data that is kept as a global variable can be accessed by any and all functions that appear in that program. Thus, data that will be used by multiple routines—such as data that is read, manipulated, stored, and printed—is usually a candidate for a global variable. While global data makes a programmer's life easier, it also allows for corruption of data. Because all functions have access to global data, a programmer may inadvertently write a routine that alters the value of a global variable.

Despite its drawbacks, including a reliance on global data and an unnatural division of data and the operations that work with data, procedural languages have been used in the vast majority of the programs written in the last couple of decades. Only recently have programmers begun to make the change from procedural programming to object-oriented programming.

Object-Oriented Programming

You've just seen that procedural programming data and the actions that work on data are separate and seemingly unrelated. The removal of this artificial division between data and functions is the primary difference between procedural and object-oriented programming. In procedural programming, the necessary data structures are first created. Then, routines that operate on the data are defined. Each routine may operate on some, much, or all of the data. In object-oriented programming, data and actions are kept together. As data is defined, so are the actions, or functions, that work with that data. Not only are data and functions defined at the same time, they are also tied together in the form of *objects*. Figure 2–3 shows this difference.

Even though the two programming methodologies shown in Figure 2–3 define the same data and the same functions, there is a big difference between them. In the procedural example, it is not clear which of the two functions works with which of the three pieces of data. In the object-oriented example in that same figure, it is clear that both functions are capable of working with all three data elements. By packaging data and functions together, the need for global data diminishes or disappears altogether in object-oriented programming.

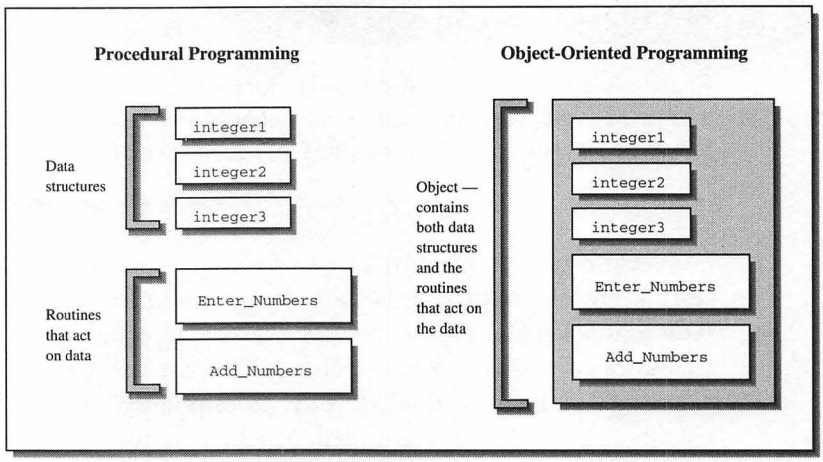


FIGURE 2–3 A procedural language doesn't clearly relate data to functions; an object-oriented language does.

As you have surely surmised, objects are the key element in object-oriented programming. Because a shape is naturally thought of as an object, let's explore objects in greater depth by looking at an example that represents a rectangle as an object.

On the Macintosh, a rectangle is defined by the pixel coordinates of its four corners. So these are the four data elements my rectangle object will need. In C++, an object's data—the variables the object works with—are called *data members.* The names of the functions that act on the object's data members are referred to as the object's *member functions.* Like the number of data members an object has, the number of member functions it has depends on what the object is representing. While I could think of dozens of different things to do with a rectangle—rotate it, shrink it, color it—for the sake of simplicity I'll define just three operations. Drawing and erasing the rectangle seem like necessities, and for variety I'll also allow the rectangle to grow. Figure 2–4 shows the data members and member functions I would define for my rectangle object.

NOTE

Data members are sometimes called *instance variables.* Member functions can also be referred to as *methods.* This book will use the terms *data members* and *member functions.*

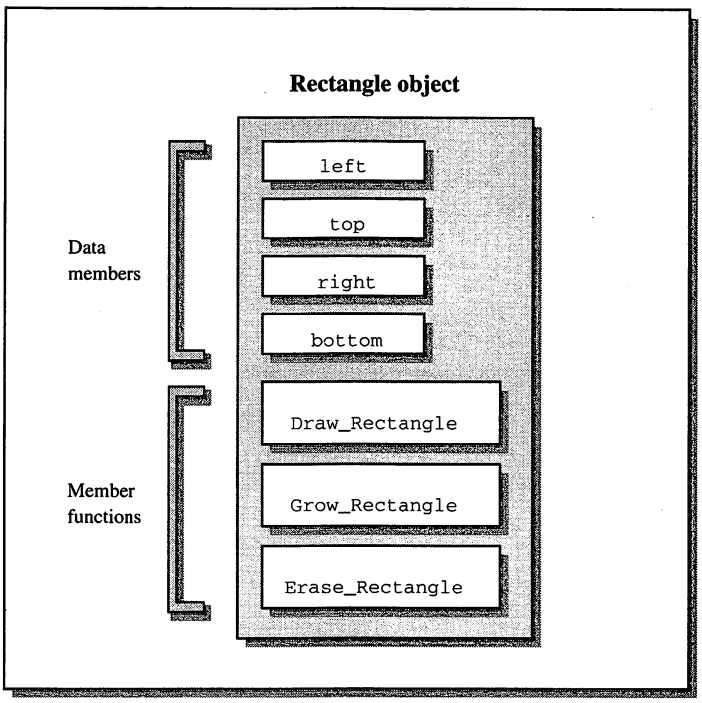


FIGURE 2–4 One way of implementing an object that represents a rectangle

What variables will be needed by the three functions that act on the rectangle? Only the four data members that are a part of the rectangle object. And how many different operations can be performed on a rectangle? Just the three that are named by the object's member functions.

In a procedural program, an action is carried out by calling a routine—a function or procedure. In a program that works with objects, the program tells an object to carry out an action by sending it a *message*. The message tells the object which of its built-in member functions should be executed. In Figure 2–5, a program sends a rectangle object a draw message so that the object executes its Draw_Rectangle member function. In Figure 2–6, the same program sends the same object a grow message so that the object executes its Grow_Rectangle member function.

Figures 2–5 and 2–6 highlight a key difference between procedural and object programming. In procedural programming, you call a routine. In object programming, you send a message to an object. The message tells the object which of its routines to use.

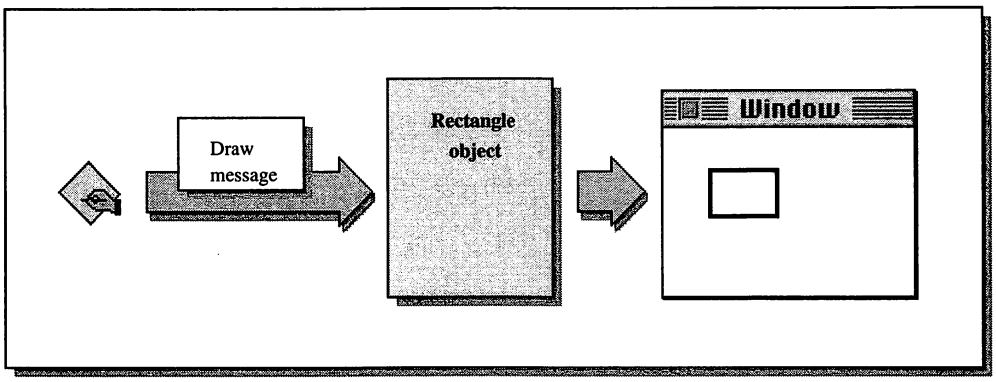


FIGURE 2–5 A program sends a rectangle object a draw message.

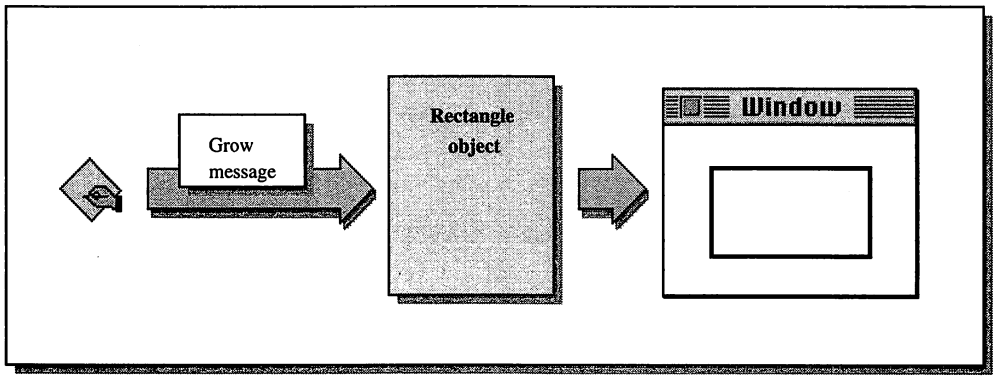


FIGURE 2–6 A program sends a rectangle object a grow message.

The Advantages of Object-Oriented Programming

Keeping together a data structure and the functions that operate on that data structure is called *encapsulation.* Encapsulation is one of the primary advantages of object-oriented programming over procedural programming. Another benefit is ease of code modification. It is easier to make changes in an object-oriented program than it is to make similar changes in a procedural program.

As a case in point, consider a Macintosh programmer who works for the Acme Information Center. This company has been asked to create a program that will offer users recommendations as to where to shop, what doctor to see, and a host of other bits of information. The recommendations will be based on several pieces of information, one of which is the gender of the user. Being well-versed in C, the programmer writes a procedural program consisting of thousands of lines of code, some of which are shown in Figure 2–7.

Since there are only two sexes, the programmer is feeling pretty smug about his selection of data and functions. That is, until his eccentric boss tells him to make the program capable of handling unisex aliens. Now, the programmer must go through the thousands of lines of code, searching for routines like Do_Stuff(). That routine, part of which is shown in Figure 2–7, handles what our unfortunate programmer thought would be the only two gender cases—male and female. Now, he will have to add more global variables—variables that will apply to activities specific to the aliens. And

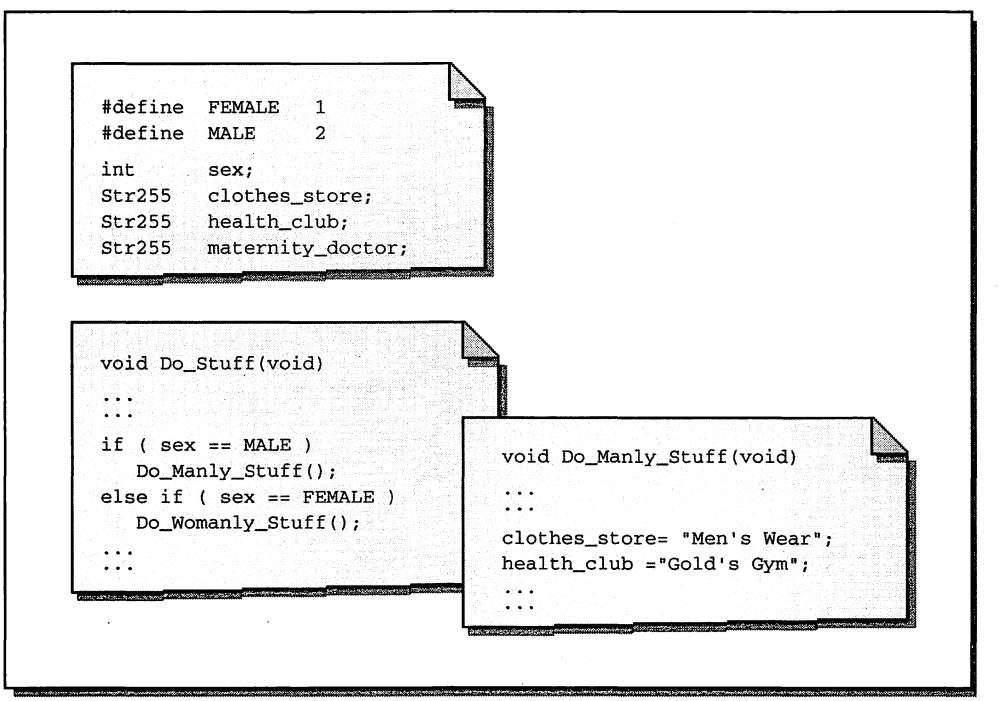


FIGURE 2–7 Some typical procedural language code

speaking of specifics, was he wise to make `maternity_doctor` a global variable, when, before the arrival of the unisex alien, it applied to only one of the two genders?

While aliens don't turn up in typical programs, changes like the ones shown here do. So while not entirely realistic, the alien example serves to illustrate the problems inherent in procedural programming. Data and the functions that operate on the data are not closely bound. The addition of a single new data type—like the alien—may require a great deal of work, because all of the source code must be searched for affected sections.

By binding data and the functions that work on that data, object-oriented programming avoids the pitfalls of procedural programming. If Acme's programmer had used object-oriented programming, his initial programming effort would have looked more like the one shown in Figure 2–8.

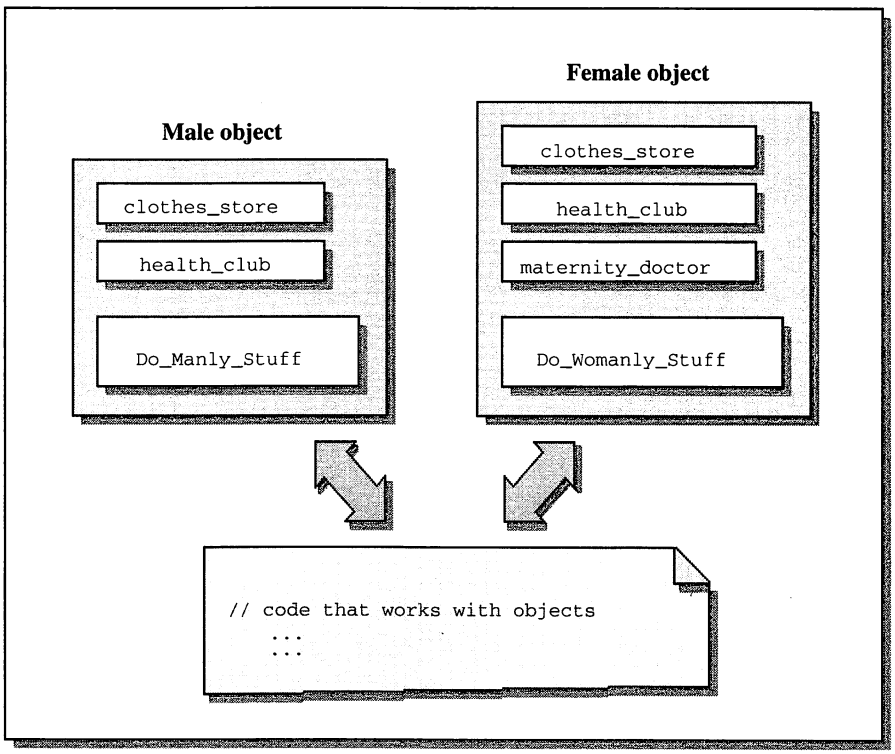


FIGURE 2–8 Using two objects to represent two different things

While I've shown the object version in only a general way, you should still be able to make some observations about it. First, notice that each object contains only the data that it needs. Thus, the `maternity_doctor` data is no longer global. It now appropriately appears in only the female object. Second, notice that the routines that act on each gender are now closely associated with the data that they use.

What about the request from the programmer's boss? What is required to add a new data type—the alien? Figure 2–9 answers the question.

Using object-oriented programming, when faced with a new data type—the alien—the programmer can give thought to the data members and member functions that will be appropriate for this new type. And because the data and functions that will apply to the alien are encapsulated and hidden from the other objects—the male and female objects—the programmer has little concern for how the new code will affect previously written code.

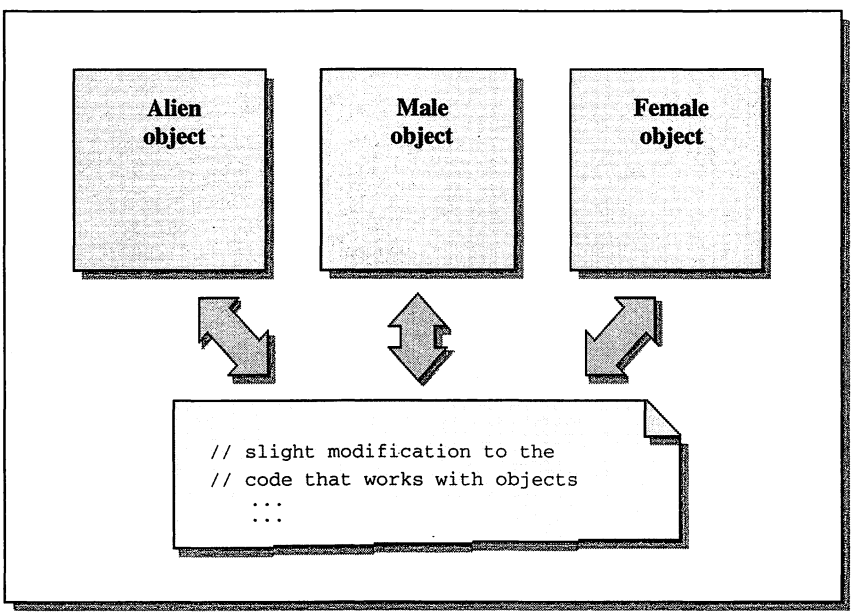


FIGURE 2–9 Little effort is needed to add a new type of object to a program.

Classes

So far I've discussed object-oriented programming in general terms, but you've seen no actual object source code. In this section, I'll delve a little deeper into objects, again using generalities. Don't worry about the specifics of C++ objects, though. There's plenty of time for that—hundreds of pages, in fact.

The Class—the Pattern of Objects

Up to this point, I've discussed an object as if it were defined and then used by the program. That's not entirely accurate. What is really defined is a *class.* A class is a pattern from which objects are created. Figure 2–10 shows that once a class is defined, more than one object can be created from it.

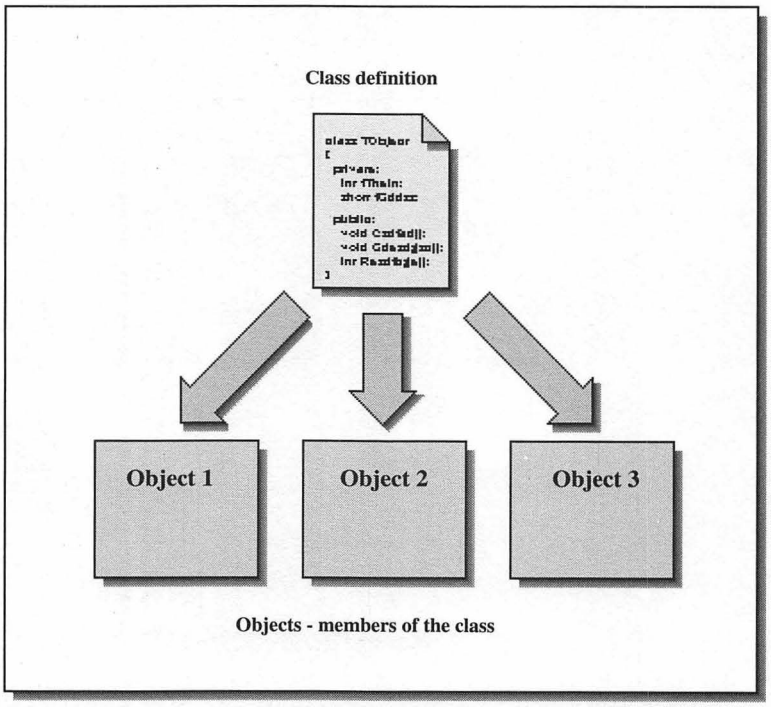


FIGURE 2–10 Many objects can be created from a single class definition.

NOTE

All right, I'll bend a little and throw a bit of code at you. In C++ there is a class keyword. Defining a class in C++ is much like defining a struct in C. In fact, the class is built on the concept of the struct. Chapter 5 will provide all the detail you'll need to write your own classes. For now, look at this example C++ class definition and note its similarities to the C struct data type:

```
class  MyClass
{
   private:
      Str255  name;
      short   age;

   public:
      void  Add_Info( Str255, short );
      void  Write_Info( void );
}
```

Why would you want more than one of the same type of object? Think of a class as just another data type—like the int, short, and float data types in C. You don't use these data types directly in a program. Rather, you declare variables to be of these types. And you quite often declare more than one variable of a given type. You are not limited to declaring a single int variable in a program. Nor are you limited to creating a single object from one class definition.

Creating Multiple Objects

Let's dig a little deeper into the idea of creating multiple objects from one class. Again, I'll use a simple example—though this one will be a little closer to real-world programming.

The Acme Bolt and Nut Company sells a large number of hardware items through various catalogs. The company wants to maintain certain information about each item—information such as the item name, part number, and which catalog the item appears in. To oblige the company, Acme's computer programmer (who is, incidentally, a Macintosh programmer) writes a program that defines a single class. This part information class is shown in Figure 2–11.

In Figure 2–11, you can see that the programmer feels that a class containing three data members and three member functions will suffice to describe and work with any item the company carries. The three data members should be self-explanatory. The three member functions need only a brief explanation.

The New_Part member function allows a user of the program to add information about a new type of item—Acme is forever designing and producing new types of bolts. The Write_Part_Info member function allows the program user to print out information about the part—its name and part number and the volume of the catalog in which it appears. Delete_Part lets the program user remove information about a discontinued part.

The definition of a class doesn't create any objects of that class; it merely defines what an object based on this class will look like. Remember, a class is like a data type. It is a pattern from which 1, 10, or 10,000 objects can be created, as shown in Figure 2–12.

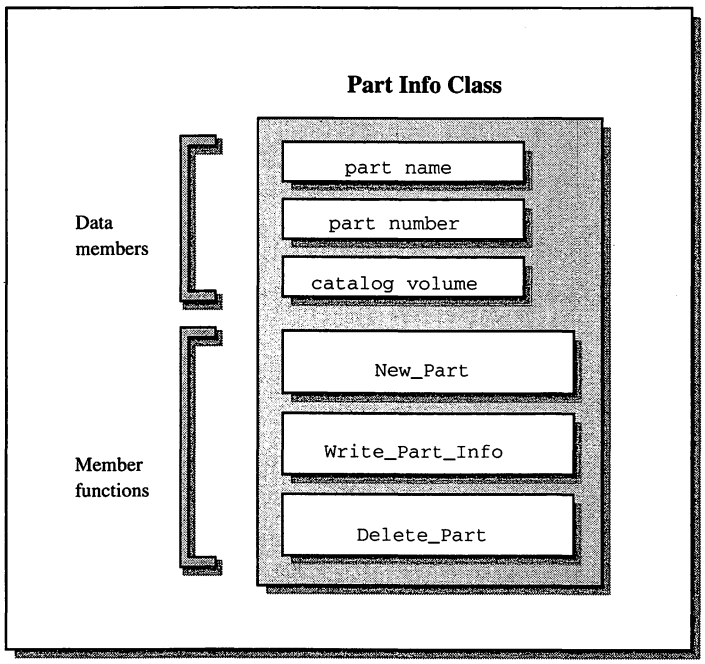


FIGURE 2–11 A class to represent a single part in a catalog

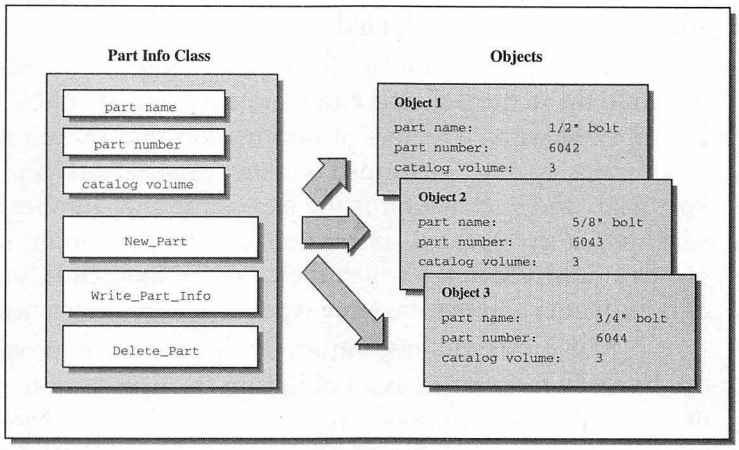


FIGURE 2–12 A class is the pattern from which objects are created.

Objects can be declared in your source code, just as variables based on C data types are declared. They can also be created *dynamically,* or *on the fly.* That is, all the objects a program will use do not have to be declared in your code; only the class or classes must be declared. Objects can be created as the program runs and as the need arises. This is typically the case, as Acme's programmer finds out. The program must be capable of creating and deleting objects at the user's whim. Because the programmer doesn't know in advance how many items will be listed in each catalog, he can't account for every object beforehand.

Object-Oriented Programming and C

In writing a C++ program, not every line of code you write will be object-like; much of it will closely resemble Macintosh C language code you've written in the past. There are two reasons for this, as you're about to see.

Everything Need Not Be an Object

Not all programming concepts lend themselves to object-oriented programming. OOP is most useful when you have a well-defined data object and a number of well-defined actions that will be performed on it. Working

with a shape—like the rectangle discussed earlier—is an example of a situation in which object-oriented methods are practical.

Another good candidate for representation as an object is a record, as shown in the Acme Bolt and Nut Company's use of objects to keep track of its inventory of parts. The bits of information that make up a record and the actions that are to be performed on them can be clearly and concisely defined. For Acme, the information is the part name and number and the catalog in which the item appeared. The actions are the creation, printing, and deletion of a part. Each record, of which there may be thousands, stores the same types of information and has the same types of actions performed on it.

A Macintosh window, intuitively thought of as an object by most users, can be easily represented as an object. In the next section, you'll see in a little more detail how a window is represented in this way. Later in this book, I'll present the source code for a C++ program that uses windows as objects.

So far, I've discussed things that can be turned into objects. What about things that can't, or shouldn't, be turned into objects? There are many instances in which there is an action to perform but nothing clearly defined on which to act. For example, I may want to evoke a sound from the Mac's built-in speaker. That's an action, but there is no object to perform the action upon. I don't need to try to represent the Macintosh speaker as an object. I simply call the Toolbox routine SysBeep(). Attempting to turn a simple action into object-based code undermines the purpose of object-oriented programming. Don't add unnecessary complexity to your programs.

C++ on the Macintosh, like C on the Mac, makes extensive use of Toolbox calls. It also uses other common C code, such as for and while loops and if and switch branches—especially in nonobject portions of your C++ code. An example is an event loop. The event loop of a Mac C++ program looks very much like an event loop in a Macintosh C program.

C++ Uses C

Objects have member functions that carry out actions on their data members. In the classes you've seen so far, the member functions have been shown as just function names. That is, in fact, how member functions are listed in a class definition. The actual body of code that makes up a member function appears elsewhere in your source code. Because I just mentioned that a Macintosh window is a good candidate for representation as an object, I'll use a window as an example. Figure 2–13 shows the class definition for a window. I've deliberately left much of the class obscure so that we can focus on just a couple of the member functions.

My window class definition has at least two member functions. I've shown the Drag_Wind and Grow_Wind member functions in Figure 2–13.

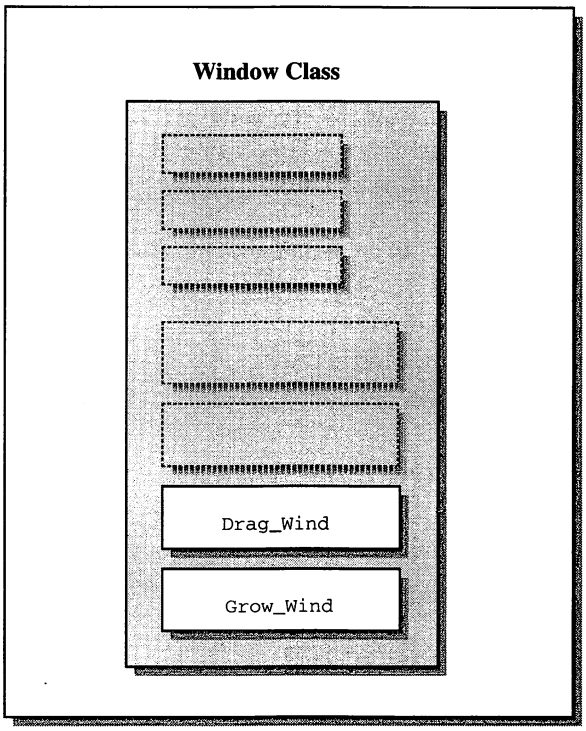


FIGURE 2–13 Part of a class that defines a window

You now know that the code that makes up these functions appears outside of the class. Here's a look at how I implemented the Drag_Wind member function:

```
void  Window :: Drag_Wind( Point where )
{
   DragWindow( the_window, where, &drag_rect );
}
```

While the double colons in the first line may seem a little cryptic, the body of the function should be very familiar. It's simply a call to the Toolbox function DragWindow(). This line of code appears just as it would in a Macintosh program written in C. Recall that C++ is a superset of C. Most valid C code is also valid C++ code. That's why much of your C++ source code will look like C code.

As further proof that C++ relies heavily on C, take a look at the code I wrote for the Grow_Wind member function. You may not understand the purpose of each line, but that's not important. What is important is that you notice that the Grow_Wind member function is made up of local variables—all of which are declared to be of data types found in Macintosh C—and quite a few Toolbox calls.

```
long  Window :: Grow_Wind( Point where )
{
   long     wind_size;
   Rect     min_wind_size;
   GrafPtr  save_port;

   min_wind_size = this->min_rect;

   wind_size = GrowWindow( the_window, where,
                           &min_wind_size );

   if ( wind_size != 0 )
   {
      GetPort( &save_port );
      SetPort( the_window );
      EraseRect( &the_window->portRect );
      SizeWindow( the_window, LoWord(wind_size),
                  HiWord(wind_size), true );
      InvalRect( &the_window->portRect );
      SetPort( save_port );
   }

   return ( wind_size );
}
```

NOTE

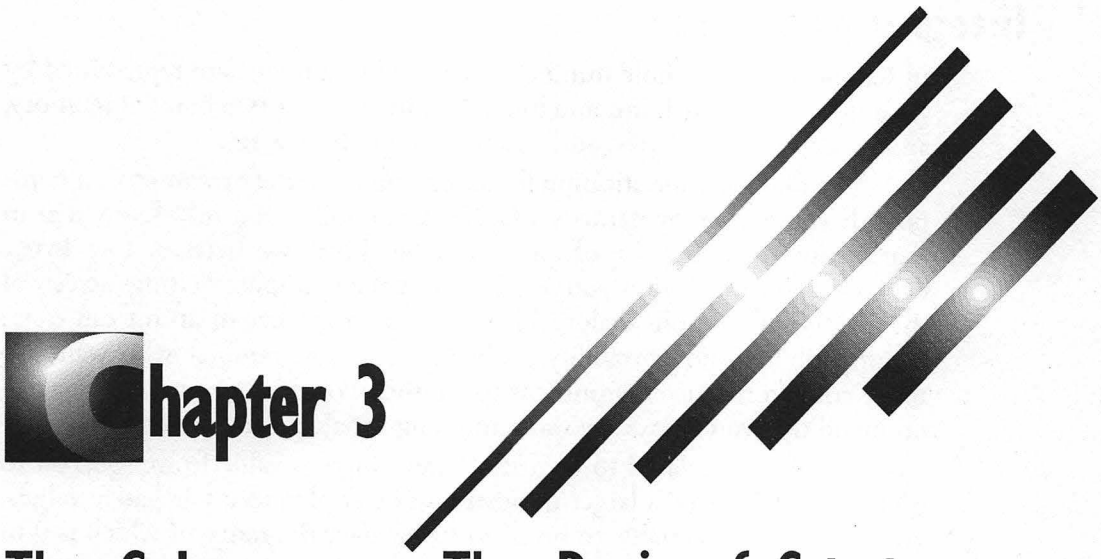
It should be encouraging to you to know that none of the effort you have put into learning C and Macintosh programming has been wasted. You'll use all of your knowledge of C and the Macintosh Toolbox in your C++ programs. Don't think of learning C++ as learning an entirely new language; think of it as adding to your existing knowledge base.

Chapter Summary

The C++ language is a superset of the C language. That means that C++ includes all of the elements of C plus some additional ones. Any time and effort you've spent learning C will not be wasted on your journey to learning C++.

Programs written in C++ usually work with objects, but they don't have to. OOP, or object-oriented programming, is not in itself a programming language. Rather, it is a way of organizing a program. The additions to the C language that spawned the C++ language were primarily those that allow programs to become object-oriented. A C++ program doesn't have to take advantage of these elements, though. If it doesn't, there is little difference between it and a C program, and it isn't considered an object-oriented program.

A procedural language, like Pascal or C, has variables that hold data and routines that act on those variables. There is, however, no direct connection between a given routine and a given piece of data. In object-oriented programming there is. Data and the functions that act on that data are grouped to form *classes*. From these class definitions come *objects*. An object is a variable that is of a certain class type. Object-oriented programming allows the programmer to create as many objects from one class definition as needed, just as a programmer can declare several variables of a standard data type such as a short or a float.



Chapter 3

The C Language: The Basis of C++

The C++ language is built on the data types and keywords that make up the C language, so you'll want to have a good understanding of C before you tackle C++. Are you well-versed in C? Are you familiar with all of the basic data types, branching statements, and looping statements? Do you recall how to declare and use a struct—the C data type upon which the C++ class data type is built? If you do, then feel free to skip this chapter; it's a review of C. You won't find anything that's strictly C++ here; that begins in Chapter 4. If you know C but feel a little rusty, skim this chapter for a refresher.

Basic Data Types

Whole numbers, floating-point numbers, and characters and strings—those are the three basic types of data found in both C and C++. This section provides a quick review of the data types that fall within those general data categories.

Integral Numbers

In C, and in C++, whole numbers, or *integral* numbers, are represented by variables of type short, int, and long. A short occupies two bytes of memory, an int occupies four bytes, and a long occupies four bytes.

The four-byte specification for an int applies to the Symantec C++ compiler. If you've used Symantec's THINK C compiler, you may know that in that environment the size of an int can be either two bytes or four bytes, depending on the option you've selected in the Compiler Settings screen of the THINK C options dialog. This variance in the size of an int can cause incompatibility and portability problems for a programmer who wants to move code from one environment to another. For this reason, I *strongly* recommend that you always use short and long variables rather than ints.

A variable declared to be a short can range in value from –32,768 to +32,767. If you need a larger number and don't plan to use negative values, you can declare a variable to be an *unsigned short*,the range of which is 0 to +65,535.

For a long variable, the range is –2,147,483,648 to +2,147,483,647. As with a short, you can declare a long variable to be unsigned, in which case the range is from 0 to +4,294,967,295. Here are a few example declarations:

```
short  dogs;                    // max. value is 32,767
unsigned short more_dogs;       // max. value is 65,535
long  usa_population;           // max. value is 2,147,483,647
unsigned long asia_population   // max. value is 4,294,967,295
```

Some Toolbox routines insist on a parameter of type long. Since a long occupies four bytes of memory, the parameter you pass must also occupy four bytes. If you see example code that has 0L as one of the parameters passed to a Toolbox routine, rather than just a zero, it's because of this requirement. An integral constant, such as zero, doesn't necessarily occupy four bytes. In C and C++, appending the letter L to a number forces it to occupy four bytes and tells the compiler that you want the integer to be treated as a long. Here's how 0 would be forced to occupy the space of a long:

```
0L
```

While a routine may accept an integral constant that hasn't been forced to the size of a long, there's no guarantee that it will.

Floating-Point Numbers

Integral numbers are easy to work with and useful data types, but they cannot be used for numbers that contain decimal points. C and C++ use variables of the data types float, double, and long double to hold floating-point numbers—numbers that have decimal points. A float occupies four bytes of memory. Doubles and long doubles occupy either eight or ten bytes, depending on settings in the Compiler Settings screen of the Symantec C++ options dialog box.

The size of floating-point numbers will seldom be an issue for you. In Macintosh programming, double is the preferred floating-point data type. Whether the compiler is set to store a double in eight or ten bytes of memory, the range of values a variable of type double holds should be plenty large enough for your needs.

The following are examples of floating-point declarations and initializations:

```
float  cost = 19.95;
double  distance = 6.5e2;
long double great_distance = 7.25e6;
```

Characters and Strings

In C and C++ you use the char data type to hold a single character. Enclose the character in single quotes, like this:

```
char  first_initial = 'D';
```

Multiple characters, or *strings,* are held in variables of type Str255. Precede the characters that make up the string with a backslash and the letter p; then enclose the whole works in double quotes, like this:

```
"\pThis is a string"
```

A string variable can be given a value at initialization, as in:

```
Str255  the_str = "\pSymantec C++";
```

Once declared, though, a string cannot be given a value in an assignment statement. While the following looks as if it should work, it doesn't.

```
Str255  the_str;

the_str = "\pSymantec C++";  // This WON'T work
```

To give a string variable a value *after* it has been declared, use a function that moves a string constant into the string variable, byte-by-byte. The following is a short, simple routine that does just that. You might want to copy it into your source code:

```
void Fill_Str255( Str255 the_str, Str255 fill_with_str )
{
   short  str_length;   // length in characters of string
   short  i;            // loop counter

   str_length = *fill_with_str;

   for ( i = str_length; i >= 0; i-- )
      the_str[i]
}
```

Call the Fill_Str255() routine whenever you want to give a string variable a value. Pass to Fill_Str255() the variable to fill along with a string constant. Here's a code snippet that does that and then writes the string to the active window or dialog box:

```
Str255  the_str;
Fill_Str255( the_str, "\pTesting 123");
MoveTo( 20, 30 );
DrawString( the_str );
```

Preprocessor Directives

The Symantec C and C++ compilers contain a *preprocessor* that makes a pass through your source code before compiling. The preprocessor can substitute numbers or characters for symbols and replace a single line with the contents of an entire source file.

The #define Directive

Data that is preset at the start of a program and doesn't change value is said to be *constant.* In C and C++, constants can be established through the use of the #define preprocessor directive. The following sets the name, or symbol, CENTS_PER_DOLLAR to the value 100:

```
#define  CENTS_PER_DOLLAR  100
```

Now, wherever the symbolic name CENTS_PER_DOLLAR appears, the compiler will replace it with the number 100 during the preprocessor's pass through the code.

The liberal use of #define directives has the positive effect of preventing numbers from being scattered about your source code. Instead, all numbers appear at the start of the file or in their own separate source code file. Here are some examples of #define directives and the code that uses them:

```
#define  INTEREST_RATE    0.06
#define  RATE_STR    "\pThe interest rate is currently 6%"

double  base_amount = 1000;
double  interest;

DrawString( RATE_STR );
interest = (base_amount * INTEREST_RATE );
```

The #include Directive

The second most common preprocessor directive is the #include. An #include, followed by a filename, tells the compiler to substitute the entire contents of the named file for the #include line. The following are two examples:

```
#include  "Initialize.h"
#include  <GestaltEqu.h>
```

Most #include files are header files—files containing #defines and function prototypes rather than executable code. When including a header file of your own, enclose the name in quotes. That tells the compiler to begin its search for the file in the directory that contains the original source code file. When including Apple-supplied header files, surround the

Table 3–1 Some of the Commonly Used C Operators

Operator	Description	Operator	Description
+	Addition	<	Less than
-	Subtraction	<=	Less than or equal
*	Multiplication	>	Greater than
/	Division	>=	Greater than or equal
++	Increment	==	Logical equals
--	Decrement	!=	Logical NOT equals
&	Address of	&&	Logical AND
.	Direct selection	||	Logical OR
->	Indirect selection	=	Assignment

filename with the < and > symbols. That tells the compiler to begin its search in the folders of #include files in the Symantec C++ folder.

Operators

The C language provides many ways to process the data you have stored in different types of variables. *Operators* are the symbols that do the processing, and C has a rich set of them. The C operators allow you perform math operations, make comparisons, alter variable values, and more. If you've mastered C operators, your skills won't be wasted—C++ uses them all. Table 3–1 lists the most commonly used C operators.

Looping Statements

Controlling the flow of a program is an important programming concept—in C and every other language. Looping statements provide a program with the means of easily repeating blocks of code. The C language has three kinds

of loops—*while*, *do-while*, and *for*. All three types can be used in C++ programs.

The while Loop

The while statement performs a test on an expression to determine if the statement below the expression should execute. The while loop has the general form shown here:

```
while ( expression )
   statement
```

The first line of a while loop is the conditional test. If the expression that lies between the parentheses evaluates to true, then the statement below the expression executes. If the expression evaluates to false, then the statement below is skipped. The statement part of the while loop can be a single statement or a block of code—a compound statement—nested between opening and closing braces. Here's an example of a loop that will execute 10 times (from x = 0 to x = 9):

```
short  x = 0;
short  total = 0;

while ( x < 10 )
{
   total += 5;
   x++;
}
```

Figure 3–1 shows the flow of control for a section of code that contains a while loop. The code that increments the loop counter is found within the loop body.

The do-while Loop

Related to the while loop is a second type of looping statement—the do-while loop. In the do-while, the test condition appears at the end of the loop body:

```
do
   statement
while ( expression );
```

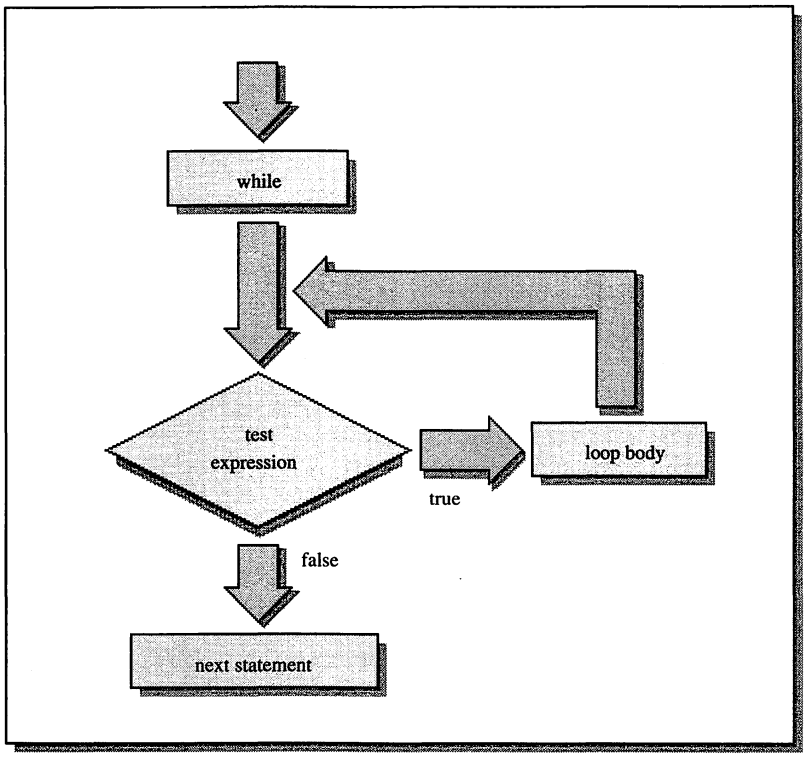


FIGURE 3–1 Program flow of a while loop

Having the test expression appear at the end of the loop means that the loop will always execute at least once. Here's an example of a do-while loop:

```
short  count = 0;
short  total = 1;

do
{
   total += 5;
   ++count;
} while (count < 3);
```

For conciseness, the incrementing of the loop counter can take place within the test expression, as shown here:

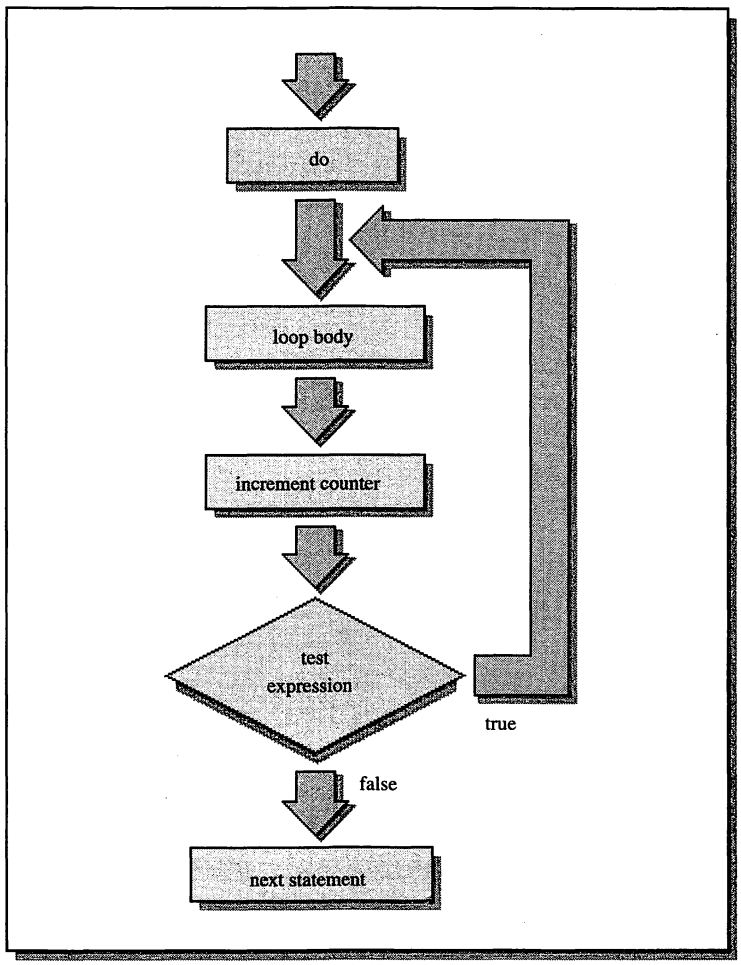


FIGURE 3–2 Program flow of a do-while loop

```
do
{
   total += 5;
} while (count++ < 3);
```

Figure 3–2 shows the program flow of a do-while loop. While the incrementing of the loop counter is shown after the loop body, it could be contained within the body itself.

The for Loop

The third C loop type is the for loop. The for loop performs three actions within the parentheses that follow the for keyword: it initializes a counter to a starting value, compares the counter to an ending value, and increments the counter. The first of these three actions is performed only once. The other two are performed after each pass through the body of the loop. Here's the format of the for loop:

```
for ( initialize counter; test counter; increment counter )
   statement
```

Here's an example of a for loop:

```
short  count;
short  total = 1;

for (count = 0; count < 2; count++)
   total *= 3;
```

Figure 3–3 shows the program flow of a for loop. The test of the expression occurs before each pass through the loop body.

Branching Statements

Looping statements control the flow of a program. Another way to control program flow—and to add complexity and decision-making power—is to use branching statements. For branching, both C and C++ rely on the *if*, *if-else*, *else-if*, and *switch* statements.

The if Branch

The if statement performs a test on an expression. The result of the evaluated expression determines if the statement below the expression is executed once. The if statement has the form shown below.

```
if ( expression )
   statement
```

The first line of the if statement holds the conditional test. If the test between the parentheses evaluates to true, the statement below the

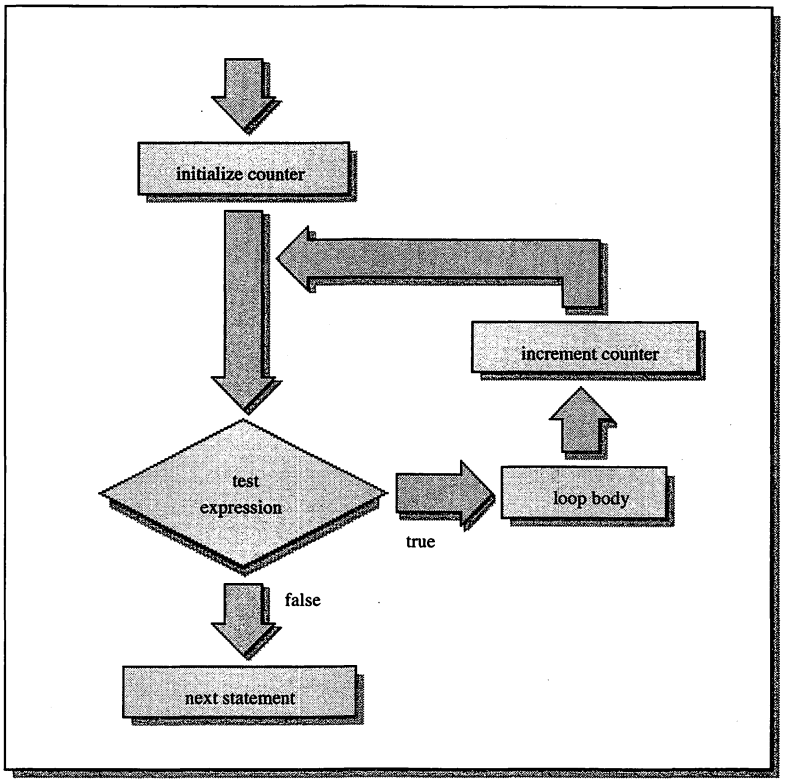


FIGURE 3–3 Program flow of a for loop

expression is executed. If the expression evaluates to false, the statement below the expression is skipped. As in loops, the statement below the if can be either a single statement or a compound statement. Here's an example of an if statement whose test evaluates to true:

```
short  x = 0;
short  total = 0;

if ( x < 10 )
{
   total += 5;
   MoveTo( 20, 30 );
   DrawString( "\pTotal has been increased by 5." );
}
```

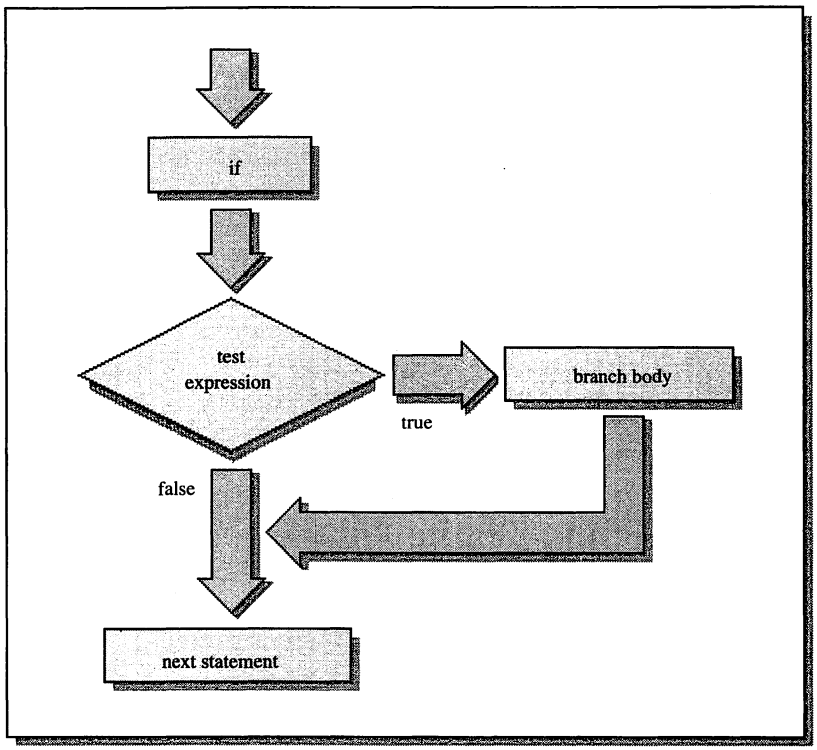


FIGURE 3–4 Program flow of an if branch

Figure 3–4 shows the flow of control for a section of code that contains an if branch.

The if-else Branch

The if statement either executes a statement or it doesn't. On some occasions you will want your program to execute a second statement when the if test fails. For cases such as this, use an expanded form of the if statement—the if-else. Here's the general form of the if-else:

```
if ( expression )
   statement_1
else
   statement_2
```

When you want to have the ability to handle two separate cases, supplement the if statement with an else section. Here's an example:

```
short  x = 15;
short  total = 0;

if ( x < 10 )
{
   total += 5;
   MoveTo( 20, 30 );
   DrawString( "\pTotal has been increased by 5." );
}
else
{
   total = 0;
   MoveTo( 20, 30 );
   DrawString( "\pTotal was too high, it was reset to 0." );
}
```

In the above example, the else section of the if-else will be executed. That's because x starts with a value of 15—a value greater than or equal to 10. When the if test fails, the code below the else is executed. Had x been initialized to a value less than 10 or assigned a value less than 10 at some other point in the code, the code under the if statement would have been executed. Figure 3–5 shows the program flow for a section of code that contains an if-else branch.

The else-if Branch

The if-else handles a situation that can result in either of two outcomes. For a condition that can have more than two outcomes, use the else-if branch. Its general usage is shown here:

```
if ( expression_1 )
   statement_1
else if ( expression_2 )
   statement_2
else
   statement_3
```

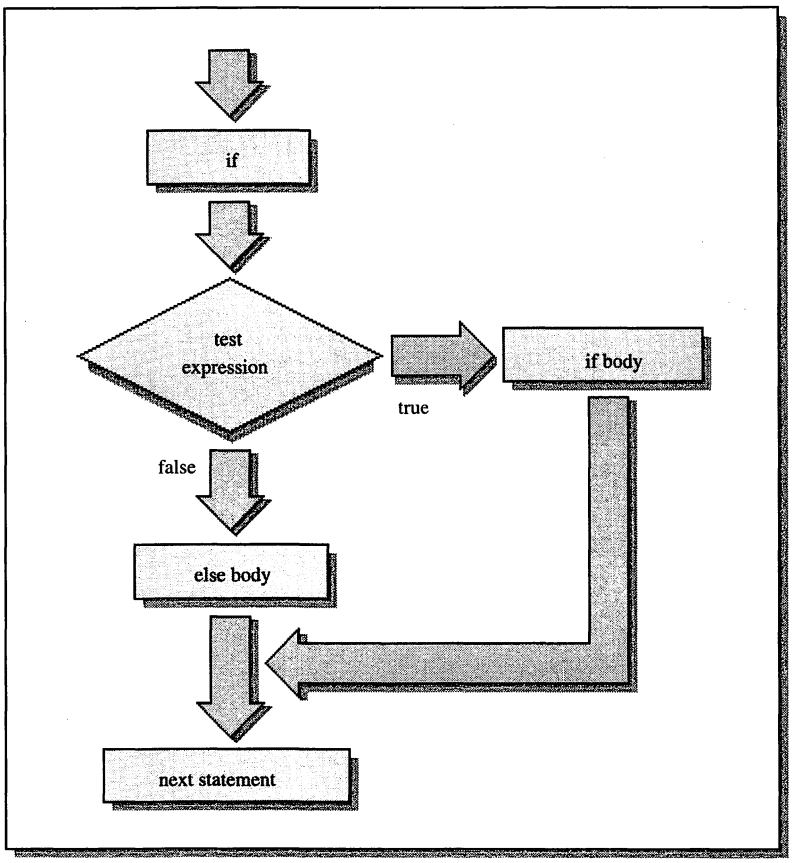


FIGURE 3–5 Program flow of an if-else branch

When an executing program reaches an else-if, it starts at the top and examines each test condition until one of them evaluates to true. Once a test is true, the statement under it is executed and the remainder of the else-if is skipped. While more than one test could be true, only the code under the first test that passes will be executed. Here's an example:

```
if ( days >= 365 )
   DrawString( "\pIt's been at least a year." );
else if ( days >= 7 )
   DrawString( "\pIt's been at least a week." );
else
   DrawString( "\pLess than a week has elapsed." );
```

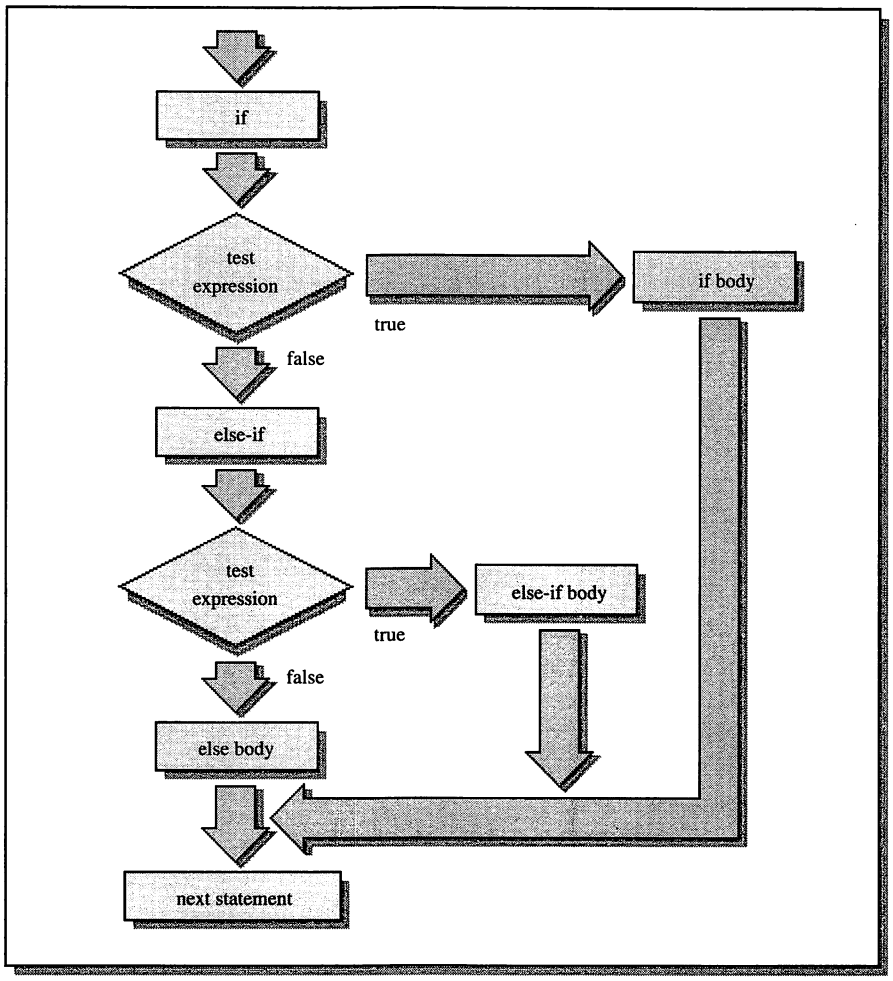


FIGURE 3–6 Program flow of an else-if branch

Figure 3–6 illustrates program flow for a section of code that has an else-if branch in it.

The switch Branch

For situations in which there are only a few possible branch paths, the if statement and its variations work just fine. When you want to handle a single situation that has multiple options, however, you'll find it more convenient to use the switch statement. Here's the format of the switch:

```
switch ( integer variable )
{
   case constant_1:
      statement
      break

   case constant_2:
      statement
      break
   . . .
   . . .

   default:
      statement
      break
}
```

The switch statement compares the value of the variable that appears in parentheses with the constant value that follows each case label. When a match is made, the statement that follows the matching case label is executed. When the break keyword is reached, the program exits the body of the switch; therefore, a break must appear after the last statement under each case label.

If the value of the variable named in the first line of the switch matches none of the case constants, the statement following the optional default keyword is executed. Here's an example of a switch statement:

```
short  card_total;

switch ( card_total )
{
   case 21:
      DrawString("\pBlack Jack!");
      break;
   case 20:
   case 19:
   case 18:
   case 17:
      DrawString("\pStand on 17 - 20.");
      break;
```

```
    default:
      DrawString("\pBust on 22 or higher, hit on 16 or less.");
      break;
}
```

More than one case value can cause a given statement to execute, as the above switch example shows. If `card_total` has a value of 17, 18, 19, or 20, the same DrawString() will execute.

When will the code under the default keyword execute? When `card_total` has a value not specifically listed in a case label. For the above example, that means that a `card_total` value of less than 17 or greater than 21 will cause the default code to execute.

Structures

A solid understanding of the C language struct is important to working with C++. That's because the class, which is the pattern, or definition, of an object, is based on the struct data type. Chapter 5 discusses the class data type in great depth. Here, I review the struct data type so that you'll be well prepared for the C++ class data type.

Defining and Declaring a struct

When a need arises to group several pieces of data together in a common variable, you'll want to use a struct. A *structure template* describes a single type of structure. Here is an example that creates a structure template called AutoDescription. Its purpose is to hold descriptive information about a single car:

```
struct AutoDescription    // template for one automobile
{
   Str255  make;          // Chrysler, Pontiac, etc.
   Str255  model;         // New Yorker, Bonneville, etc.
   long    year;          // 1987, 1993, etc.
};
```

A structure template begins with the C keyword struct, followed by the template name. In the above example the template name, or *tag*, is AutoDescription. A structure is a record of information. The information is

enclosed in the *body* of the structure between the braces that follow the structure name. The body contains the structure *members.* A structure may contain as few or as many members as needed. Each member is a declaration. The AutoDescription structure template has three members: the *make* of the auto, stored as a string—a Str255; the *model* of the auto, which is also a Str255; and the car's *year* of manufacture, which is stored as a long integer.

The definition of a structure template tells the compiler what the structure will look like, but it doesn't create a structure. The structure template is a *type* rather than a variable, just as Str255, short, and float are data types, not variables. What type is it? Whatever name you've selected as the template tag. In the above example, I've created a struct type of AutoDescription.

After you've created a structure template, you can declare variables of the struct type. In the following example, two AutoDescription struct variables are declared. The example also declares a variable of type short—just to show that struct variables can be declared right along with your other variables.

```
struct  AutoDescription  my_junker;
struct  AutoDescription  better_car;
short   num_cars;
```

Accessing struct Members

A structure variable holds several pieces of data, so you need a way to get at a given field. That's called *member access.* To assign a value to a structure member, you use the structure member operator, which is simply a period (.). In the following example, the member operator is used to assign a value to one member of the `my_junker` variable.

```
struct AutoDescription
{
   Str255  make;
   Str255  model;
   long    year;
};

struct  AutoDescription  my_junker;

my_junker.year = 1981;
```

When you apply the member operator to a struct variable, as in the case of my_junker.year, then that struct and that member together act like any variable of the member's type. You can use my_junker.year as you would any long integer variable:

```
struct  AutoDescription  my_junker;
long    the_year;

my_junker.year = 1981;       // assign my_junker.year a value
the_year = my_junker.year; // the_year now has a value of 1981
```

Assigning values to the other two values in the AutoDescription variable is just a little trickier. The problem isn't with the struct itself but, rather, with the way Macintosh C and C++ work with strings. Earlier in this chapter, you saw a routine I named Fill_Str255(). Pass it a string variable and a string, and the routine will fill the Str255 variable with the string. That's how you assign a value to a string—whether the string is a member of a struct or not. The example below uses Fill_Str255() to assign values to the two Str255 members of the my_junker struct variable.

```
struct AutoDescription
{
   Str255  make;
   Str255  model;
   long    year;
};


struct  AutoDescription  my_junker;


Fill_Str255( my_junker.make, "\pChevrolet");
Fill_Str255( my_junker.model, "\pCitation");
my_junker.year  = 1981;
```

Now that the members of the struct variable have been assigned values, those values can be easily drawn to a window:

```
MoveTo( 20, 20 );
DrawString( my_junker.make );
MoveTo( 20, 35 );
```

```
DrawString( my_junker.model );
MoveTo( 20, 50 );
NumToString( my_junker.year, the_str );
DrawString( the_str );
```

Notice that in the above code my_junker.make and my_junker.model are treated just like normal Str255 variables; the Toolbox function DrawString() accepts them as Str255 parameters. The same applies to the year member of the my_junker struct. It's declared as a long variable, and the Toolbox function NumToString(), which requires a long as its first parameter, takes my_junker.year.

It's always good to look at a complete source code example to see how a programming concept works. The following program—aptly named CarStruct—uses the above code to create a struct, assign values to the struct members, and write the member values to a window.

```
// ******************  CarStruct.cp  ******************

struct AutoDescription    // template for one automobile
{
   Str255  make;          // Chrysler, Pontiac, etc.
   Str255  model;         // New Yorker, Bonneville, etc.
   long    year;          // 1987, 1993, etc.
};

struct  AutoDescription  my_junker;  // declare a struct
                                     // variable


void  main( void )
{
   WindowPtr  the_window;
   Rect       window_rect;
   Str255     the_str;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
```

```
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr) -1L,
                            true, 0 );
    SetPort( the_window );

    Fill_Str255( my_junker.make, "\pChevrolet");
    Fill_Str255( my_junker.model, "\pCitation");
    my_junker.year  = 1981;

    MoveTo( 20, 20 );
    DrawString( my_junker.make );
    MoveTo( 20, 35 );
    DrawString( my_junker.model );
    MoveTo( 20, 50 );
    NumToString( my_junker.year, the_str );
    DrawString( the_str );

    while ( !Button() )
       ;
}
```

NOTE

Remember, Fill_Str255() isn't a Toolbox routine. It's a short utility function I've written to fill a variable of type Str255 with a string. You'll find the source code for it in the Characters and Strings section of this chapter.

You'll find the project file and source code for the CarStruct program on the accompanying disk along with the project files and code for every complete example program presented in this book. If you have Symantec C++ 6.0 or 7.0, fire it up and try CarStruct for yourself. When you do, you'll see a window like the one shown in Figure 3–7.

The struct and class Data Types

In Chapter 2, I discussed a C++ data structure called the class. There I said that the class was a template from which objects were created. Though I didn't go into the specifics of how a class was created in C++, I did mention

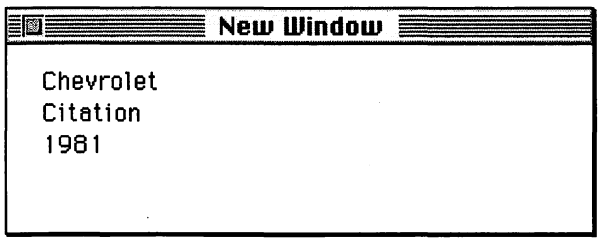


FIGURE 3–7 The output of the CarStruct program

that the class was based on the struct. The example I used in Chapter 2 was for the Acme Bolt and Nut Company. They wanted a program that would keep track of information about each item in their catalogs of hardware parts. Figure 3–8 shows how a class might be set up for such a program.

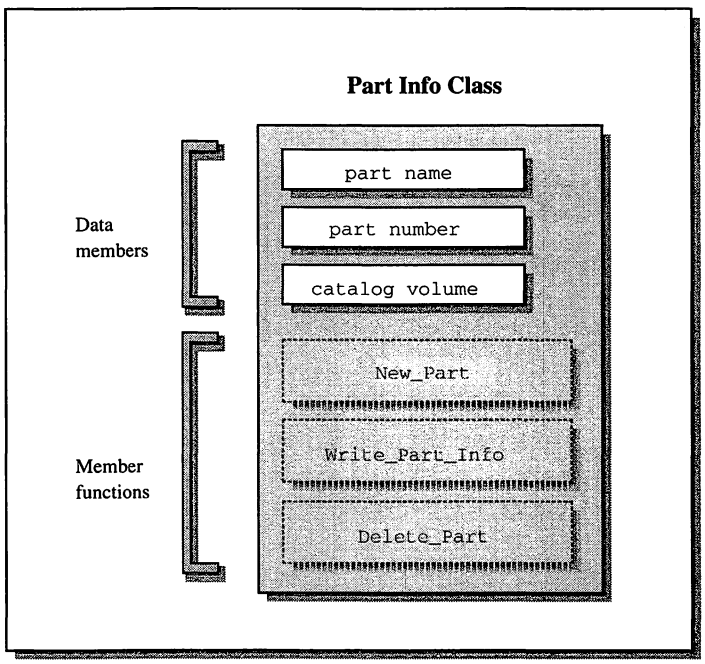


FIGURE 3–8 The struct can be used to represent the data members of a class.

In Figure 3–8, the member functions have been dimmed. That's because I'm going to give an example of how a struct might be used to hold at least some of the information that is shown in the class template. Using a struct, I'll be able to create a data structure that holds the information held in the data members of the class, but I won't be able to include the function information—the member functions—that are contained in the class. Here's a look at the struct I'm talking about:

```
struct CatalogInfo          // template for one catalog item
{
   Str255  part_name;       // the name of the part
   long    part_number;     // the part number
   long    catalog_vol;     // volume of catalog it
                            // appears in
};
```

Now let's create a variable of CatalogInfo type and assign each member a value:

```
struct  CatalogInfo  half_inch_bolt;

Fill_Str255( half_inch_bolt.part_name, "\p½ in. bolt");
half_inch_bolt.part_number = 5002;
half_inch_bolt.catalog_vol = 4;
```

I've tied together the code from the catalog example to make a short program called CatalogStruct. You'll find the source code for it here and on the disk.

```
// *****************  CatalogStruct.cp  *****************

struct CatalogInfo          // template for one catalog item
{
   Str255  part_name;       // the name of the part
   long    part_number;     // the part number
   long    catalog_vol;     // volume of catalog it
                            // appears in
};

struct     CatalogInfo  half_inch_bolt;  // declare a
                                         // struct variable
```

```
void  main( void )
{
   WindowPtr  the_window;
   Rect       window_rect;
   Str255     the_str;


   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   Fill_Str255( half_inch_bolt.part_name, "\p½ in. bolt");
   half_inch_bolt.part_number = 5002;
   half_inch_bolt.catalog_vol = 4;

   MoveTo( 20, 20 );
   DrawString( half_inch_bolt.part_name );

   NumToString( half_inch_bolt.part_number, the_str );
   MoveTo( 20, 35 );
   DrawString( the_str );

   MoveTo( 20, 50 );
   NumToString( half_inch_bolt.catalog_vol, the_str );
   DrawString( the_str );

   while ( !Button() )
      ;
}
```

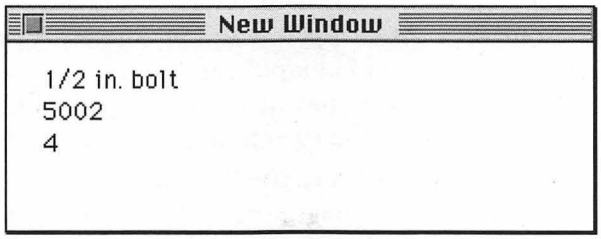


FIGURE 3–9 The output of the CatalogStruct program

Figure 3–9 shows the window that you would see if you ran the CatalogStruct program.

Using a struct, I've succeeded in keeping the catalog information together in one data structure. I've also managed to assign values to the fields of information and print them to the screen. But I wasn't able to bind these operations to the data itself—as the C++ class data structure and object-oriented programming techniques would have allowed me to. To find out how to do that, you'll have to wait until Chapter 5.

Chapter Summary

Because the C++ language is built on the C language, you'll want to have a good understanding of C before you start learning C++.

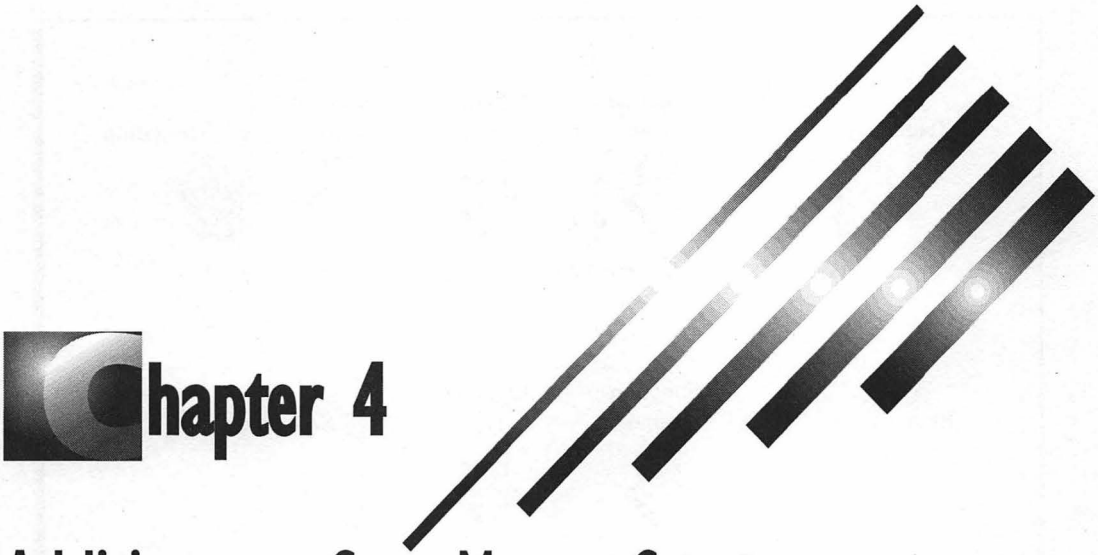
You'll use all of the C data types that hold numbers, including short, int, long, and float, in your C++ programs. To hold strings of text, you'll count on the Str255 data type. If you need to assign a value to a Str255 variable after declaring it, you'll want to write your own short utility function like the Fill_Str255() routine found in this chapter.

Like the Symantec THINK C compiler, the Symantec C++ compiler contains a *preprocessor* that makes a pass through your source code before compiling it. The preprocessor is capable of substituting numbers or characters for symbols. You'll use #define directives to create these symbols. The preprocessor also can replace a single line with the contents of an entire source file; use the #include directive to accomplish that task.

The wealth of operators that are part of the C language are also part of the C++ language. Refer to Table 3–1 to see a list of the most commonly used operators.

C++ code doesn't consist solely of objects; you must write supporting code to make things happen with the objects. A large part of this code will make use of standard C looping and branching statements. The while, do-while, and for loops found in C are also part of C++. The same applies to the if, if-else, else-if, and switch branching statements.

Before tackling C++, you'll want to be sure you have a solid understanding of the C language struct data type. The C++ class data type—the pattern from which objects are created—is based on the struct data type.



Chapter 4

Additions to C... Means C++

C++ is based on the C language. That's good for you, because it means that the time and energy you put into learning C will not have been wasted. And speaking of learning, the period of review is over. Now is the time to buckle down and learn some specifics of C++.

In this chapter, you'll see how C++ allows you to use *function overloading* to create multiple versions of a function. You'll also learn about C++ memory allocation—how it's similar to C and how it's different. Finally, you will read about *scope resolution*—the C++ technique that gives you control in determining which of two identically named variables will be used in a given statement.

The Very Basics

Before delving into the really important differences between C and C++, I'll take just a page or two to cover two very basic differences.

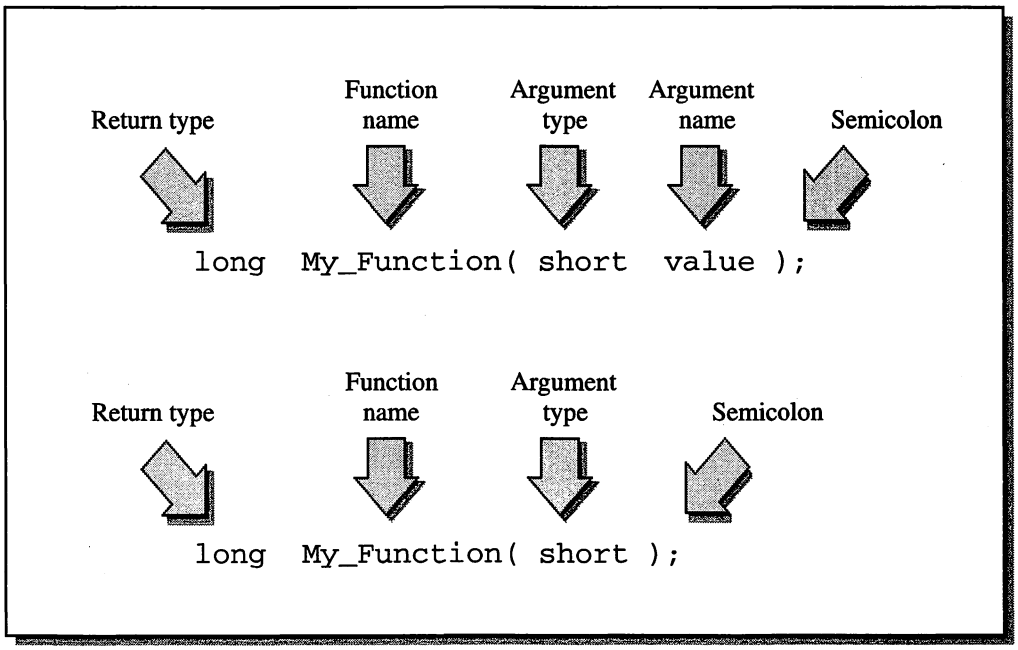


FIGURE 4–1 Two forms of function prototypes

Functions

In C++, functions work just as they do in C. But whereas a C compiler may be forgiving if you don't include function prototypes, a C++ compiler is not. A function prototype tells the compiler what a function will look like—before the compiler ever actually sees the function. A function prototype can have either of two forms, as shown in Figure 4–1.

The only difference between the two forms of prototypes shown in Figure 4–1 is that in one form you list both the type and the name of each argument, while in the other form you list only the argument type. The compiler doesn't care which of the two forms you use—it's simply a matter of preference. In this book, function prototypes will be shown without the argument names, as in the prototype at the bottom of Figure 4–1.

If you haven't used function prototypes in the past, start using them now. Here are a few more examples:

```
void  Write_Warning( void );          // no return value,
                                      // no arguments
void  Write_Message( Str255 );        // no return value,
                                      // one argument
long  Add_Numbers( long, long );      // return value,
                                      // two arguments
```

Comments

You've already seen how comments can be written in C and C++. Because this chapter deals with the features found in C++ but not C, I thought it best to mention the subject again.

In both C and C++, comments begin with /* and end with */. C++ goes C one step better by providing a second method of identifying comments. In C++, you can use a double slash to make a single line a comment. Here is an example of each comment type:

```
/* This line is a comment in both C and C++. As you can
   see, it can occupy more than one line */

// This line is a comment in C++. It's limited to one line
```

Function Overloading

In C, two functions cannot share an identical name. That makes perfect sense. If two functions did have the same name and your source code made a call to one, how would the compiler know which of the two functions to execute? While having multiple functions with the same name is not permissible in C, it is in C++.

Functions with a Different Number of Arguments

In C++, two functions can have the same name as long as they have a different number of arguments. Below you'll see the function prototypes for two functions, both of which are named Draw_Line().

```
void  Draw_Line( void );
void  Draw_Line( short, short );
```

The first of the two functions has no parameters. Calling it results in a line 200 pixels long being drawn. Here's the code for the first Draw_Line():

```
void  Draw_Line( void )
{
   Line( 200, 0 );            // draw a horizontal line
}
```

The second Draw_Line() is a more versatile line-drawing routine. It accepts two parameters. The first is the thickness of the line, and the second is the length of the line. Here's the code for the second version of Draw_Line():

```
void  Draw_Line( short thickness, short length )
{
   PenSize( 1, thickness );  // change the pen size
   Line( length, 0 );        // draw a horizontal line
   PenNormal();              // set the pen to a
                             // thickness of 1,1
}
```

Now let's take a look at two calls to Draw_Line():

```
MoveTo( 20, 20 );
Draw_Line();

MoveTo( 20, 40 );
Draw_Line( 5, 100 );
```

In the above code, the compiler knows which of the two Draw_Line() functions to execute at each Draw_Line() call. The different number of arguments is the key that helps the C++ compiler resolve which function to execute. Figure 4–2 shows that when a call to Draw_Line() is made with two parameters passed to it, the C++ compiler properly determines which function to execute—the Draw_Line() that accepts two parameters.

I've taken the Draw_Line() code and placed it in a C++ program named FunctionOverload. The program first opens a window and then calls Draw_Line() twice. Figure 4–3, shows the results of running Function-Overload. If you have Symantec C++, run the FunctionOverload program included on the disk to verify that the program does indeed execute both functions.

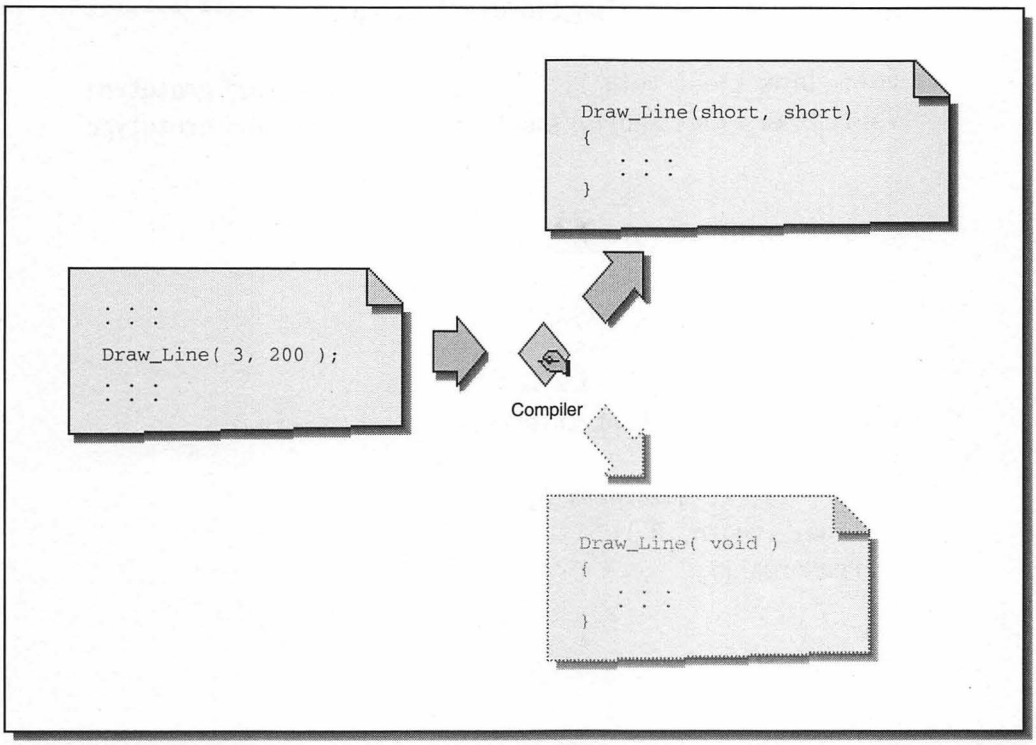


FIGURE 4–2 Function overloading: the compiler determines which function to execute.

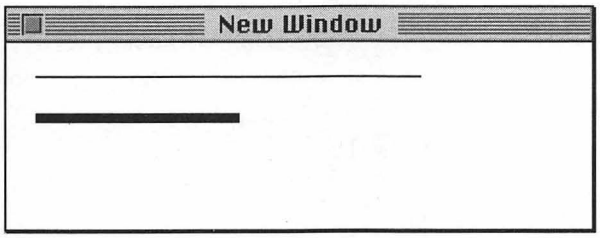


FIGURE 4–3 The output of the FunctionOverload program

```
// ***************  FunctionOverload.cp  ****************

void  Draw_Line( void );              // function prototype
void  Draw_Line( short, short );      // function prototype


void  Draw_Line( void )
{
   Line( 200, 0 );
}

void  Draw_Line( short thickness, short length )
{
   PenSize( 1, thickness );
   Line( length, 0 );
   PenNormal();
}


void  main( void )
{
   WindowPtr  the_window;
   Rect       window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   MoveTo( 20, 20 );
   Draw_Line();          // call one version of Draw_Line()

   MoveTo( 20, 40 );
```

```
    Draw_Line( 5, 100 );  // call a DIFFERENT version of
                          // Draw_Line()

    while ( !Button() )
       ;
}
```

Functions with Different Argument Types

The technique of overloading a function can also be used to create functions with the same name and same number of arguments—provided the arguments are of different types. Determine_Tax_Rate() accepts a float variable that represents a person's income. Based on the value of the passed parameter, the function determines a tax rate and returns it as a float. Here's the function prototype for Determine_Tax_Rate():

```
float  Determine_Tax_Rate( float );
```

Now, what if I want the program I'm writing to also accept a person's income as an integer, and I want the tax rate to be rounded to a whole number rather than a float? If that's the case I'll create a second Determine_Tax_Rate() function that satisfies those requirements. This is the function prototype for the second version of the function:

```
short  Determine_Tax_Rate( long );
```

Each of the two functions accepts a single parameter, but each parameter is of a different type. And that difference is what allows the Symantec C++ compiler to determine which function is executed in response to a call to Determine_Tax_Rate().

NOTE

Note that only the function argument type, not the return type, differentiates one function from another. Overloaded functions must have either different numbers of arguments or different types of arguments (or both). Having different return types alone will not do it. Thus, the following two functions could *not* both be used in the same program—in C or C++.

```
float Convert_String_To_Number( Str255 );
long  Convert_String_To_Number( Str255 );
```

The first function converts a string to a float; the second converts a string to a long. But each has the same number of arguments—one—and the same type of argument—a string. And that's not good.

Why Create Functions with the Same Name?

Function overloading is an interesting, almost mystical, programming technique; it demonstrates just how powerful the C++ compiler is. But that doesn't explain *why* you would want to create different functions with the same name.

For large projects, function overloading minimizes the number of function names you have to keep track of. Mathematical functions are especially good applications for overloading. A simplistic example might involve several functions named Add_Numbers(). Each would be responsible for adding numbers, but the parameters and return types would be different. Consider the following prototypes for four identically named functions:

```
long  Add_Numbers( long, long );
long  Add_Numbers( long, long, long );
float Add_Numbers( float, float );
float Add_Numbers( float, float, float );
```

Below I've shown how these four functions might look:

```
long  Add_Numbers( long a, long b )
{
   return ( a + b );
}

long  Add_Numbers( long a, long b, long c )
{
   return ( a + b + c );
}

float Add_Numbers( float a, float b )
{
   return ( a + b );
}
```

```
float Add_Numbers( float a, float b, float c )
{
   return ( a + b + c );
}
```

Your source code could then use all four, and each would yield different results. Here's a code snippet that calls two of the above functions:

```
long aL, bL, cL, dL;

cL = Add_Numbers( aL, bL );
dL = Add_Numbers( aL, bL, cL );
```

The technique of function overloading has another important use. Objects, created from class definitions, benefit from function overloading—as you'll see later in the book.

Allocating Memory in C

This section discusses the C language method of allocating memory. In particular, I will emphasize how pointers are used with struct variables. C++ does things a little bit differently than C does—so why spend time working with C? Because there are strong similarities between the two. And the class data type, which you'll be working with in Chapter 5, is based on the struct.

Pointer Review

A pointer is a variable that holds the address of a different variable. For the compiler, working with the address of a variable, rather than the variable itself, has certain advantages. Pointers make the passing of data to a function easy, for example. Passing the memory address of data, rather than the data itself, eliminates the need for the receiving function to be concerned with the size of the data.

In Figure 4–4, a function named Do_Stuff() is being called. A pointer to the data held in a variable named `my_data` is being passed. The figure shows that the address of the start of the data, rather than the actual data, is passed. Because Do_Stuff() is receiving a pointer, it isn't concerned with the size of `my_data`.

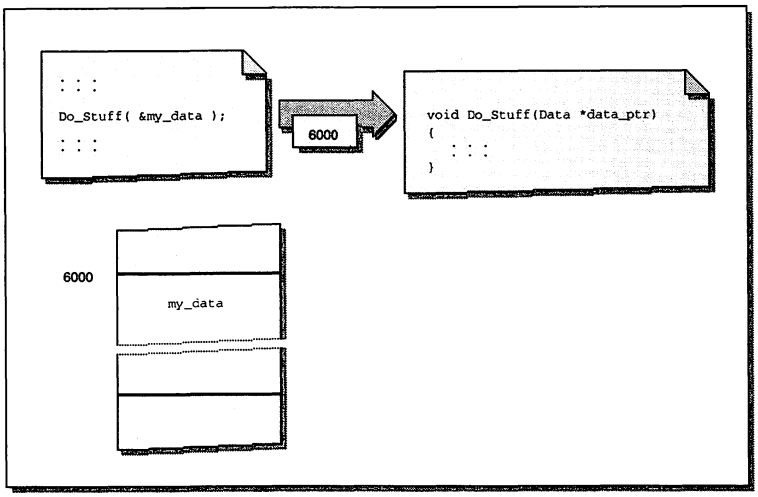


FIGURE 4–4 Passing a pointer to a function

Using Pointers

In C, the first step in reserving, or allocating, memory for data is to declare a pointer to a data type. In a declaration, preceding a variable name with an asterisk tells the compiler that the declared variable is to be a pointer. After declaring a pointer to a data type, you allocate memory by using a standard C routine like malloc(). The following code declares a pointer to a long and then allocates memory to hold one long integer:

```
long  *long_ptr;

long_ptr = (long *) malloc( sizeof(long) );
```

The one parameter that malloc() requires is a number that represents the size of the data to allocate memory for. The sizeof() function is used to provide that number. In the above example, I want to allocate memory for a single long integer. That's why I've asked the sizeof() function to return the size of the long data type. The use of sizeof() in the call to malloc() is shown in Figure 4–5.

Now that malloc() knows how much memory it should allocate, it does so. It then returns a pointer to that memory, as shown in Figure 4–6.

When malloc() returns a pointer to your program, it returns a *generic* pointer—a pointer that has no particular type associated with it. Since I do

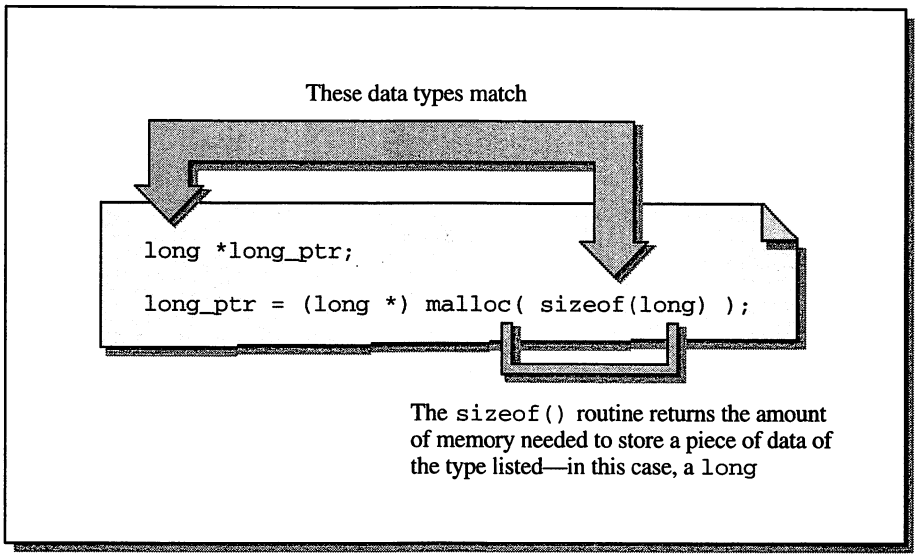


FIGURE 4–5 Using the sizeof() function to determine the amount of memory malloc() should allocate

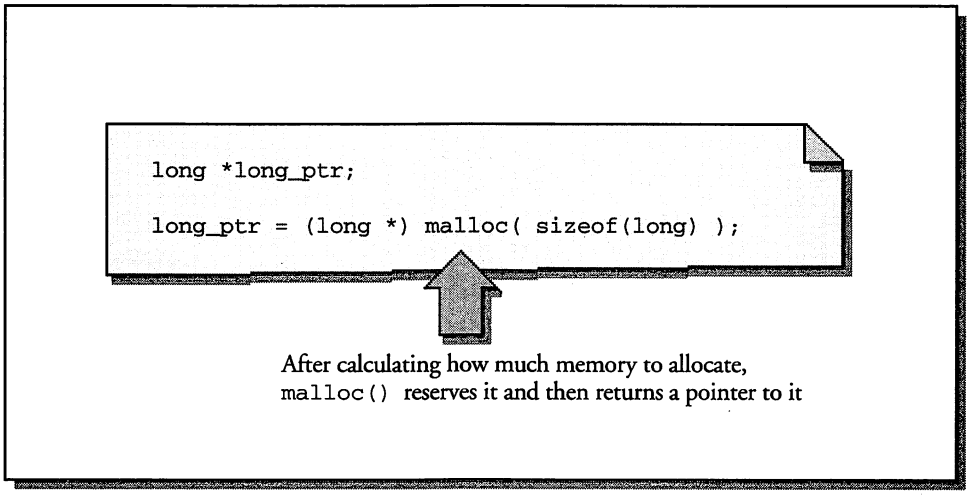


FIGURE 4–6 The malloc() function allocates memory and returns a pointer to it.

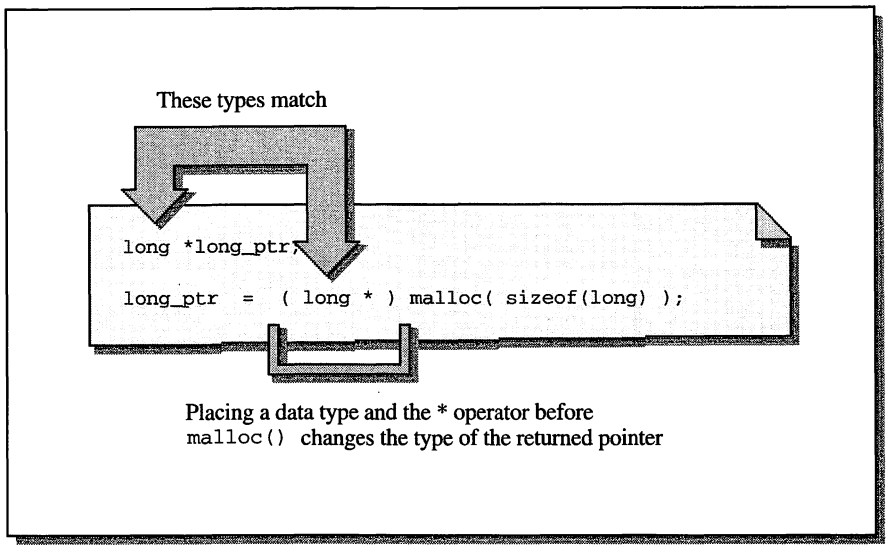


FIGURE 4–7 Typecasting turns the pointer into one that has the data type associated with it.

need a particular kind of pointer—a pointer to a long—I have to typecast the generic pointer to a long pointer. Figure 4–7 shows how this is done.

Figure 4–8 summarizes the above discussion. It shows what memory might look like after each of the following two lines of code:

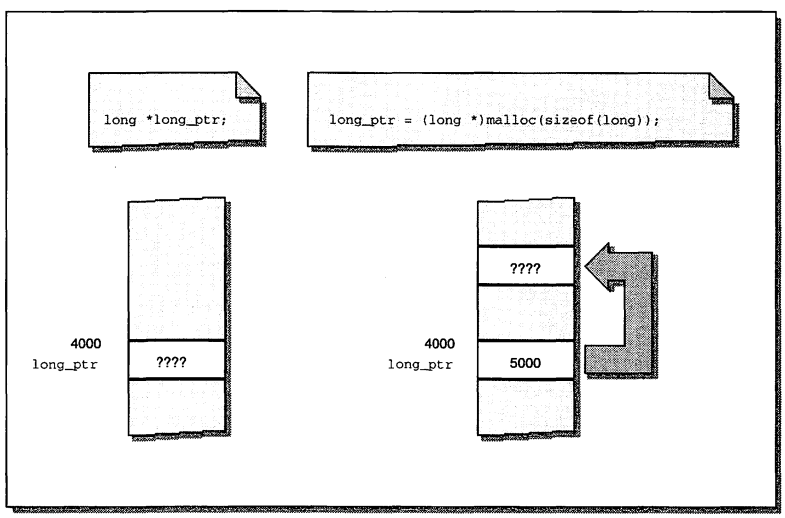


FIGURE 4–8 Memory after the execution of two different lines of code

```
long  *long_ptr;

long_ptr = (long *) malloc( sizeof(long) );
```

On the left side of Figure 4–8 you can see that while the declaration of a pointer sets aside memory for that pointer, the declaration doesn't allocate memory to hold whatever it is that the pointer will eventually point to. For that, you have to call malloc(), as shown on the left side of Figure 4–8.

NOTE

If you use malloc(), you must include the stdlib.h header file. This file tells the compiler what the malloc() function looks like—what its arguments and return type are. Don't worry about the stdlib.h file too much, though. Later in this chapter you'll see how C++ simplifies memory allocation, allowing you to reserve memory without using either stdlib.h or malloc().

Pointers and struct Variables

Pointers aren't used only with variables like the long type. They can also be used when working with struct variables. In Chapter 3, you saw the following structure definition:

```
struct AutoDescription
{
   Str255  make;
   Str255  model;
   long    year;
};
```

This same structure can also be defined using a slightly different syntax:

```
typedef struct
{
   Str255  make;
   Str255  model;
   long    year;
} AutoDescription;
```

The above two forms yield the same result—a new data type called AutoDescription. The second form is advantageous in that it makes it easy to create a pointer type to the structure as well:

```
typedef struct
{
   Str255  make;
   Str255  model;
   long    year;
} AutoDescription, *AutoDescPtr;
```

The following snippet contains a struct definition, along with the definition of a pointer to a struct. Here I'm defining a PersonInfo structure type and a data type called PersonPtr that can be used to point to such a structure:

```
typedef struct
{
   Str255  name;
   Str255  job;
   long    age;
} PersonInfo, *PersonPtr;
```

Here's a second example that defines both a structure and a pointer to the structure:

```
typedef  struct
{
   short  days;
   short  months;
   short  years;
} TimeInfo, *TimeInfoPtr;
```

Now that I've defined my own pointer data type, I can declare a variable of that type:

```
TimeInfoPtr  time_ptr;
```

Before using the `time_ptr` pointer I'll have to write a statement that allocates memory for the pointer to point to:

```
time_ptr = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );
```

NOTE

In the above code, notice that the * operator isn't used in typecasting the pointer. That's because unlike the earlier example of malloc(), which reserved memory for a long, TimeInfoPtr is *already* a pointer data type.

The above example makes `time_ptr` a valid pointer—valid in the sense that it now points to a particular memory location that has been set aside to hold one TimeInfo data structure. To place values in the memory location pointed to by `time_ptr` I would use the `->` operator, like this:

```
time_ptr->days   = 31;
time_ptr->months =  6;
time_ptr->years  =  4;
```

The following program, named GoodCAllocation, shows how a pointer and the malloc() function can be used to allocate memory for a structure. It also uses assignment statements to assign values to the members of the struct—via the pointer to the struct. When it no longer needs the memory allocated by malloc(), the program calls free()—a routine that frees, or releases, the memory that was occupied by the memory allocated using malloc().

The GoodCAllocation program appears on the accompanying disk. Before running it, read the next section to see how you can use the Symantec debugger to better understand what's happening as the code runs.

```
// ***************  GoodCAllocation.cp  ***************

#include <stdlib.h>       // header file for malloc
                          // definition - needed so that
                          // calls to malloc() and free()
                          // are recognized
typedef  struct
{
   short  days;
   short  months;
   short  years;
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  time_ptr;
```

```
void  main( void )
{
   time_ptr = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );

   time_ptr->days   = 1;
   time_ptr->months = 2;
   time_ptr->years  = 3;

   free( time_ptr );
}
```

Using the Symantec Debugger

A source-level debugger like the one included with Symantec C++ is a tool that can be invaluable in tracking down errors in your programs. Watching a program execute while the debugger is turned on is the best way to find out what's going on in a program that uses pointers. For that reason the next two sections of this chapter will show the output of the debugger while code that uses pointers and structs is running. If you need a refresher on how the debugger works, refer to Chapter 1.

Using the Debugger to Verify a Proper Memory Allocation

Pointers and memory allocation are often misunderstood. Here, we'll use the debugger to verify that the following lines of code do in fact set a pointer "pointing in the right direction."

```
TimeInfoPtr  time_ptr;

time_ptr = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );
```

Just a few paragraphs ago, you were introduced to a short program called GoodCAllocation. I'll use that very same source code to peek inside memory and make sure that my pointer is working as intended. If you have the Symantec compiler, follow along (the GoodCAllocation project is on the accompanying disk and is all set up to run with the compiler). If you don't have Symantec C++ you can follow along in Chapter 4 of the Simulator C++ tutorial software.

After selecting Run from the Project menu, you'll see the two debugger windows. Click on the Go button so that the program moves to the breakpoint, as pictured in Figure 4–9. Remember, the line of code that the debugger stops at is the line that is to be executed next; it has not yet been executed. That's why the value of `time_ptr->days` isn't 1 yet.

Next, click on the Step button. That causes the current line to be executed. The assignment to `time_ptr->days` is made, and the days member of the TimeInfo structure takes on a value of 1, as shown in Figure 4–10.

Now the true test. Click on the Step button again. The black arrow in the Source window moves down a line as yet another line of code is executed. And the value of `time_ptr->days`? It remains at 1, as shown in Figure 4–11. Why is this a test of the validity of `time_ptr`? If `time_ptr` hadn't been properly set up, the value of something it pointed to—`time_ptr->days`, for example—might not retain its value. While there is a chance it might hold and maintain the correct value, it might not. You don't want to take a chance to find out. In the next section, you'll see just why this is so.

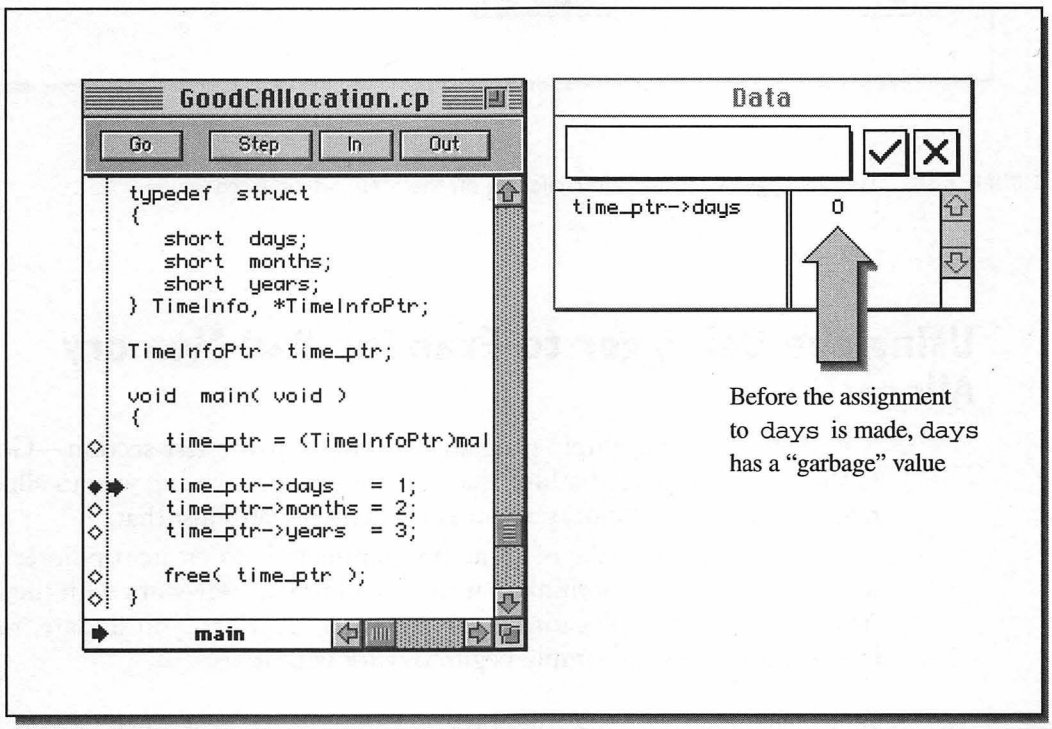


FIGURE 4–9 The debugger windows after clicking on the Go button in the Source window

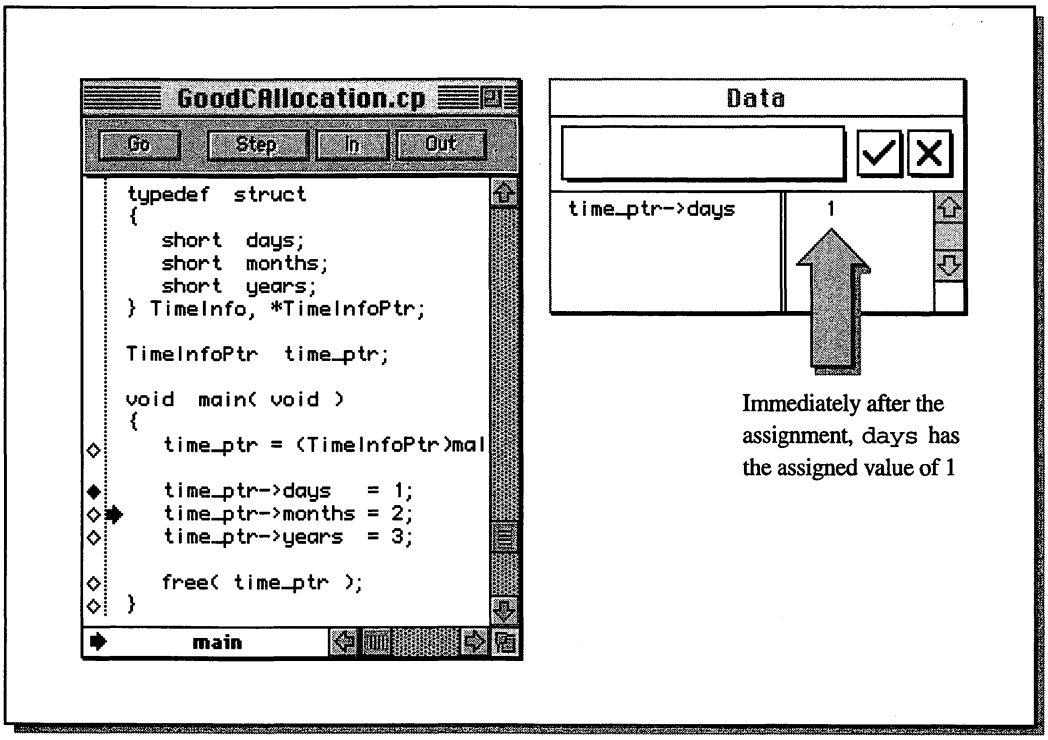


FIGURE 4–10 The debugger windows after clicking on the Step button once

Using the Debugger to Examine Bad Memory Allocation

The name of the example program presented in the last section—GoodCAllocation—provided a hint that there might be a wrong way to allocate memory. The BadCAllocation program demonstrates just that.

A common mistake of some programmers is to create a pointer to a data structure and then think that they can immediately work with the data structure—without allocating memory. After all, when you declare, say, a long variable, you can simply begin to work with it:

```
long  the_long;       // declare a variable of type long

the_long = 5;         // start using it
```

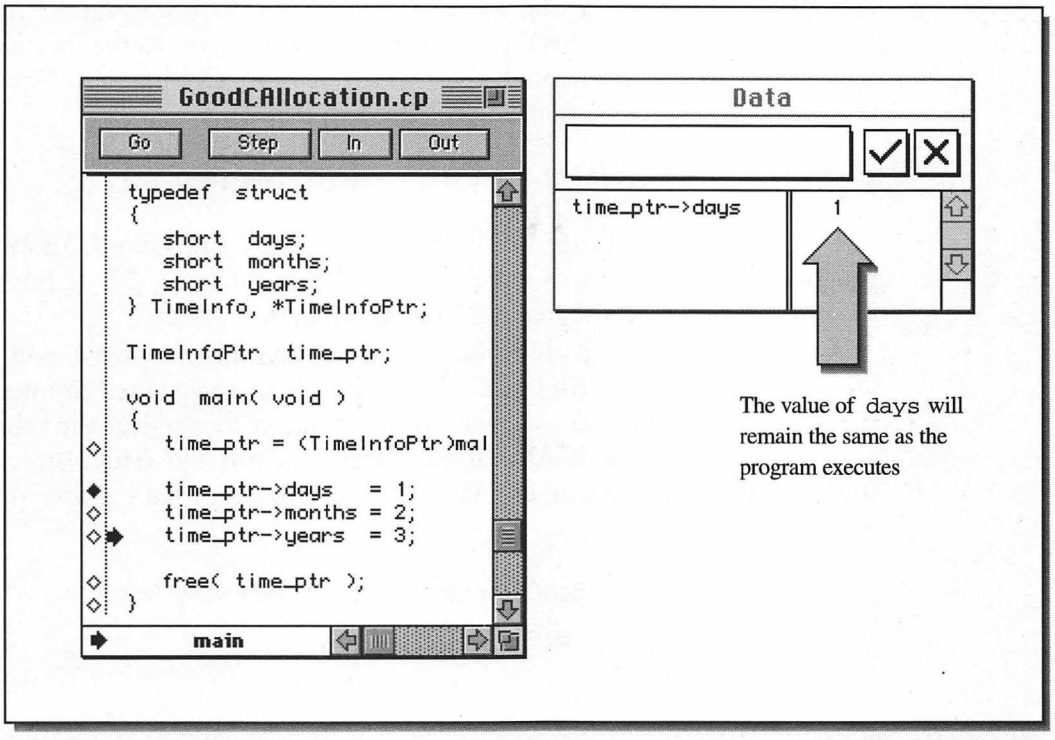


FIGURE 4–11 The debugger windows after clicking on the Step button a second time

When you define a structure data type and a data type that points to such a structure and then declare a variable of the pointer type, it might seem as if you could then begin to work with the pointer type. That's the reasoning that leads some programmers to omit the malloc() statement that allocates memory and assigns the pointer to point to a particular location:

```
TimeInfoPtr  time_ptr;

time_ptr->days = 1;      // time_ptr has not been assigned to
                         // point to a particular memory
                         // location
```

At a later point in a program, when it's time to retrieve the information held in `time_ptr`, there's a good chance that the expected information will not be found. As a program runs, the Mac often shifts the contents of

memory about; that's a one of its normal memory management chores. Only pointers that have been properly initialized to point to the location of a declared variable will be usable after such memory shifting. The line that performs that "proper initialization" is shown below:

```
time_ptr = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );
```

The following program, BadCAllocation, is the GoodCAllocation program with two changes—two lines have been omitted. Those lines are the ones that contain the malloc() and free() calls.

The purpose of BadCAllocation is to demonstrate what happens when a programmer writes code that relies on an unassigned pointer. In the previous section, you used the source debugger to monitor the value of a variable in the GoodCAllocation. After you run the BadCAllocation source code, you'll again use the debugger to see what's going on in memory.

```
// ***************  BadCAllocation.cp  ***************

#include <stdlib.h>

typedef  struct
{
   short  days;
   short  months;
   short  years;
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  time_ptr;

void  main( void )
{
   // malloc() omitted

   time_ptr->days   = 1;
   time_ptr->months = 2;
   time_ptr->years  = 3;
}
```

Incidentally, the BadCAllocation project compiles just fine—with nary an error message. That, in fact, is the reason that invalid pointer usage can occur in the first place. If you have Symantec C++, you can

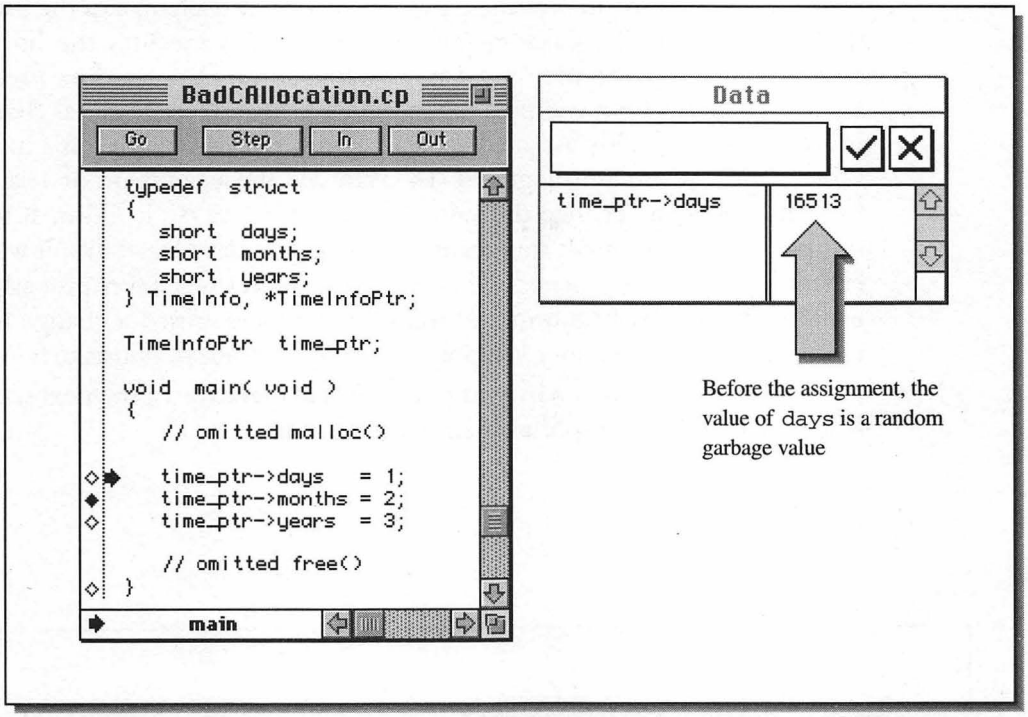


FIGURE 4–12 The debugger windows before clicking on the Go button

compile the included BadCAllocation code yourself. Try running the BadCAllocation example, too. Make sure that Use Debugger is checked in the Project menu. If you don't have Symantec C++, check out Chapter 4 in the Simulator C++ software; it includes a QuickTime movie that animates this lesson.

Now let's see how the BadCAllocation program runs. With the Use Debugger item checked, I'll choose Run from the Project menu. Before I click on the Go button, you can see that the value I'm monitoring, `time_ptr->days`, holds garbage—whatever value was last left at the memory location that `time_ptr->days` now occupies. Figure 4–12 illustrates this.

After clicking on the Go button, the assignment to time_ptr->days is made. The black arrow in the Source window has moved down a line, and the Data window shows that time_ptr->days now has a value of 1. Figure 4–13 shows this.

Now comes the interesting part—the part that really makes the use of a debugger worthwhile. Clicking on the Step button executes the line that assigns a value to `time_ptr->months`. Looking at the Data window, I see that after this step has been completed, the value of `time_ptr->days` has changed. It has lost its previously assigned value of 1 and has returned to some mystery value. This is pictured in Figure 4–14. Why did the assignment of a value to `time_ptr->months` change the value of `time_ptr->days`? It didn't. It wasn't this particular assignment statement that affected `time_ptr->days`; it was the fact that *something* happened. The `time_ptr` pointer was never properly initialized, so any one of a hundred things could have caused a change in the value stored in the memory location that `time_ptr->days` points to.

Figure 4–14 shows why you must allocate memory. In the next section, you'll see how C++ simplifies memory allocation.

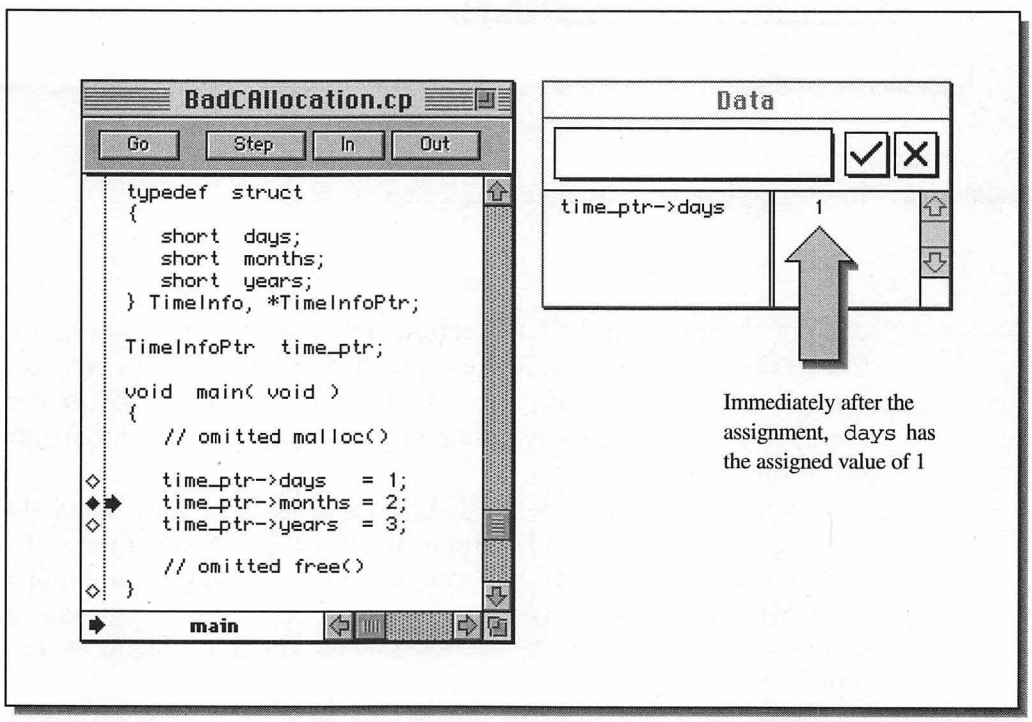


FIGURE 4–13 The debugger windows after clicking on the Go button

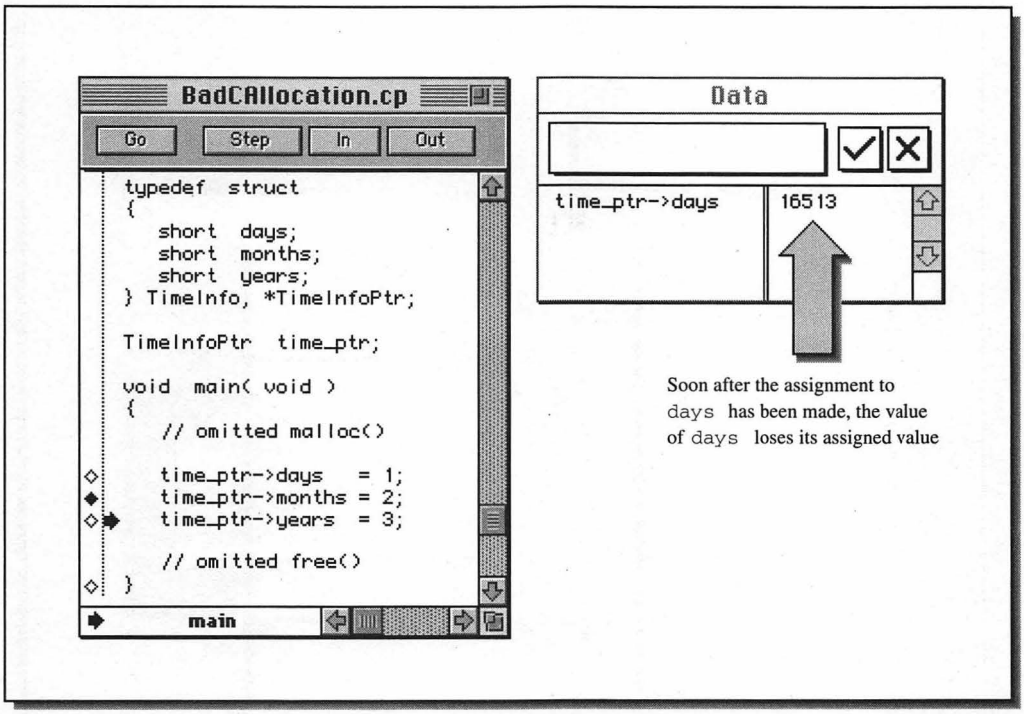


FIGURE 4–14 The debugger windows after clicking on the Step button once

Allocating Memory in C++

In C++, a pointer is initialized through use of the *new* operator. The new operator is unique to C++, so don't page through your C books to find it. Here's an example that declares a pointer to a long and then assigns to the pointer the address of a block of memory:

```
long  *long_ptr;        // declare a pointer to a long

long_ptr = new long;    // obtain memory, set pointer to
                        // point to it
```

Figure 4–15 shows what a section of memory could look like after each of the two above lines of code executed.

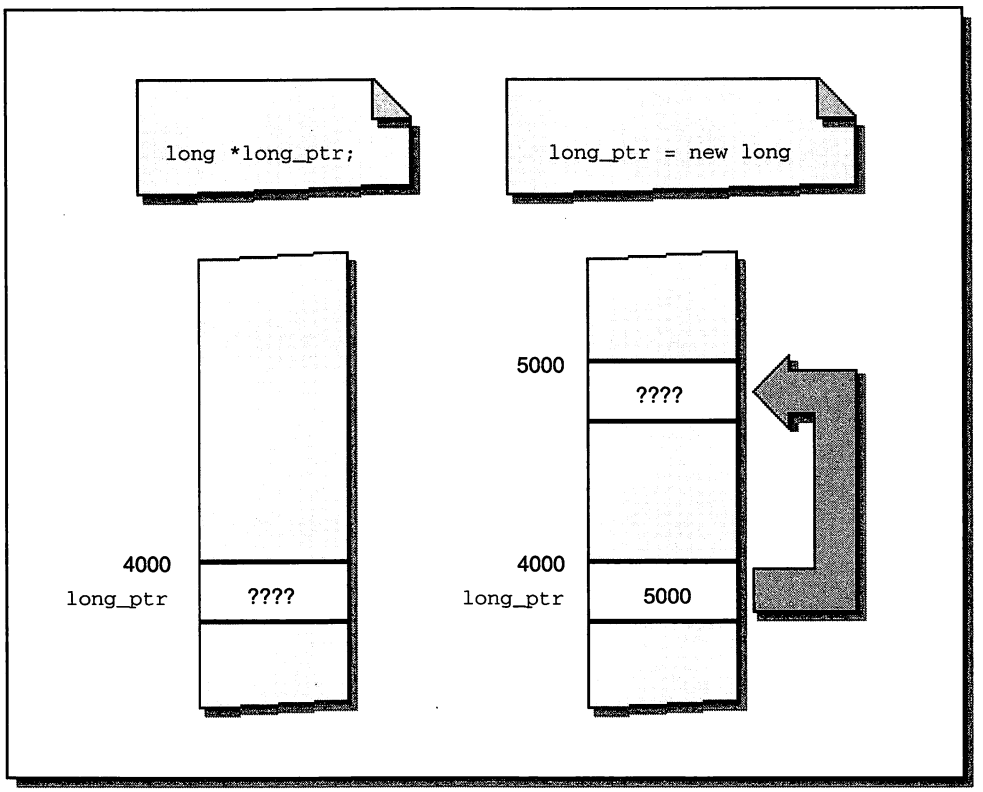


FIGURE 4–15 Memory after declaring a pointer and initializing the pointer using the newer operator

Compare the above allocation that uses the new operator with the way I achieved the same result using C:

```
long  *long_ptr;

long_ptr = (long *) malloc( sizeof(long) );
```

The new operator can be used with any data type—including the struct. Since you are accustomed to seeing the TimeInfo structure, I'll use that struct in this snippet that defines a struct, declares a pointer to it, and then uses the new operator to make the struct pointer valid:

```
typedef  struct
{
   short  days;
   short  months;
   short  years;
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  time_ptr;     // declare a pointer to
                           // struct data

time_ptr = new TimeInfo;  // obtain memory, set pointer
                          // to point to it
```

For comparison, here's how you could achieve the above results in C:

```
typedef  struct
{
   short  days;
   short  months;
   short  years;
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  time_ptr;
time_ptr = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );
```

The new operator obtains, or sets aside, an appropriately sized block of memory and returns a pointer to its starting point. What is the appropriate amount of memory? Whatever amount is needed to hold a variable of the type specified after the new operator. Because a long occupies four bytes of memory, the following lines of code set aside four bytes:

```
long  *long_ptr;

long_ptr = new long;
```

The new operator is always smart enough to determine the amount of memory to obtain—even when it is used with data types that you define, such as the struct. Because a short is stored in two bytes and my TimeInfo struct consists of three shorts, using new to obtain memory for a TimeInfo structure would result in a block of six bytes being reserved:

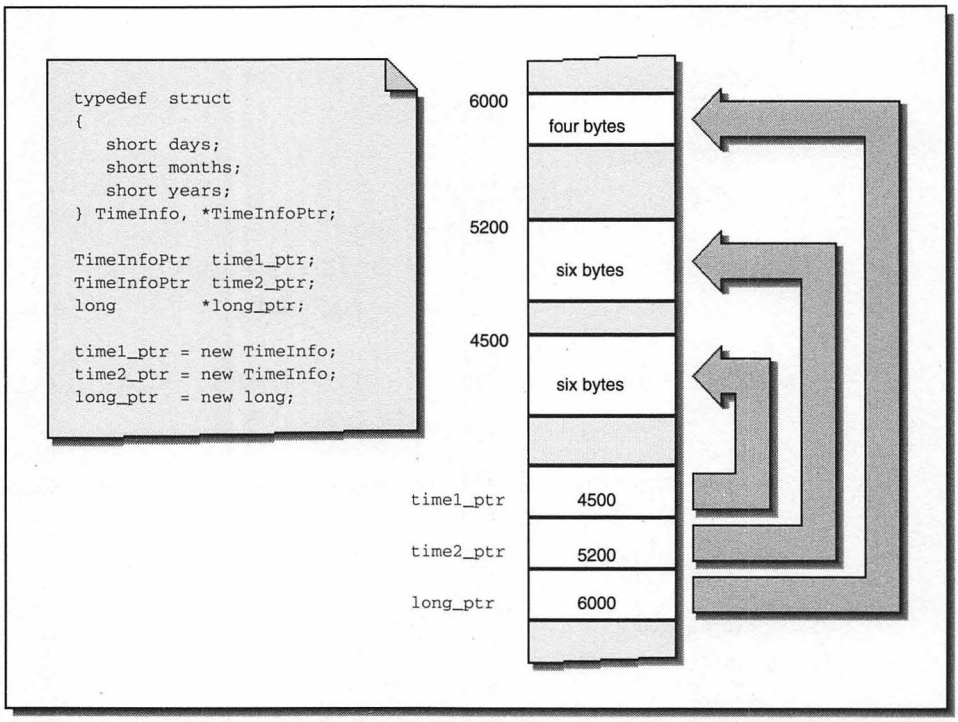


FIGURE 4–16 Memory after using the new operator to initialize three pointer variables

```
typedef struct
{
   short  days;       // two bytes
   short  months;     // two bytes
   short  years;      // two bytes
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  time_ptr;

time_ptr = new TimeInfo;   // reserve six bytes
```

Figure 4–16 shows how memory might look after the declaration of two TimeInfoPtr pointers and a long pointer.

Once you have used new to obtain the memory for a pointer to point to, you can access the pointed to memory just as you did in C. Use the -> operator to access what a pointer points to, as shown here:

```
TimeInfoPtr  time_ptr;

time_ptr = new TimeInfo;

time_ptr->days = 1;
```

When your program is finished with a pointer, the memory allocated by the new operator should be disposed of, or released, by using the delete operator.

```
TimeInfoPtr  time_ptr;

time_ptr = new TimeInfo;        // allocate memory

// do stuff with the pointer

delete time_ptr;                // dispose of memory
```

Disposing of the memory that an unused pointer points to isn't mandatory; your program will continue to run if you don't do it—that is, if it doesn't run out of memory. Until you use delete on an unused pointer, the memory block that the pointer points to cannot be reused by your program. While my forgetting to dispose of the six bytes that an unused TimeInfo structure occupies probably won't matter, forgetting to dispose of several much larger structures may. When you no longer need a data structure, always call delete to return the allocated memory to the pool of free memory available for your program's use.

NOTE

As you've seen, the malloc() and free() library functions do work properly in C++. The preferred method of C++ memory allocation, however, is through the use of new and delete. If you're using malloc() and free(), get in the habit of replacing them with calls to new and delete

The new and delete operators will get a workout with the class data type covered in Chapter 5. Until then, take a look at the NewDelete program to see a simple example of the new and delete operators in action. If you'd like, you can step through the program using the Symantec debugger. If you do, you'll notice that the values to which

time_ptr points will contain garbage—even after the new operator is used. The new operator doesn't place any values in memory; it just allocates the memory. Once time_ptr is used in assignment statements, however, time_ptr will point to valid data—data that remains valid while the program is running.

```
// ******************  NewDelete.cp  ******************

typedef  struct
{
   short  days;
   short  months;
   short  years;
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  time_ptr;

void  main( void )
{
   time_ptr = new TimeInfo;

   time_ptr->days   = 1;
   time_ptr->months = 2;
   time_ptr->years  = 3;

   delete time_ptr;
}
```

IMPORTANT

When memory is allocated using the new operator, it can be freed with the delete operator. But don't try using delete without first allocating memory with new. What will be the result? I don't know. And that's just the point. The program will attempt to free memory that hasn't been allocated to it, and the results will be unpredictable at best, disastrous at worst.

The Scope Resolution Operator

In C, two variables can have the same name—provided they have a different scope. That is, they cannot both be declared within the same function. In the following code, a long variable named days is declared globally—outside all functions—and locally to the main() function. If days is used within main(), which value will the program be working with, 365 or 31? Here's that code:

```
long  days = 365;          // global to entire program

void  main( void )
{
   long  days = 31;        // local to main()
   long  days_in_year;

   days_in_year = days;    // won't work as intended
}
```

The last comment in the above code gives away the answer as to which days is used—the one declared in the main() function. If I was expecting to assign days_in_year a value of 365, I'd be disappointed. In C, the compiler uses the variable that is "closest to home." Because the days_in_year assignment statement is in the main() function, the compiler first checks to see if a days variable has been declared in main(). It has, so it uses the value of that variable—not the globally declared variable as my program intended.

When confronted with multiple variables with the same name, a compiler needs to have a set of rules for determining which variable to use. Otherwise, it can't achieve predictable results. C compilers use the scope of a variable to resolve issues that arise when identically named variables are present. C++ compilers do the same thing. But C++ compilers also provide the programmer with a way to override the compiler's scope resolution rules. By using the *scope resolution operator*, you can specify which variable to use when more than one identically named variable exists. I've repeated the previous example below, with one change:

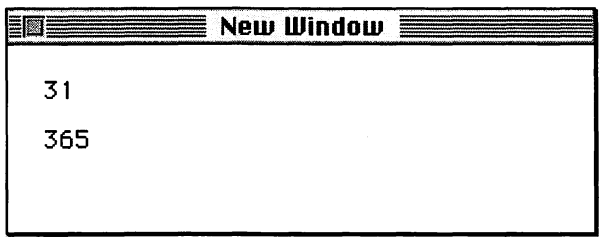


FIGURE 4–17 The output of the ScopeOperator program

```
long  days = 365;          // global to entire program

void  main( void )
{
   long  days = 31;        // local to main()
   long  days_in_year;

   days_in_year = ::days;  // works as intended!!
}
```

By preceding a variable name with two colons—the scope resolution operator—you tell the compiler to look outside the current function. In the above example, the assignment statement that gives `days_in_year` a value tells the compiler to search outside of main() for a variable named `days` and to ignore the `days` variable that is local to main().

The following program, ScopeOperator, is an example of the use of the scope resolution operator. When you run the program, you should see a window like the one shown in Figure 4–17.

```
// ***************  ScopeOperator.cp  ***************

long  days = 365;           // global to entire program

void  main( void )
{
   Str255     the_str;
   WindowPtr  the_window;
   Rect       window_rect;
```

```
    long        days = 31;    // local to main()


    InitGraf( &thePort );
    InitFonts();
    InitWindows();

    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    MoveTo( 20, 30 );
    NumToString( days, the_str );   // uses the local
                                    // variable
    DrawString( the_str );

    MoveTo( 20, 50 );
    NumToString( ::days, the_str ); // uses the global
                                    // variable
    DrawString( the_str );

    while ( !Button() )
       ;
}
```

While the scope resolution operator occasionally comes in handy for working with variables, its greatest power appears when working with the member functions that are a part of each class you define. You'll see plenty of examples of this in the next chapter.

Chapter Summary

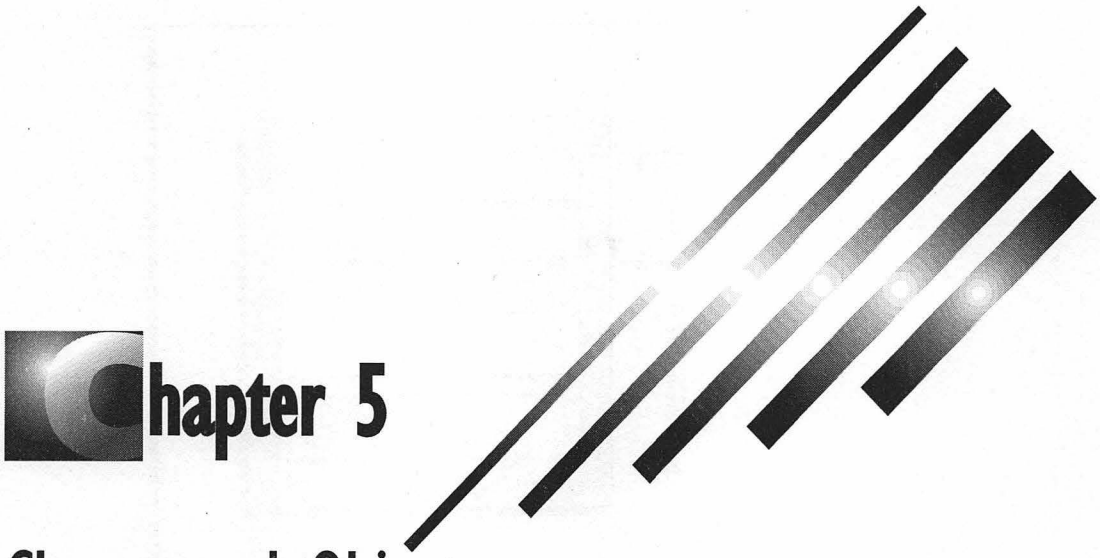
While C++ is based on C, it contains both minor and major enhancements. One of these differences concerns the use of function prototypes. In this

area, C++ is less forgiving than C. If you haven't used function prototypes in the past, you should start using them now. A second change you'll want to adjust to is in the use of comments. C++ uses both the C style comment and its own brand of single-line comments. Provided your text doesn't occupy more than one line, you can preface it by a double slash (//) to turn it into a comment.

In C, two functions cannot share an identical name; in C++, they can. As long as two functions have either a different number of arguments or different types of arguments, the functions can have the same name. This feature of C++ is called *function overloading.*

Memory allocation in C++ differs from allocation in C. For the programmer, this is good news, because the C++ way of doing things is easier. Instead of using the malloc() and free() library functions, you'll simply use the new and delete operators to allocate blocks of memory.

In C, two variables can have the same name only if they have different scopes—that is, they can't reside in the same function. In C++ you can use the *scope resolution operator* (::) to allow the use of identically named variables anywhere in your program.



Chapter 5

Classes and Objects

Chapter 3 was a review of the C language—with great emphasis on working with the struct type. I have already mentioned several times in this book that the C++ class type is based on the C struct type; now, your study of the struct is going to pay off. In this chapter, you'll see the specifics of how classes are written in C++.

A class defines what data an object of that class type will hold and what actions can be performed on that data. Once you have defined a class, you can use it as the basis for creating as many objects as you like, and in this chapter, you'll do just that.

This chapter is one of the longest in the book, and there's good reason for the extra paper; together, classes and objects *are* object-oriented programming.

Declaring a Class

In this section, you'll learn how to declare a class. If you understand how to create a C struct, you're half way to understanding how to create a C++

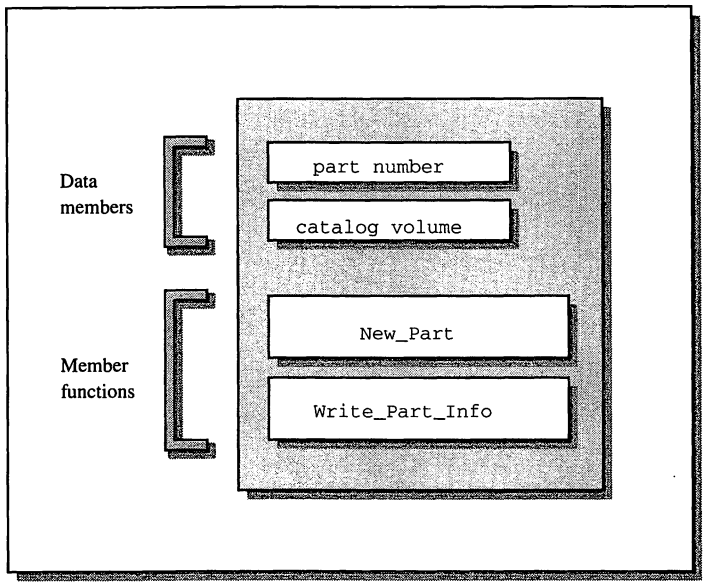


FIGURE 5–1 Representation of an object that holds information about a catalog item

class. Since you're already familiar with Acme Bolt and Nut Company's efforts to write an inventory program, I'll carry on with that example.

For the sake of simplicity, let's assume that Acme will be satisfied with a program that keeps track of only the part number and the catalog in which that part appears. Not only that, but the company will be happy if the program just allows the user to enter information about a new part or print the information about an existing part. With those specifications in mind, an object that defines a single part will look—figuratively—like the object pictured in Figure 5–1.

Using Figure 5–1 as my guide, I created a class type called PartInfo, which is shown here:

```
class  PartInfo
{
   private:
      long  part_number;              // data member
      long  catalog_vol;              // data member

   public:
```

```
    void  New_Part( long, long );    // member function
    void  Write_Info( void );        // member function
};
```

The PartInfo class, like most classes, consists of data members and member functions. Figure 5–2 sheds some light on the format of a class declaration.

Figure 5–2 shows that a class begins with the class keyword, followed by the name of the class. As you would for a struct, you supply a name that describes what the data structure will be used for. The contents of the class lie between braces. As mentioned, the contents of the class are the class data members and member functions. Don't forget to end the class definition with a semicolon after the closing brace.

Both of the data members of the PartInfo class are of type long, but members can be declared to be of any valid C++ type, including short, long,

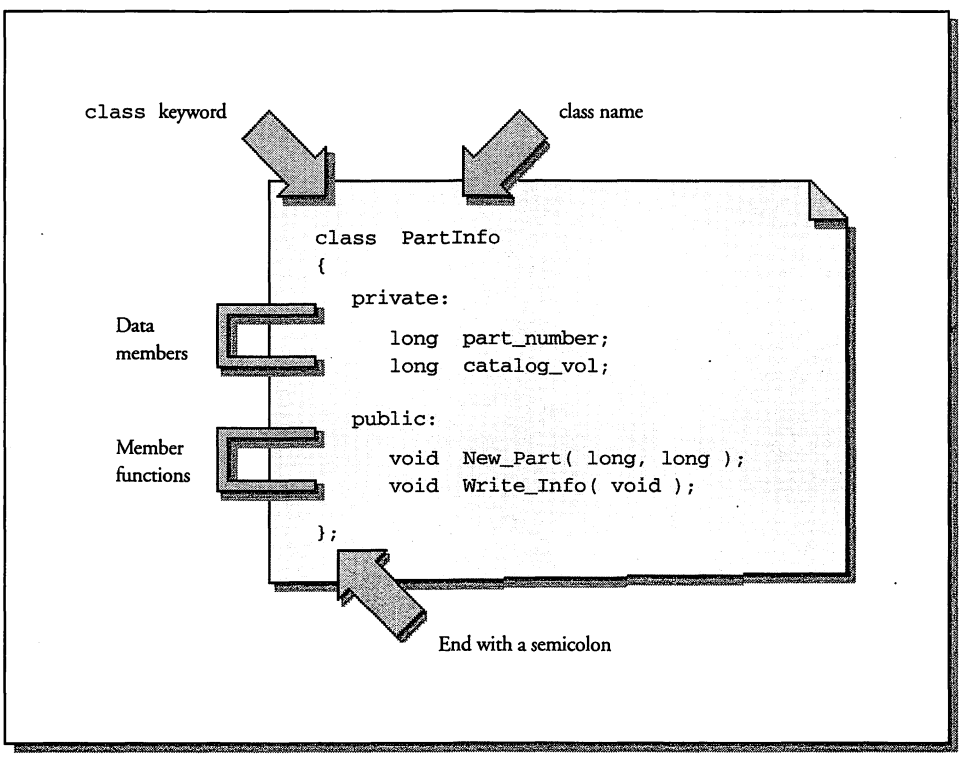


FIGURE 5–2 Declaring a class named PartInfo

Str255, and pointers to those types. Data members are defined in the same way that the members of a struct are defined. And, in fact, they behave much as the members of a struct behave. Member functions, on the other hand, have no struct analogy. They are the part of a class that binds the class data to the actions that are performed on the data, and they are the part of a class that makes a class much more powerful than a struct.

The PartInfo class includes two C++ keywords that I haven't discussed—*private* and *public*. I'll have more to say about these keywords later in this chapter.

Defining Member Functions

The member functions listed in a class declaration are the names of the functions that are to be part of the class; they are not the functions themselves. You must define each member function elsewhere in your code.

Writing the Header of a Member Function

My PartInfo class lists two member functions, so I'll have to write two functions. I've repeated the PartInfo class declaration below, along with the start of the two member functions.

```
class  PartInfo
{
   private:
      long  part_number;                  // data member
      long  catalog_vol;                  // data member

   public:
      void  New_Part( long, long );       // member function
      void  Write_Info( void );           // member function
};


void  PartInfo :: New_Part( long part, long catalog )
{
   // function code here
}
```

```
void  PartInfo :: Write_Info( void )
{
   // function code here
}
```

The very first line of a C function is the function header. The header consists of the function return type, the function name, and the list of function arguments. For a C++ member function, the syntax of the header is a little different—but not much. The difference is the presence of a class name and the scope resolution operator. In Chapter 4, you saw the scope resolution operator used to help the compiler resolve which of two identically named variables it should use. Here's that example:

```
long  days = 365;          // global to entire program

void  main( void )
{
   long  days = 31;        // local to main()
   long  days_in_year;

   days_in_year = ::days;  // use global version of this
                           // variable
}
```

When working with classes, the scope resolution operator is the glue that binds a function to a class. In the header of the Write_Info() member function, note that the name of the class to which the member function belongs appears before the scope resolution operator:

```
void  PartInfo :: Write_Info( void )
{
   // function code here
}
```

Without the :: operator, the compiler would not know that the function was meant to be a part of the PartInfo class. The relationship between a class and one of its member functions is shown in Figure 5–3.

Since the function Write_Info() is listed in the class declaration, you may wonder why the C++ compiler isn't smart enough to make the connection *without* the use of the scope resolution operator. The answer to that

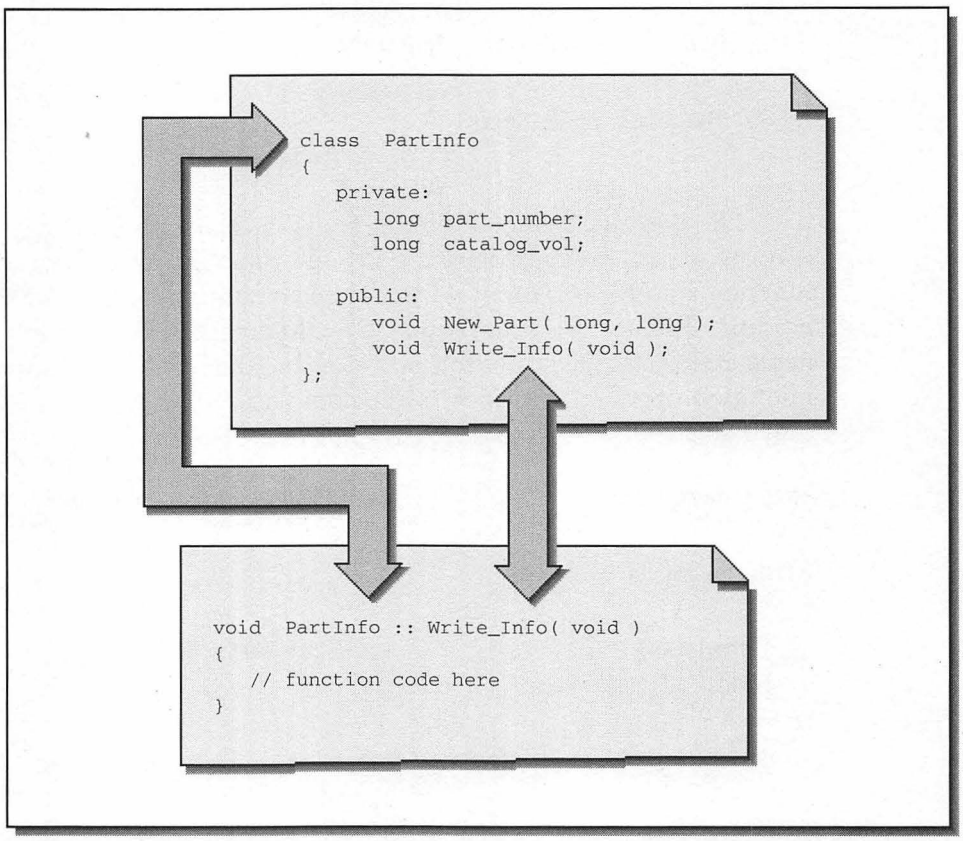


FIGURE 5–3 The relationship between a class and one of the functions listed as a class member function

question lies in a topic covered in Chapter 4—function overloading. Remember, C++ allows more than one function to share the same name. If my program had two Write_Info() functions and the class and scope resolution operator *weren't* used, how would the compiler know which one to bind to the PartInfo class?

NOTE

While many of the functions in a C++ program may be member functions, not all of them are. C++ programs also have "normal" functions. The header of a C++ function that is *not* a member function looks just like a C function header. Later in this book, you'll see example source code that includes both member

functions and regular functions. Already though, you have seen one non-member function—the main() function.

Writing the Body of a Member Function

The body, or contents, of a member function is written such that it performs the required action on the appropriate class data member or members. A member function acts on data members of the same class as the member function. Thus, my Write_Info() member function for the PartInfo class should perform some action on either or both of the PartInfo data members—`part_number` and `catalog_vol`. The purpose of Write_Info() is to write the values of both class data members to the active window, so it works with both data members. Here's that function:

```
void  PartInfo :: Write_Info( void )
{
   Str255  the_str;

   NumToString( part_number, the_str );
   DrawString( the_str );

   Move( 20, 0 );

   NumToString( catalog_vol, the_str );
   DrawString( the_str );
}
```

Write_Info() first converts the value of the `part_number` data member from a long integer to a Str255. That allows the value to be written using the Toolbox function DrawString(). After moving several pixels across the window, the function does the same with the `catalog_vol` data member. Aside from the class name and :: that appear in the function header, Write_Info() looks much like a function found in a C language program.

Notice that in the Write_Info() routine the data members are used as variables. To make use of a data member in a member function you do not have to dereference the data member as you would a struct member. Nor is there any need to specify which class the members are from—that's spelled out in the very first line of the function:

```
void  PartInfo :: Write_Info( void )    // PartInfo class
```

Because this function is a member function of the PartInfo class, the compiler assumes that any name that matches a PartInfo data member name is in fact that data member. Figure 5–4 illustrates this idea.

Now that you've seen the Write_Info() member function, the other PartInfo member function, New_Part(), will make sense:

```
void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}
```

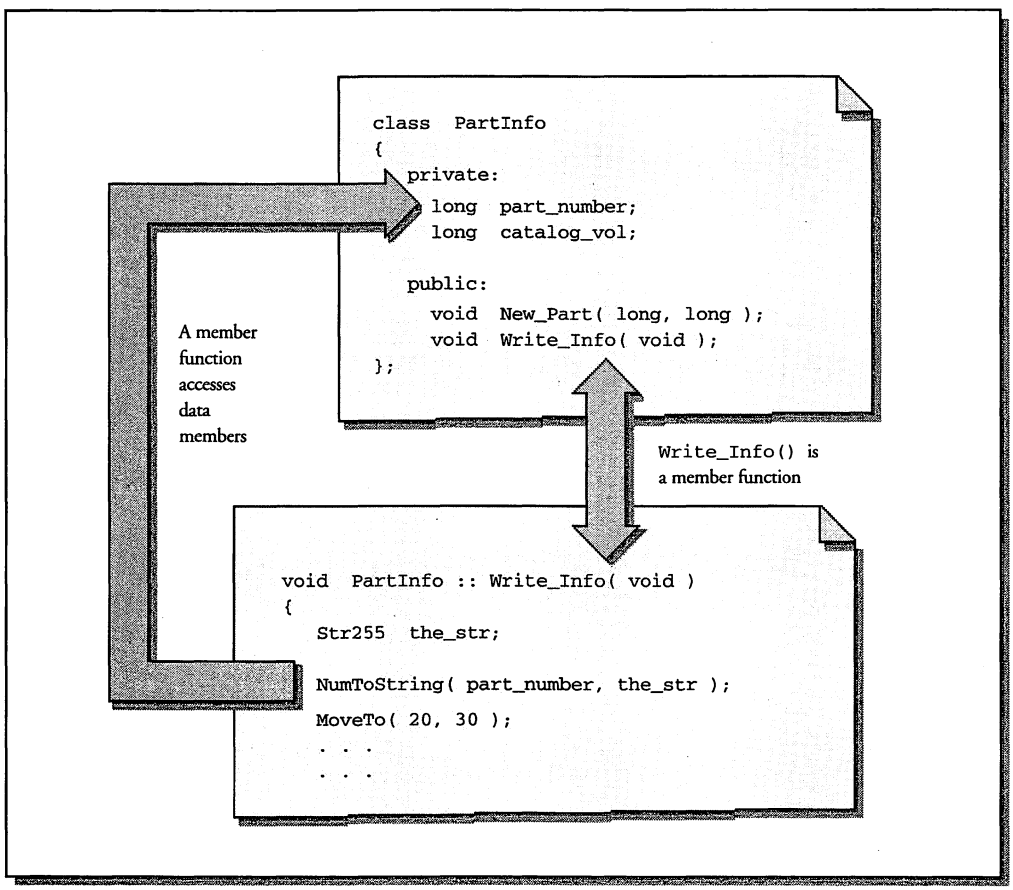


FIGURE 5–4 Class data member names can be used in a member function belonging to that class.

New_Part() is a member function of the PartInfo class, so it contains the name of that class and the scope resolution operator in its header. It also contains something the other member function didn't have—arguments. Member functions, like other C and C++ functions, may have parameters passed to them. The New_Part() function will receive two parameters—both of type long:

```
void  PartInfo :: New_Part( long part, long catalog )
```

The purpose of New_Part() is to add the information about a new hardware part to the Acme company's inventory. The information about a part is held in the two class members of the PartInfo class, so New_Part() will act on these two class members. To do this, the New_Part() function uses two assignment statements:

```
part_number = part;
catalog_vol = catalog;
```

The first line assigns to the class member `part_number` whatever value was passed to New_Part() in `part`. The second line assigns the class member `catalog_vol` the value that was passed in `catalog`. If it appears that New_Part() is acting like a C function, your powers of observation are keen. And if you've guessed that member functions are invoked—and passed parameters—much as C functions are, you're again correct. The next section provides more details about the calling of member functions.

Working with Objects

A class declaration is a pattern that defines what a C++ object will look like; it does not, however, actually create an object. This section will examine how to create and work with objects.

Declaring an Object

In the past I've likened C++ classes to C structures. I'll do that again here as I discuss how objects are created.

In C, you first set up a struct template. You then declare variables of that struct type or variables that are pointers to that struct. The following example defines a struct template that holds audio CD information. It consists of two members—Str255 types that hold the title of the CD and the name of the musical group. After defining the structure, a struct pointer

variable is declared. Then memory is allocated so that the pointer points to something.

```
typedef  struct
{
   Str255  title;
   Str255  band;
}; CDInfo, *CDInfoPtr;

CDInfoPtr  the_CD;

the_CD = ( CDInfoPtr ) malloc( sizeof( CDInfo ) );
```

Figure 5–5 shows the above code snippet and a section of memory. The left side of Figure 5–5 shows how memory looks after the struct template is declared. The right side of the figure shows memory after the variable is declared and memory is allocated.

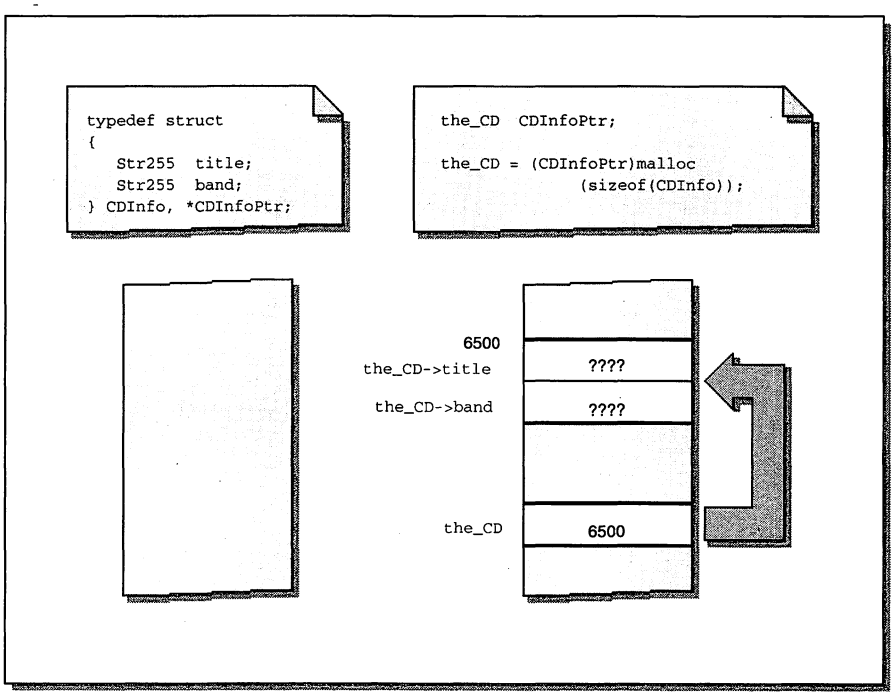


FIGURE 5–5 Memory is allocated for a struct using malloc()—but not for a struct type declaration.

The point of Figure 5–5 is that the definition of a struct template uses no memory. The definition is just an indicator of what variables of the struct type will eventually look like. Not until a variable of type struct is declared is memory set aside. A similar situation occurs when defining class templates in C++.

The definition of a class—like the definition of a struct—reserves no memory. To allocate memory you must create an *instance* of the class. An instance of the class is more commonly called an *object*. The following code snippet uses PartInfo as the class definition from which an object is created:

```
class  PartInfo
{
   private:
      long  part_number;              // data member
      long  catalog_vol;              // data member

   public:
      void  New_Part( long, long );   // member function
      void  Write_Info( void );       // member function
};

PartInfo  *the_bolt;          // a pointer to an object

the_bolt = new PartInfo;
```

NOTE

Instance and *object* mean the same thing. For consistency, I'll use *object* in this book. Be aware that in other books you may see objects referred to as instances.

To create an object, first declare an object pointer. That simply involves declaring a pointer to the class. Then use the new operator to allocate the memory for the object. In C++, the new operator is used instead of malloc(). Recall from Chapter 4 that the new operator, followed by a data type, sets aside an amount of memory equal to the size of that data type. The new operator also returns a pointer to the allocated memory. Here PartInfo is the data type, and `the_bolt` is the pointer that holds the address of the memory reserved for the new object.

IMPORTANT

A variable declared to be of type struct is called just that—a struct variable. A variable declared to be of type class, on the other hand, is called by its own special name—an object.

Figure 5–6 shows how memory looks after the class template is declared and after memory is set aside for a new object.

As in the case of the declaration of the struct template, a declaration of a class template reserves no memory. Figure 5–6 emphasizes that point. Memory allocation takes place when an object is created via use of the new operator, as Figure 5–6 shows. You may have noticed that I've left the memory contents in Figure 5–6 a bit unclear. The figure doesn't specifically show how the memory devoted to the object is divvied up. That will become clear in the next section when you take a close look at how member functions are accessed.

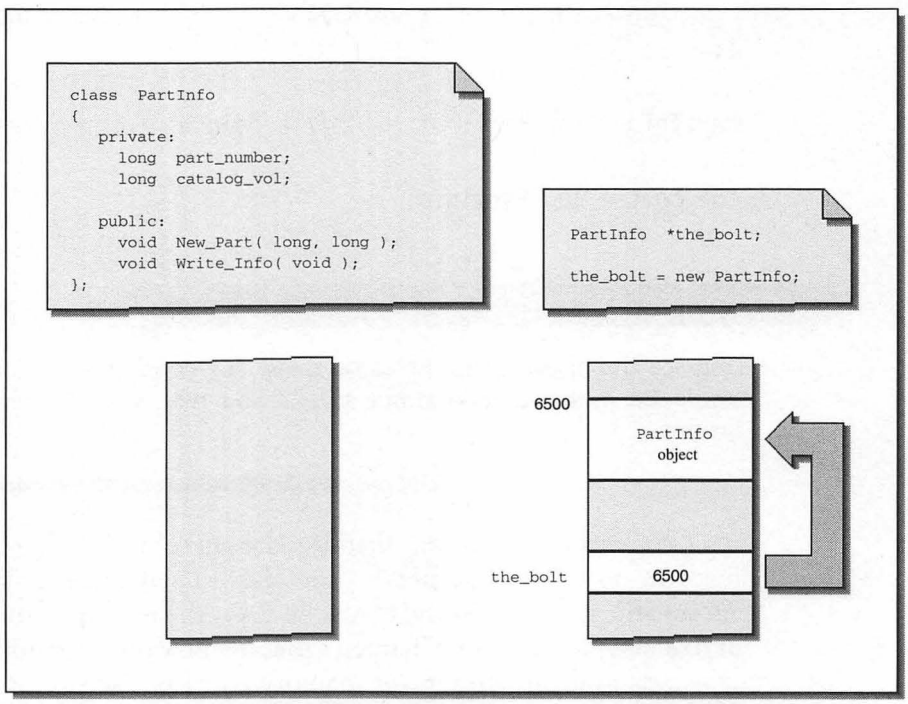


FIGURE 5–6 Memory is allocated for an object, but not for a class declaration.

Objects and Member Functions

Earlier in this chapter, you saw that you must define each member function in your code. When an object is created, pointers to the member functions are also created. The member functions that are a part of an object are simply pointers to functions; they are not the functions themselves.

In Figure 5–7, a single object of the PartInfo class type is created using the new operator. The object pointer, `the_bolt`, points to the memory that holds one object. The object itself consists of two longs—`part_number` and `catalog_vol`—and two pointers—a pointer to the code that makes up the New_Part() member function and a pointer to the code that makes up the Write_Info() member function. Figure 5–7 doesn't show exactly where in memory the code that makes up these functions appears. Nor does it show the address of the object pointer or the object itself. You need not be concerned about where these things end up in memory. After all, it's the pointer's job to keep track of these things—not yours. You might want to note, however, that an object's data members are data, while an object's member functions are pointers to the actual functions.

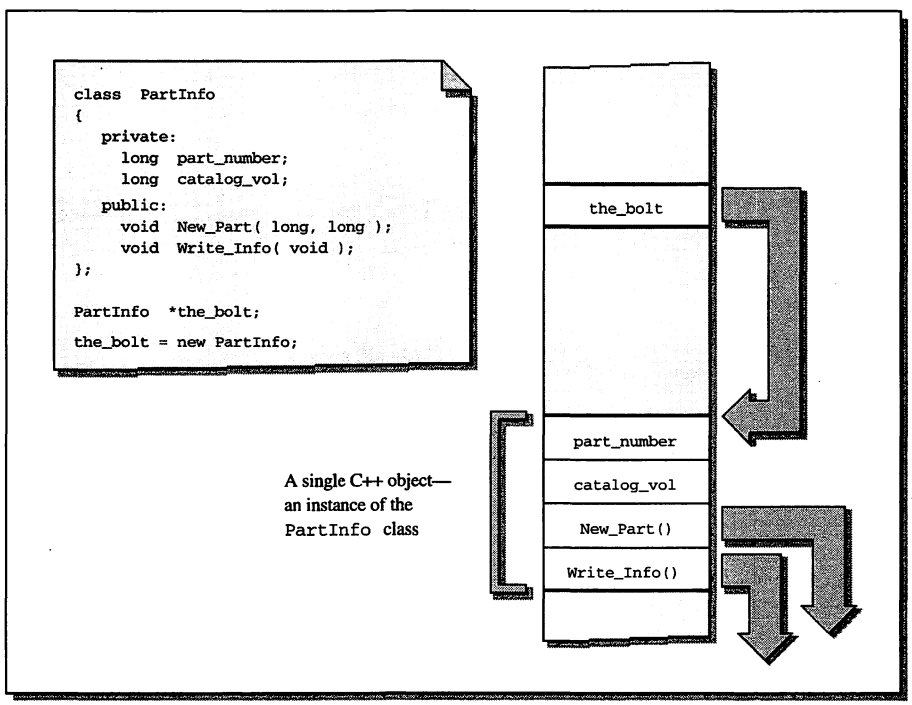


FIGURE 5–7 An object and the pointer that points to it in memory

Once a class is defined, you can create as many instances, or objects, of that class as you wish. Below I've declared two pointers to objects of the PartInfo class and then used the new operator to allocate memory and set the pointers to point to that memory.

```
PartInfo  *the_bolt;          // first pointer to a
                               // PartInfo object
PartInfo  *the_washer;        // second pointer to a
                               // PartInfo object

the_bolt   = new PartInfo;    // allocate first object
                               // memory
the_washer = new PartInfo;    // allocate second object
                               // memory
```

Figure 5–8 shows how memory will look after two PartInfo objects have been created. Note that while each of the two objects has its own `part_number` and `catalog_vol` data members, they both point to the same New_Part() and Write_Info() member functions.

An object needs its own data members so that it can hold information specific to itself. In the catalog example I've been using, each object represents a hardware part, or item, in the catalog. So each object must keep track of the part number of a single part. On the other hand, the actions performed on an object, which are represented by the object's member functions, can be common to all the objects of a class. For that reason, all objects of a class can make use of the same functions. That's why a single function like New_Part() is loaded into memory and used by all PartInfo class objects.

Invoking a Member Function

A member function is invoked using the -> operator. After an object is created, follow the object's name with the -> operator and the name of the member function to call. If the member function requires that parameters be passed to it, include them as you would in a normal function call. The next bit of code I show calls the New_Part() member function. Here's a reminder of what that function looks like:

```
void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}
```

The following example declares a PartInfo object (actually, a pointer to an object), allocates the memory for the object, and then calls the New_Part() member function:

```
PartInfo  *the_bolt;

the_bolt = new  PartInfo;

the_bolt->New_Part( 5002, 4 );
```

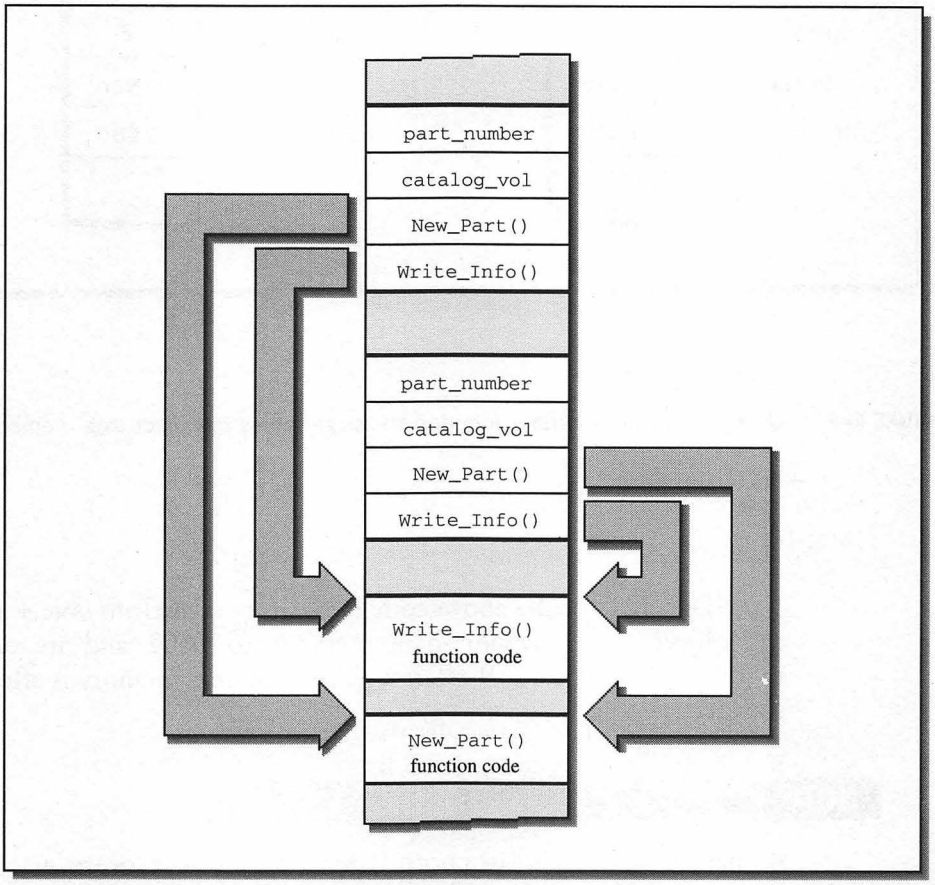


FIGURE 5–8 Two objects in memory and the functions they point to

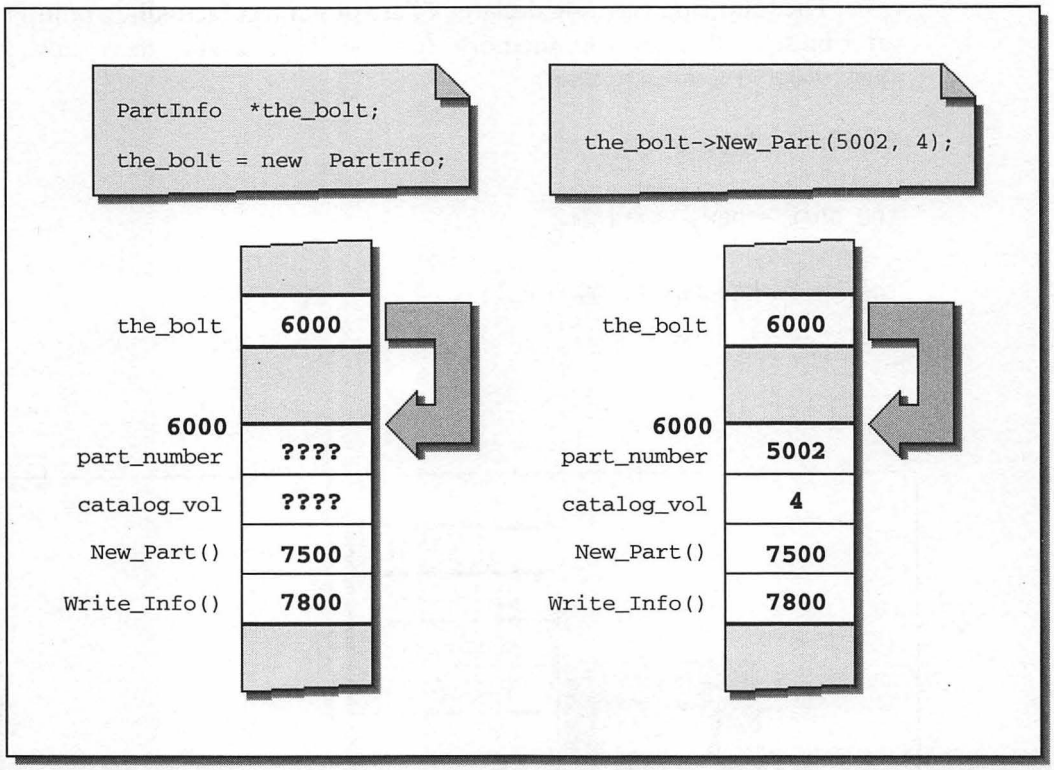


FIGURE 5–9 Using an object's member function to assign values to object data members

The effect of the above code is to create a PartInfo object and then set the object's `part_number` data member to 5002 and its `catalog_vol` member to 4. Figure 5–9 offers a glimpse of how memory is affected by this snippet of code.

NOTE

A member function is like a normal function—except for the manner in which it is invoked. You call a member function only via an object. In Chapter 2, I said that a program tells an object to carry out an action by sending it a message. The message tells the object which member function should execute. A line

like the following is then sending a message to the PartInfo object the_bolt. And the message is? To execute the New_Part() function.

```
the_bolt->New_Part( 5002, 4 );
```

The program MemberFunctions, found on the accompanying disk, is an example of a program that makes use of a class and an object. It declares a class and its member functions, creates an object, and then invokes the object's member functions. The MemberFunctions program invokes the New_Part() member function to assign values to the object's two data members and then calls the object's Write_Info() member function to write that information to the active window. Figure 5–10 shows what the program window looks like after running MemberFunctions.

```
// ***************  MemberFunctions.cp  ****************

class  PartInfo
{
   private:
      long  part_number;              // data member
      long  catalog_vol;              // data member

   public:
      void  New_Part( long, long );   // member function
      void  Write_Info( void );       // member function
};
```

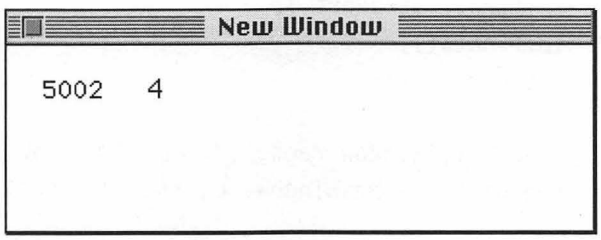


FIGURE 5–10 The output of the MemberFunctions program

```
void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}

void  PartInfo :: Write_Info( void )
{
   Str255  the_str;

   NumToString( part_number, the_str );
   DrawString( the_str );

   Move( 20, 0 );

   NumToString( catalog_vol, the_str );
   DrawString( the_str );
}


PartInfo   *the_bolt;    // declare an object of type
                         // PartInfo
PartInfo   *the_washer;  // declare an object of type
                         // PartInfo

void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr) -1L,
                           true, 0 );
```

```
    SetPort( the_window );

    the_bolt = new PartInfo;        // allocate object memory

    the_bolt->New_Part( 5002, 4 ); // invoke member function

    MoveTo( 20, 30 );
    the_bolt->Write_Info();         // invoke member function

    while ( !Button() )
        ;
}
```

Objects and Member Functions—Round Two

In previous sections, you saw objects being declared through the use of pointers. That's how I'll be declaring them throughout this book. For the sake of completeness, however, I'll mention here that objects can also be declared directly—without using pointers. Again, it's similar to the way in which you work with structs:

```
typedef  struct
{
    short  days;
    short  months;
    short  years;
} TimeInfo, *TimeInfoPtr;

TimeInfoPtr  the_time;
TimeInfo     another_time;

the_time = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );

the_time->days = 31;

another_time.days = 28;
```

In the above example, you can see that the information for a single struct can be stored in memory using malloc() and then accessed using a pointer—the_time:

```
TimeInfoPtr  the_time;

the_time = ( TimeInfoPtr )malloc( sizeof( TimeInfo ) );

the_time->days = 31;
```

Additionally, information for a struct can be kept track of by declaring a variable of the struct type itself. This type of declaration allocates memory without the use of malloc():

```
TimeInfo  another_time;

another_time.days = 28;
```

NOTE

Since declaring a struct variable eliminates the need for using malloc(), why not just always use struct variables rather than struct pointers? Because with struct variables, you must know in advance how many structs you'll be working with. Then you declare a variable for each. It's more advantageous to use pointers, because you can work with them dynamically—that is, as the user works with your completed program, memory is allocated and released using malloc() and free() or new and delete.

Classes work the same way. Although I've been working with pointers to an object, you can declare objects directly.

```
class  PartInfo
{
   private:
      long  part_number;              // data member
      long  catalog_vol;              // data member

   public:
      void  New_Part( long, long );   // member function
      void  Write_Info( void );       // member function
};

PartInfo  *the_bolt;
```

```
PartInfo   another_bolt;

the_bolt = new PartInfo;

the_bolt->New_Part( 5002, 4 );

another_bolt.New_Part( 5003, 5 );
```

NOTE

The direct selection, or dot, operator (.) is used to access members through a class variable. The indirect operator (->) is used to access members using a class pointer just as it is for structs.

Using class pointers and the new operator has advantages over working directly with objects. These advantages will become evident in later chapters.

Deleting an Object

Chapter 4 mentioned the new and delete operators. There you saw how both of these C++ operators could be used on pointers:

```
TimeInfoPtr  time_ptr;

time_ptr = new TimeInfo;      // use new to allocate memory

// do stuff with the pointer here

delete time_ptr;              // use delete to dispose
                              // the pointer
```

You've seen that the new operator is used to allocate memory for an object. And by now you've probably guessed that when you have finished with an object, it's the delete operator that is used to dispose of it.

```
PartInfo  *the_bolt;              // declare an object

the_bolt = new PartInfo;          // allocate object memory
```

```
// do stuff with the object here

delete the_bolt;                    // delete the object
```

NOTE

Keep in mind that when you delete, or dispose of, a pointer you are not actually deleting memory. You're freeing up the memory that the pointer referenced—making it available for future use by your program.

Multiple Objects

Once a class is declared, your program can create as many objects of that class type as it needs—that's one of the powers of object-oriented programming. As you saw several pages back, to create two objects you simply declare two pointers and use the new operator twice:

```
PartInfo  *the_bolt;          // first  pointer to a
                              // PartInfo object
PartInfo  *the_washer;        // second pointer to a
                              // PartInfo object

the_bolt   = new PartInfo;    // allocate first object
                              // memory
the_washer = new PartInfo;    // allocate second object
                              // memory
```

While each of the two objects has its own data members, they both share the same member functions. To assign each object's data members values, invoke the New_Part() member function for each:

```
the_bolt->New_Part( 5002, 4 );
the_washer->New_Part( 37, 3 );
```

Looking at the code for the New_Part() member function, you might wonder how the function knows which object's data members should be assigned values. After all, neither of the function's two assignment statements mentions a particular object, and no object is passed as a function parameter:

```
void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}
```

The New_Part() function knows which object's data members to work with by the way it is invoked. New_Part() performs its actions on the data members that belong to the object whose name is used in the function call. Remember, you must preface the function call with the name of an object and the -> operator. You can't just directly call New_Part()—or any other member function:

```
New_Part( 5002, 4 );              // won't compile -  won't work

the_bolt->New_Part( 5002, 4 );  // this is correct
```

Figure 5–11 shows how a call to the_bolt->New_Part() causes data members belonging to the `the_bolt` object, not the data members of the `the_washer` object, to be affected.

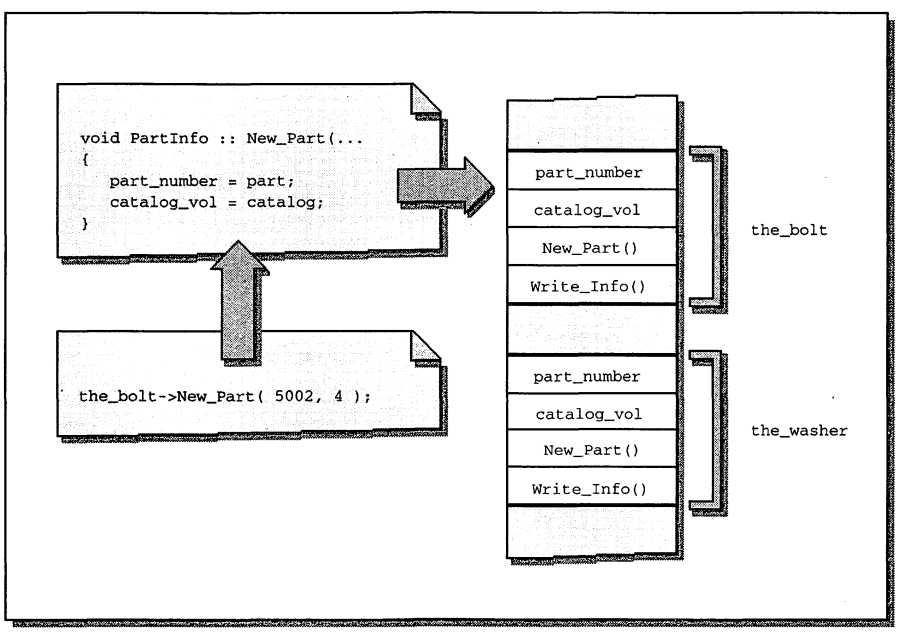


FIGURE 5–11 A member function acts only on the object that invoked it.

The MultipleObjects program that is found on the accompanying disk demonstrates how to create two objects from the same class type. If you understood this chapter's MemberFunctions program, you'll easily be able to follow what's going on in the MultipleObjects program. MultipleObjects is the MemberFunctions program with just a few lines added to it. Study the code; then look at Figure 5–12 to see what the program's output looks like.

```
// ***************  MultipleObjects.cp  ****************

class  PartInfo
{
   private:
      long  part_number;                  // data member
      long  catalog_vol;                  // data member

   public:
      void  New_Part( long, long );       // member function
      void  Write_Info( void );           // member function
};

void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}


void  PartInfo :: Write_Info( void )
{
   Str255  the_str;

   NumToString( part_number, the_str );
   DrawString( the_str );

   Move( 20, 0 );

   NumToString( catalog_vol, the_str );
   DrawString( the_str );
}
```

```
PartInfo   *the_bolt;      // declare an object of type
                           // PartInfo
PartInfo   *the_washer;    // declare an object of type
                           // PartInfo

void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   the_bolt   = new PartInfo;     // allocate object memory
   the_washer = new PartInfo;     // allocate object memory

   the_bolt->New_Part( 5002, 4 ); // invoke member function
   the_washer->New_Part( 37, 3 ); // invoke member function

   MoveTo( 20, 30 );
   the_bolt->Write_Info();        // invoke member function
   MoveTo( 20, 50 );
   the_washer->Write_Info();      // invoke member function

   while ( !Button() )
      ;

   delete the_bolt;               // delete the object
   delete the_washer;             // delete the object
}
```

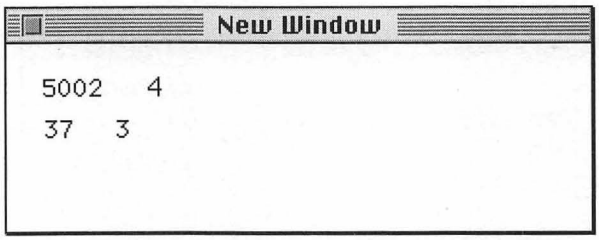


FIGURE 5–12 The output of the MultipleObjects program

Accessing Data Members

In object-oriented programming, an object's data and the functions that act on that data are bound together within an object. This concept of binding data and action routines is called *encapsulation.* Another important idea in OOP is *data hiding*—the prevention of access to an object's data by routines that are not declared in the object's list of member functions.

Data Access via Member Functions

You've already seen that an object's data members can be accessed through the object's member functions. For example, to set the values of the data members of a PartInfo object, you invoke the object's New_Part() member function. To examine, or display, the values of PartInfo data members, you use Write_Info().

An object's member functions can always be used to access the object's data members, as shown in Figure 5–13.

Using the private and public Keywords to Limit Access

You now know that an object's member functions are used to access the object's data members. But can an object's members be accessed without going through a member function? The answer to that is, "yes and no." When declaring a class, you set the level of access control for the data members. To set the access control, you use the C++ keywords *private* and *public.*

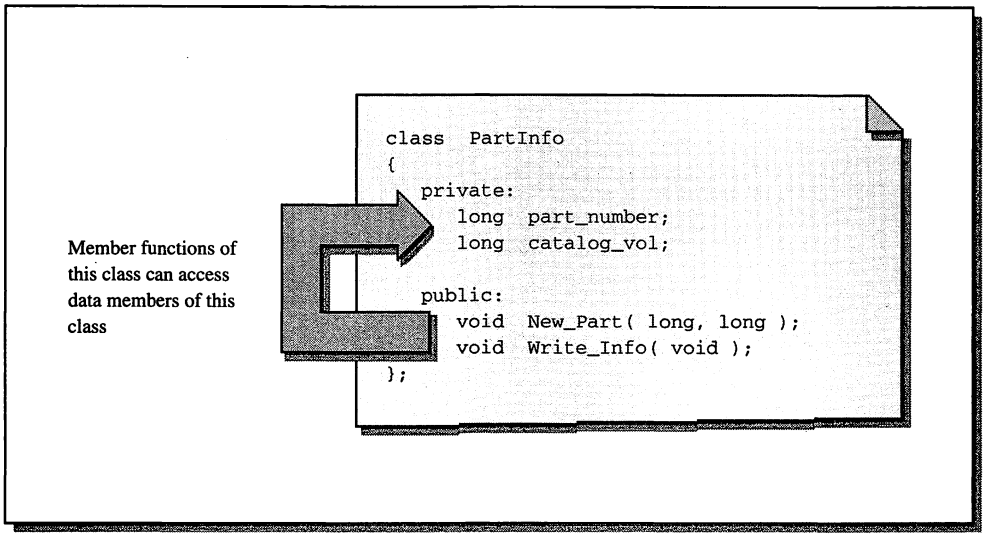


FIGURE 5–13 Data members are accessed via member functions

The private and public keywords let your program know if code that is not part of an object can access code that is part of the object. Parts of an object marked as private cannot be accessed by outside code, while parts that are marked as public can be.

Typically, the data members of an object will be marked private, while the object's member functions will be marked public. Making data private limits access to it. Only something that is private to the class—namely a member function—can access the data. Making the member functions public allows code outside of the object—public code—to call the member functions. This is shown in Figure 5–14.

In Figure 5–14, you see that the private data members of PartInfo are being accessed by Write_Info(). That's all right, because Write_Info() is a member function. The figure also shows that the PartInfo member function Write_Info() is being called from within main()—a function that isn't a part of PartInfo.. This too is all right, because the member functions have been marked as public.

Accessing Data without Using Member Functions

Marking an object's data members as private severely limits your program's access to that data. Only the marked object's member functions can work with that object's data, as Figure 5–15 shows.

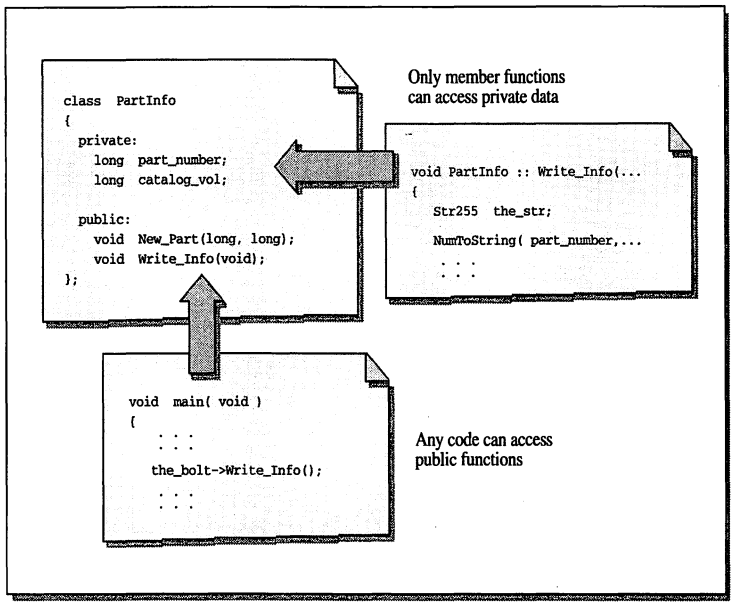


FIGURE 5–14 Private data can be accessed only through member functions, while public functions can be referenced anywhere in a program.

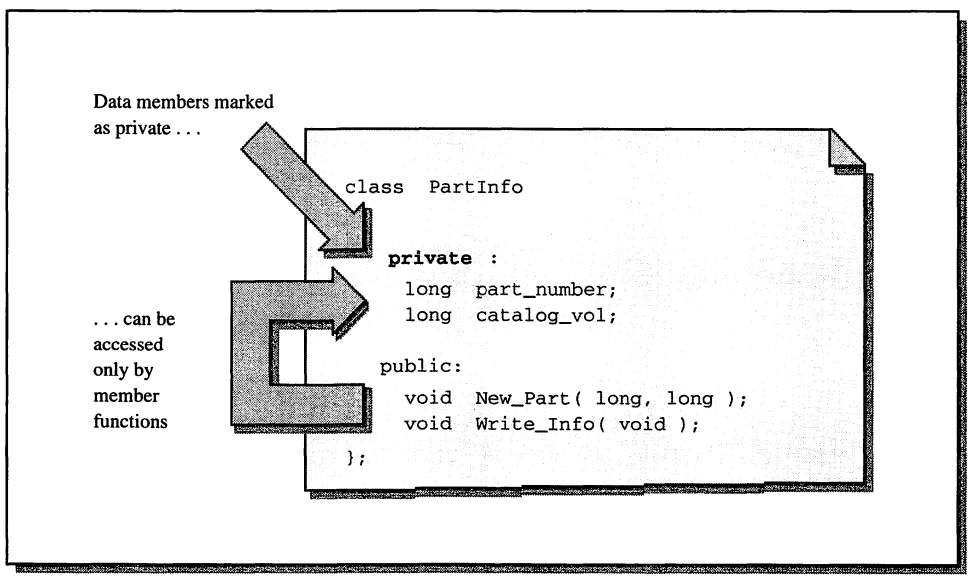


FIGURE 5–15 The private keyword prevents access to data by any means other than a member function.

This access restriction may cause you to wonder if there's a way around the limitations of the private keyword. There is, but I won't recommend its use. I will, however, cover it here so that you get a better understanding of the public and private keywords.

Data members don't have to be private. You can declare any or all of them public. In the following code, `part_number` is public and `catalog_vol` is private.

```
class  PartInfo
{
   public:
      long  part_number;
   private:
      long  catalog_vol;

   public:
      void  New_Part( long, long );
      void  Write_Info( void );
};
```

Now that `part_number` is public, how does its level of access change? You can now work with the data member directly—without first going through a member function. To do that, use the -> operator with the object's name. In the following snippet, I set the value of `part_number` directly rather than through the New_Part() member function.

```
PartInfo   *the_bolt;

the_bolt = new PartInfo;

the_bolt->part_number = 6789;
```

The above example shows that using the public keyword makes it easier to access an object's data members than using the private keyword. In programming, however, *easy* isn't always synonymous with *good.* This is one such case.

Encapsulation—the binding of an object's data with the routines that operate on that data—breaks down when data members are made public. When class data is kept private, a glance at the class member func-

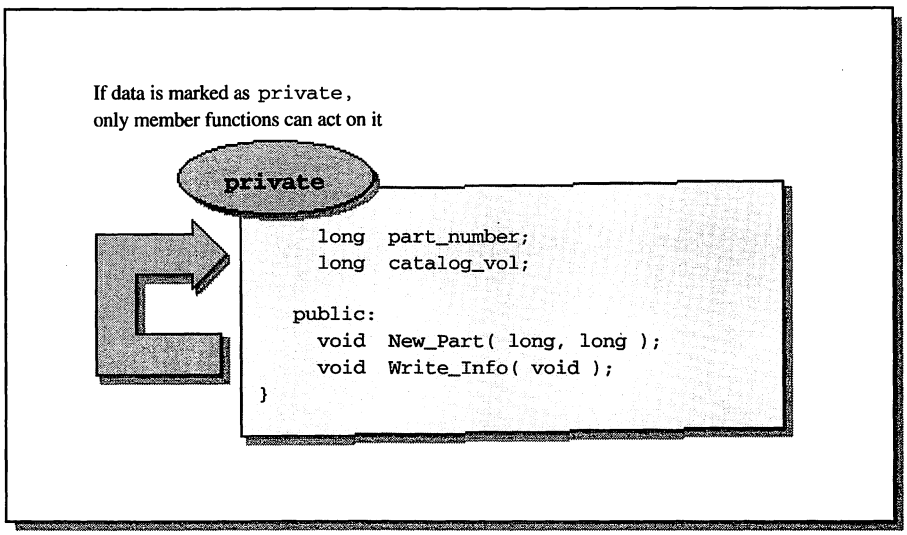


FIGURE 5–16 When the private keyword is applied to class data, encapsulation is maintained.

tions reveals all the possible actions that can be performed on an object of that class type. Because private data cannot be accessed by functions other than member functions, you know that the list of public member functions tells the entire story of what actions can be performed on the data of an object of a class. Figure 5–16 illustrates this idea. Figure 5–17 shows how class data that is declared to be public can be accessed by functions other than class member functions.

Many people who have programmed in C recognize the similarity between stucts and classes, and they instinctively try to work with a class in the same way they worked with a struct. While it is understandable that a person would gravitate toward the familiar, doing so in the case of trying to work with a class as you would a struct defeats the purpose of object-oriented programming. One of the key concepts of object-oriented programming is that objects are acted on only through their public member functions. This makes an object self-contained. Not just any function should have the ability to examine or alter the member variables of an object. Restricted access to object data is one of the key *advantages* of object-oriented programming; don't override this feature by making data members public.

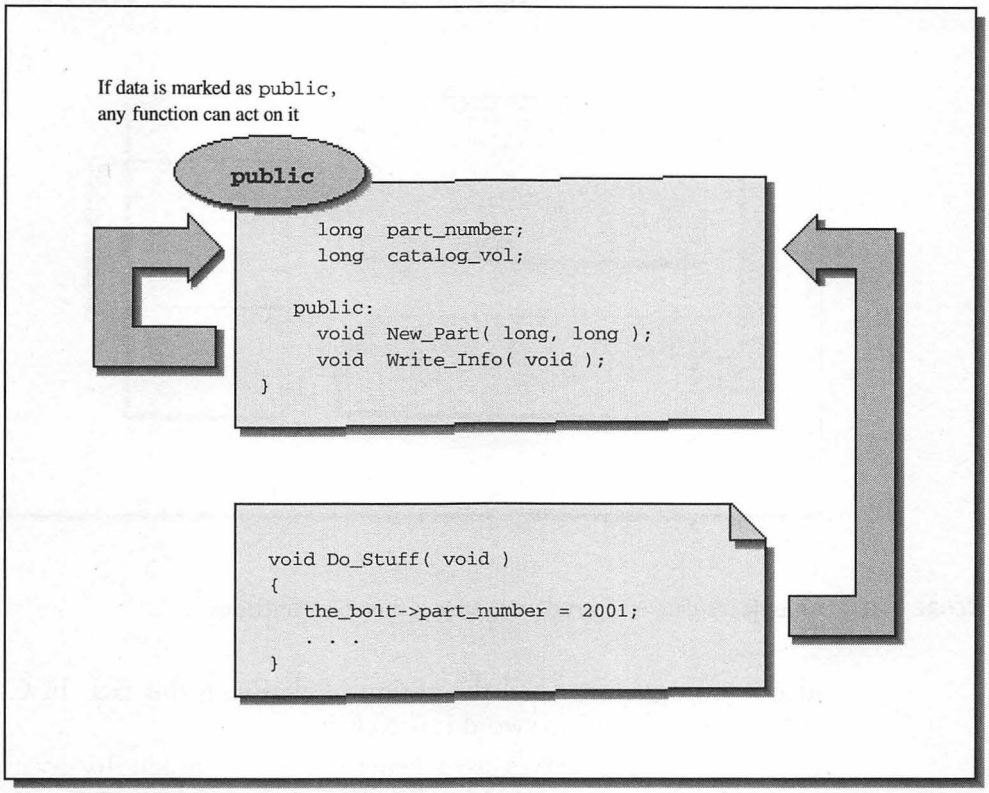


FIGURE 5–17 When the public keyword is applied to class data, encapsulation is lost.

The this Operator

You've seen that the code that makes up a member function acts upon the data of the object that invoked it, as shown in Figure 5–18. In that figure, an object named `the_bolt` is invoking the New_Part() member function, so the values of the data members of the object `the_bolt` are changed in memory.

The C++ compiler always knows that a member function is working with the data of an object. But for humans—especially humans used to programming in a procedural language like C—it would be nice to be reminded of this. While it isn't necessary to specify that an assignment used in a member function is working on an object's data member, it would be

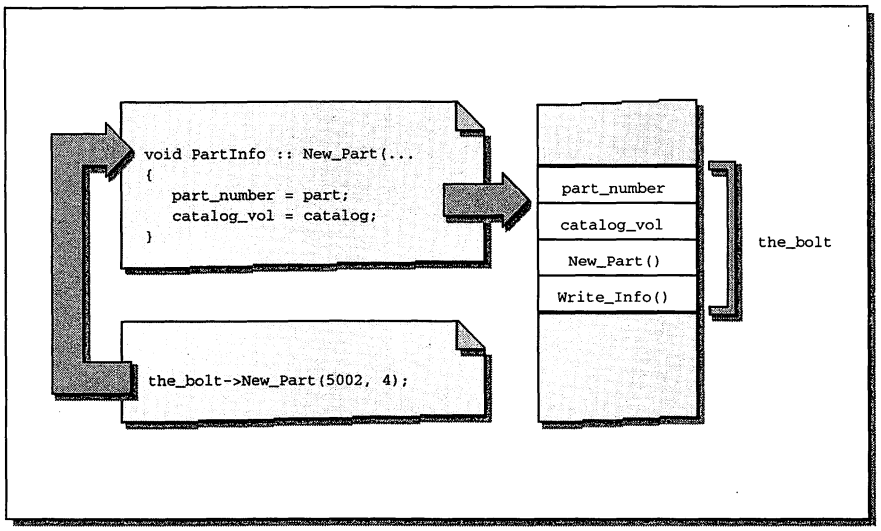


FIGURE 5–18 An object's data is altered using object member functions.

nice if a programmer had the option of clarifying this fact. In C++, that's exactly what the *this* keyword is used for.

In C++, this serves as a generic object pointer. Its use, which is optional, lets you or anyone reading your code quickly see that a statement is working with an object's data member. Here now is the New_Part() member function, written twice—once without the *this* keyword and once with it. Keep in mind that both functions have identical outcomes.

```
void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}

void  PartInfo :: New_Part( long part, long catalog )
{
   this->part_number = part;
   this->catalog_vol = catalog;
}
```

Whether or not I choose to use the *this* keyword in the New_Part() function, the compiler knows to treat `part_number` and `catalog_vol` as

data members of the object that invoked New_Part(). If I call New_Part() through an object named `the_bolt`—as the following lines of code do—the compiler uses the `the_bolt` object in the New_Part() function, as shown in Figure 5–19.

```
PartInfo   *the_bolt;

the_bolt = new PartInfo;

the_bolt->New_Part( 5002, 4 );
```

Using the *this* keyword reminds you that a member function is working with an object's data members. This reminder is especially helpful if your program uses a local or global variable that has a name similar to the name of a data member declared in a class. For instance, what if a program I wrote declared a global variable named `part_number`? In the following code, which `part_number` would be used by New_Part() in the first

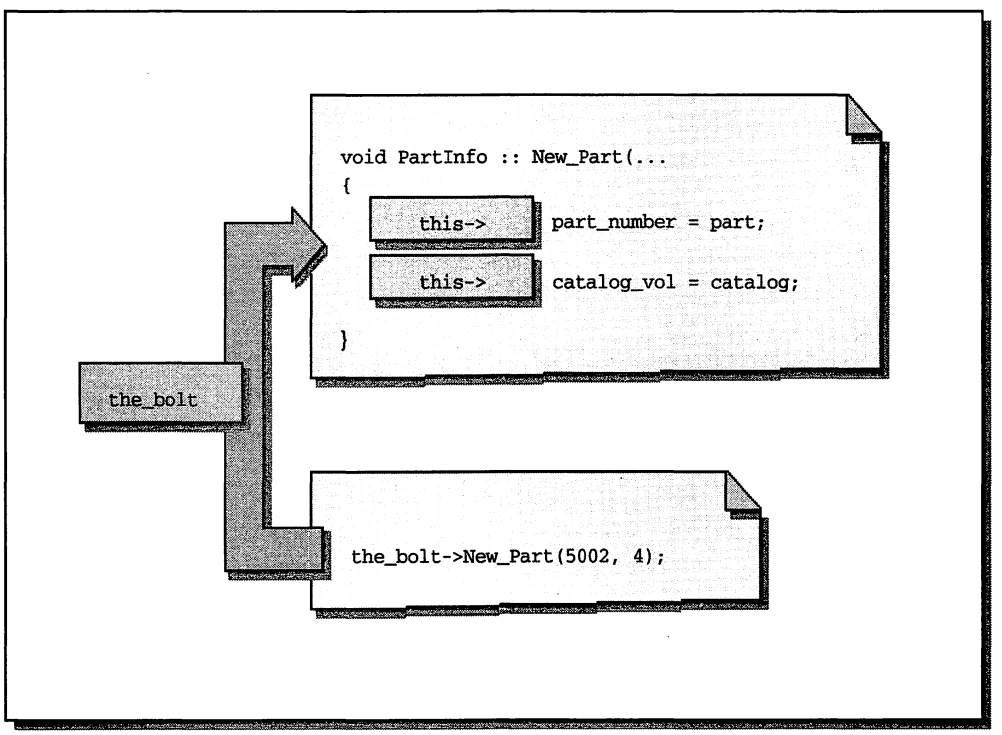


FIGURE 5–19 A member function always acts on the data of the object that invokes the function.

assignment statement of that function—the global variable or the object's data member?

```
class  PartInfo
{
   private:
      long  part_number;         // data member
      long  catalog_vol;

   public:
      void  New_Part( long, long );
      void  Write_Info( void );
};


void  PartInfo :: New_Part( long part, long catalog )
{
   part_number = part;
   catalog_vol = catalog;
}


PartInfo   *the_bolt;
long        part_number;        // global variable

the_bolt = new PartInfo;

the_bolt->New_Part( 5002, 4);
```

The answer is that a member function acts on the data members of the object that invoked the member function. Thus, when faced with two part_number variables, New_Part() will use the object data member rather than the global variable. By rewriting New_Part() to include the *this* keyword, as I've done below, you lessen the potential for confusion.

```
void  PartInfo :: New_Part( long part, long catalog )
{
   this->part_number = part;
   this->catalog_vol = catalog;
}
```

NOTE

Confusion can also be reduced by choosing sensible variable names. In a small program written by one person, it's easy to ensure that variable names aren't similar to data member names. In a large OOP program—one on which several programmers may be working, for example—variable names that are similar to data member names can occur. That's when the use of the *this* keyword can really help to minimize mix-ups.

Since the *this* keyword can keep C++ code clear, I'll be using it in the remainder of the book.

Constructors and Destructors

Supplying a data structure with initial, or default, values is a chore every programmer encounters. And, when he or she has finished with a data structure, cleaning up—which usually involves disposing of allocated memory—is another responsibility of the programmer. Through the use of two special member functions—the *constructor* and the *destructor*—C++ makes these tasks a little easier. And, perhaps more important, it makes it very evident where in your code these tasks take place.

Constructors

Every class you write can contain an optional member function called a *constructor*. When an object is created using the new operator, the constructor function—if declared as part of the class the object is based on—will automatically be called.

The purpose of a constructor function is to initialize data members or allocate additional memory. The format of the definition of a constructor function is always the same—the constructor's name is the class name. And it does not have a return type listed. Here's how the PartInfo class would look if it had a constructor:

```
class  PartInfo
{
   private:
      long  part_number;
      long  catalog_vol;
```

```
   public:
            PartInfo( void );          // constructor
      void  New_Part( long, long );    // member function
      void  Write_Info( void );        // member function
};
```

The header of a member function consists of the function's return type, the class name, the scope resolution operator, the member function's name, and the list of arguments. Here's the header for the New_Part() member function:

```
void  PartInfo :: New_Part( long part, long catalog )
```

The header for a constructor follows the same pattern as that of any other member function—with one exception. No return type is listed. Here's the header for the PartInfo constructor:

```
PartInfo :: PartInfo( void )
```

Now let's take a look at an actual constructor. A PartInfo object has two data members—`part_number` and `catalog_vol`. Although I've made it a practice to assign values to these data members soon after an object is created, it might be a good idea to initialize the values to 0. If for some reason my program fails to call New_Part() to give values to the data members, at least I'll know that the object won't have data member values that duplicate those of some other object. Here's the code for the PartInfo constructor:

```
PartInfo :: PartInfo( void )
{
   this->part_number = 0;
   this->catalog_vol = 0;
}
```

The constructor function is called automatically each time a new object is created. The new operator is responsible for making the call to the constructor; you'll never have to explicitly call the function yourself. Figure 5–20 illustrates this idea.

A constructor doesn't have to contain only assignment statements. It can do anything that a normal C or C++ function can do. I'll demonstrate

that by adding a couple of lines to the PartInfo constructor. The new version of the constructor writes out a message to the active window to let the user know that a new part was indeed created:

```
PartInfo :: PartInfo( void )
{
   this->part_number = 0;
   this->catalog_vol = 0;
   MoveTo( 20, 30 );
   DrawString( "\pItem created." );
}
```

The Constructor program uses the above constructor to initialize the data members of a PartInfo object. Look over the code and the output of the program—shown in Figure 5–21. The program is included on the disk, so you can verify the results yourself.

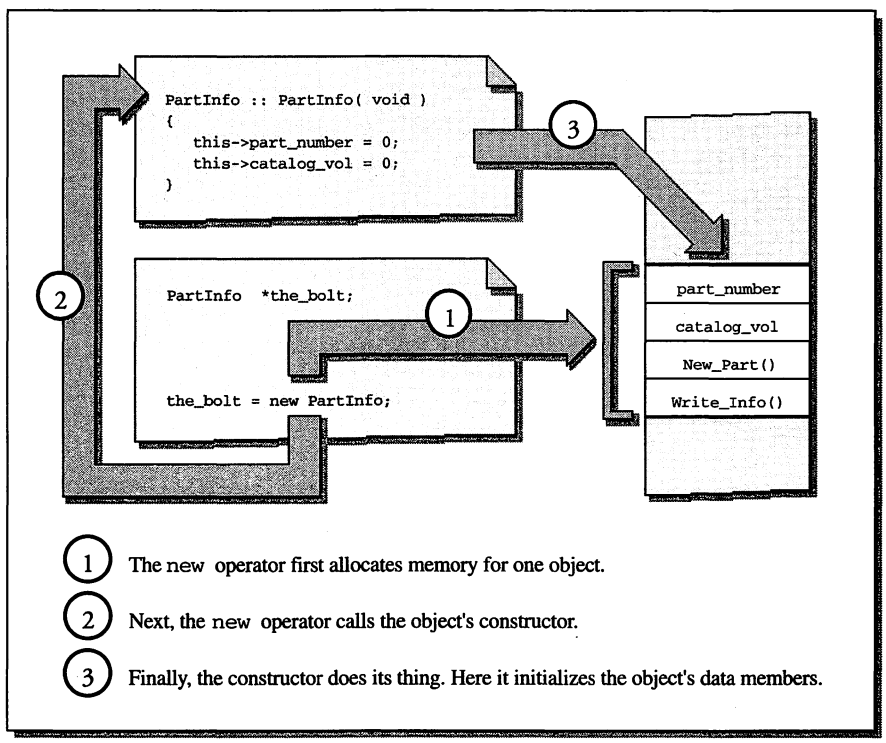


FIGURE 5–20 If an object has a constructor function, the new operator will cause it to execute.

```
// ****************  Constructor.cp  ****************

class  PartInfo
{
   private:
      long  part_number;
      long  catalog_vol;

   public:
            PartInfo( void );          // constructor
      void  New_Part( long, long );    // member function
      void  Write_Info( void );        // member function
};


PartInfo :: PartInfo( void )
{
   this->part_number = 0;
   this->catalog_vol = 0;
   MoveTo( 20, 30 );
   DrawString( "\pItem created." );
}


void  PartInfo :: New_Part( long part, long catalog )
{
   this->part_number = part;
   this->catalog_vol = catalog;
}


void  PartInfo :: Write_Info( void )
{
   Str255  the_str;

   NumToString( this->part_number, the_str );
   MoveTo( 20, 50 );
   DrawString( the_str );
```

```
    NumToString( this->catalog_vol, the_str );
    MoveTo( 20, 70 );
    DrawString( the_str );
}


PartInfo  *the_bolt;


void  main( void )
{
    WindowPtr     the_window;
    Rect          window_rect;

    InitGraf( &thePort );
    InitFonts();
    InitWindows();

    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    the_bolt = new PartInfo;

    the_bolt->New_Part( 5002, 4 );

    the_bolt->Write_Info();

    delete the_bolt;

    while ( !Button() )
        ;
}
```

The fact that the string `Item created` appears in the window should be enough proof that the constructor function was really invoked, because the constructor is the function that writes this text. If you want additional

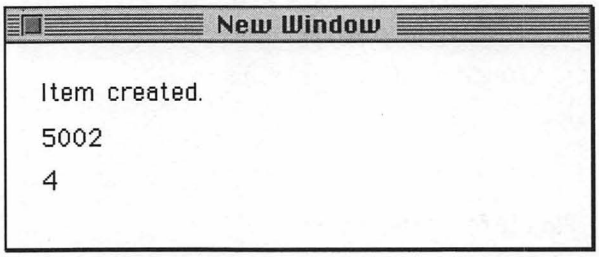


FIGURE 5–21 The output of the Constructor program

proof, try running the program with the Use Debugger option in the Project menu selected. When you select Run from the Project menu, the debugger windows will open. Scroll to the constructor function and click on the diamond that appears to the left of the first line of that function, as shown in Figure 5–22.

Next, click on the Go button in the Source window. The program will run until it hits a breakpoint. You've only set one—in the constructor function—and that's exactly where the program stops. Even though you never wrote a call to the PartInfo constructor, the program enters that routine.

Now, click on the Step button. That moves the arrow down a line in the constructor. Click on the Step button until the black arrow points to the very last line in the constructor. Then, click on the Step button one more time. The arrow moves out of the constructor and back into main(). In particular, it moves to the line of code that creates the PartInfo object, as shown in Figure 5–23. This line, which contains the new operator, is the line that called the PartInfo constructor.

In Figure 5–23, note that the arrow is hollow. This means that the program has just returned from executing a function (the constructor) and there are still several instructions left to execute.

I said that a constructor holds data member initializations and memory allocation code. That last part may seem redundant. Doesn't the new operator allocate memory for an object when the object is created? Yes. So you have to allocate additional memory for an object only in certain instances. I'll discuss such an instance in the next section, and you'll see an example later in this book when I create a class that represents a window.

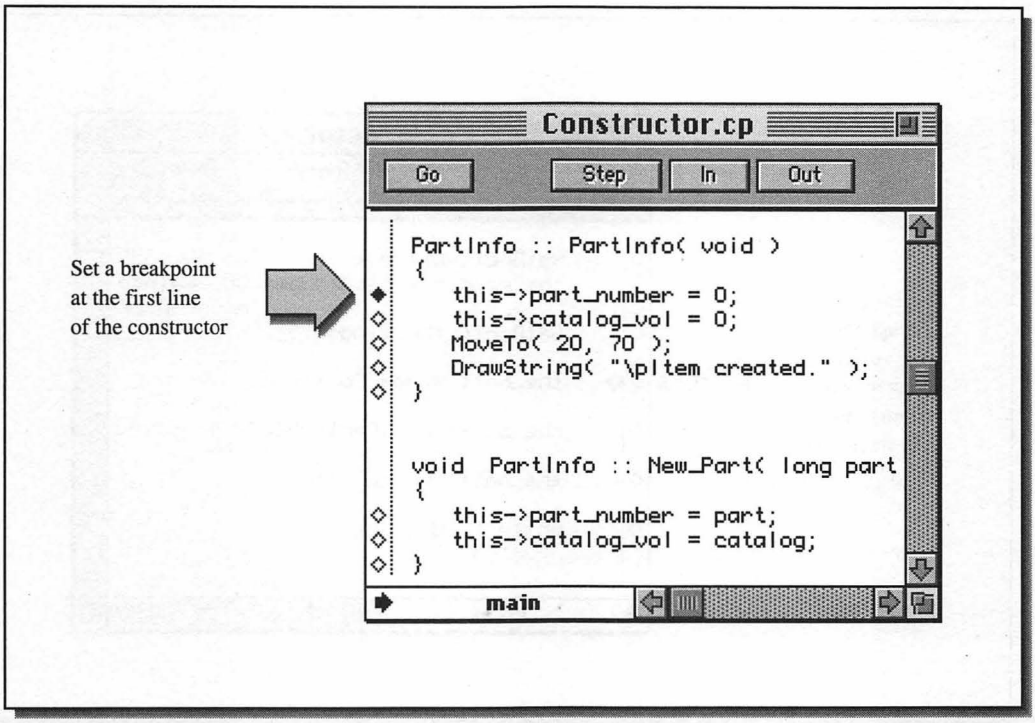


FIGURE 5–22 Setting a breakpoint in the debugger

Destructors

Knowing that the new operator calls a special constructor function has probably made you wonder whether the delete operator also calls a function. It does—the destructor. Like the constructor, the class destructor is optional. If you list one as a member function of a class, any object of that class will call the destructor when it is deleted.

The purpose of the destructor function is usually to free any additional memory that may have been allocated by an object. The format of the definition of a destructor is class name preceded by a tilde. Like the constructor, it does not have a return type listed. Here's the PartInfo class with both a constructor and destructor:

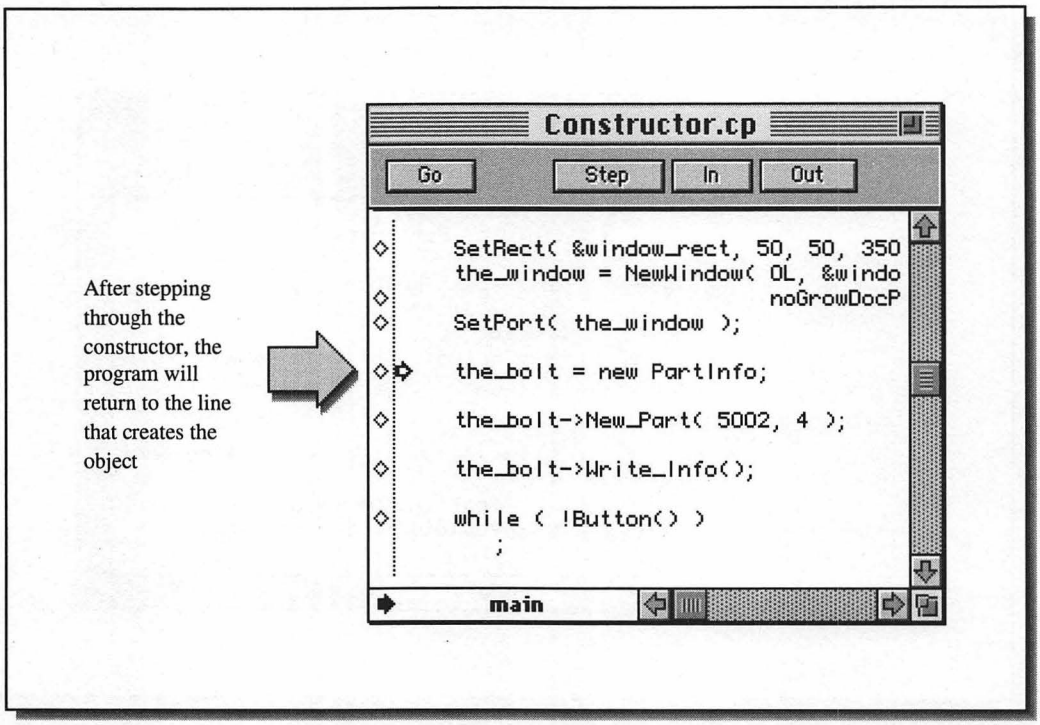


FIGURE 5–23 Stepping through part of the program and back to the object allocation line

```
class  PartInfo
{
   private:
      long  part_number;
      long  catalog_vol;

   public:
            PartInfo( void );          // constructor
           ~PartInfo( void );          // destructor
      void  New_Part( long, long );    // member function
      void  Write_Info( void );        // member function
};
```

The header for a destructor looks like that of the constructor—with the addition of the tilde. This is how the header for the PartInfo destructor would look:

```
PartInfo :: ~PartInfo( void )
```

The following code shows what a very simple destructor might look like. This one simply writes a message to the active window to let the user know a part was deleted from the catalog:

```
PartInfo :: ~PartInfo( void )
{
   MoveTo( 20, 90 );
   DrawString( "\pItem deleted." );
}
```

The primary purpose of a destructor is to deallocate memory, and my PartInfo class doesn't allocate any special memory. True, an object created of this class type will occupy memory. But memory for such things as the long data members `part_number` and `catalog_vol` will be deallocated by the delete operator—without the help of a destructor. After all, the purpose of delete is to return to the free pool of memory the memory that an object occupied.

Figure 5–24 shows an example of a situation in which a destructor might be necessary. On the left of the figure is the PartInfo class along with one PartInfo object in memory. When delete is used to dispose of the object, the delete operator returns all of the memory to the program. On the right of the figure is a different class—WindClass. Like PartInfo, this class has two data members and two member functions. Unlike PartInfo, one of the data members is a pointer—a window pointer.

When an object of the WindClass is disposed of, the delete operator returns only the memory that was occupied by the two data members and the two member function pointers. The delete operator does not deallocate the memory that the wind data member pointed to—the WindowRecord. All WindowPtrs point to WindowRecords. And unless you specifically release the memory occupied by the WindowRecord, it will be unusable by your program. The WindClass would be an ideal candidate for a destructor function—one that released the WindowRecord memory. Later in this book, you'll see an example that does just that.

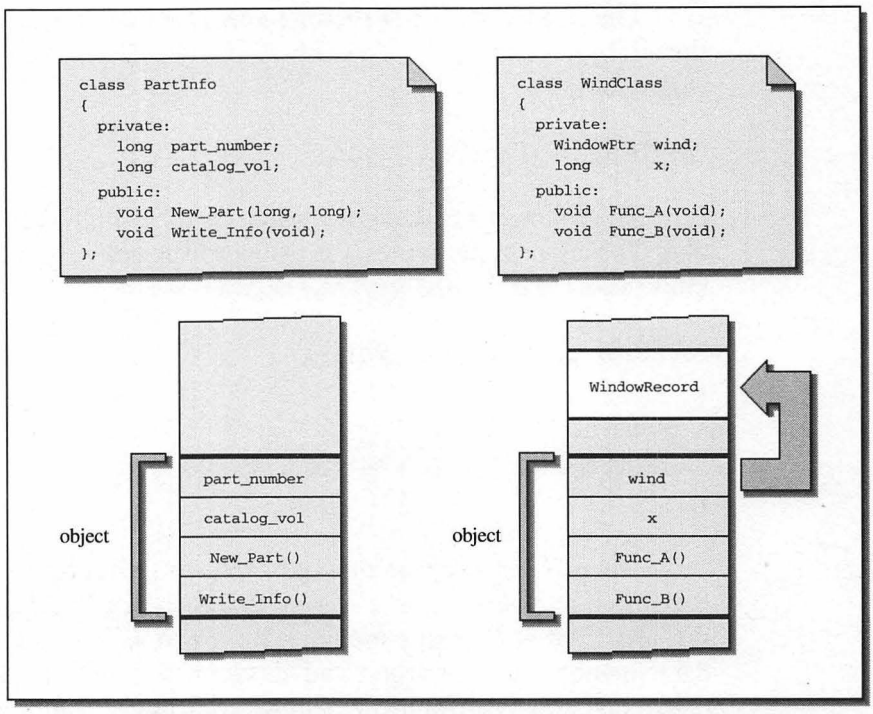


FIGURE 5–24 Objects that use pointers as data members allocate additional memory.

NOTE

Paying attention to memory that is allocated is important. Each program has a finite amount of memory that it is allowed to work with—no matter how much RAM the user of your program might have. Not releasing used memory can eventually lead to your program running out of memory.

IMPORTANT

If special steps must be taken to properly free up the memory occupied by something a data member points to—such as the WindowRecord that a WindowPtr points to—then shouldn't similar steps be taken for the member functions? After all, they're simply pointers to other code—the code that makes up the functions themselves. No, because those functions stay in memory, to be used by all objects of that class.

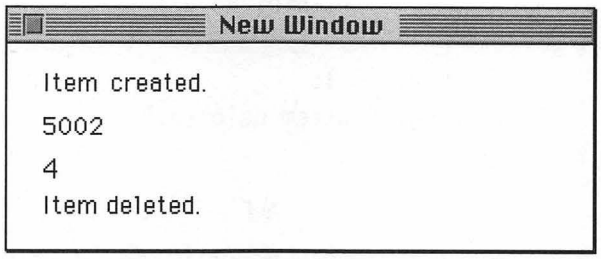


FIGURE 5–25 The output of the Destructor program

The Destructor program—the code for which you'll find below and on the accompanying disk—adds both a constructor and a destructor to the PartInfo class. You've already seen all the code that makes up this program; Destructor just ties it all together. When you run the program, you'll see a window like the one pictured in Figure 5–25.

```
// ******************  Destructor.cp  ******************

class  PartInfo
{
   private:
      long  part_number;
      long  catalog_vol;

   public:
            PartInfo( void );          // constructor
           ~PartInfo( void );          // destructor
      void  New_Part( long, long );    // member function
      void  Write_Info( void );        // member function
};


PartInfo :: PartInfo( void )
{
   this->part_number = 0;
   this->catalog_vol = 0;
   MoveTo( 20, 20 );
   DrawString( "\pItem created." );
}
```

```
  PartInfo :: ~PartInfo( void )
{
   MoveTo( 20, 90 );
   DrawString( "\pItem deleted." );
}


void  PartInfo :: New_Part( long part, long catalog )

{
   this->part_number = part;
   this->catalog_vol = catalog;
}


void  PartInfo :: Write_Info( void )
{
   Str255  the_str;

   NumToString( this->part_number, the_str );
   MoveTo( 20, 50 );
   DrawString( the_str );

   NumToString( this->catalog_vol, the_str );
   MoveTo( 20, 70 );
   DrawString( the_str );
}


PartInfo  *the_bolt;


void  main( void )
{
   WindowPtr     the_window;
   Rect          window_rect;

   InitGraf( &thePort );
   InitFonts();
```

```
    InitWindows();

    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    the_bolt = new PartInfo;

    the_bolt->New_Part( 5002, 4 );

    the_bolt->Write_Info();

    delete the_bolt;

    while ( !Button() )
        ;
}
```

NOTE

Note that I delete the PartInfo object before the while statement near the end of the code rather than after the while as I've done in the past. Normally, I wait until the user clicks the button and ends the program before deleting the program's object. If I did that here, however, you wouldn't see the output of the destructor—the line of text that says `Item deleted` That's because the destructor is called only after the delete operator is used. If I waited until the program ended before calling delete, the window would be closing as the destructor wrote its text to it.

It's been a long chapter—the longest of the book. But the knowledge you've gained about classes and objects will be the base from which you'll create all your object-oriented programming.

Chapter Summary

As a standard C or C++ data type defines the nature of a variable, a class defines the nature of an object. Together, classes and objects *are* object-oriented programming.

To define a class you begin with the class keyword, followed by the name of the class. The contents of the class lie between braces. The class contents consist of data members and member functions. Data members, obviously enough, hold data. Use the private keyword before listing data members to ensure that the data is accessible only to objects of the class type. Member functions are the functions that act on, or work with, the class data members. Use the public keyword before listing member functions so that member functions can be called from anywhere in your source code. Data members can be likened to the members of a struct data structure. Member functions have no C language analog.

The class definition lists the names of the functions that are the class member functions, but the definition doesn't actually define the body of those functions. They are defined outside of the class. The body of a member function might look like that of a normal C function. The difference between a C++ member function and a C function isn't in the content of the function but, rather, in the fact that a C++ member function is always associated with a particular set of data (the class data members), while a C function works with any data.

Once a class is defined, it is an easy matter to create objects based on that class. Usually a pointer to an object is declared by listing first the class name and then the pointer name preceded by the * operator. The declaration doesn't allocate memory for an object or create a new object. To do that you'll use the new operator. Once an object is created, its data members can be accessed by using the -> operator. The short code snippet below demonstrates how to work with an object. First, a pointer to an object of a class named PartInfo is declared. Then memory is allocated for the object using the new operator, and data members of the object are accessed through a member function named New_Part(). Finally, the object is deleted using the delete operator:

```
PartInfo  *the_bolt;

the_bolt = new  PartInfo;
the_bolt->New_Part( 5002, 4 );
delete the_bolt;
```

Two special class-related functions are the constructor function and the destructor function. The class constructor is invoked automatically when an object is created using the new operator. The class destructor function is invoked automatically when an object is deleted using the delete operator.



Chapter 6

Derived Classes

In Chapter 5, you saw that the class is a powerful programming feature. But there's more to come. Classes aren't created indiscriminately in a C++ program. Instead, a program usually has several classes that are related to one another. Rather than forcing you to create each related class from scratch, C++ allows you to name one class as a base from which other classes are derived. These derived classes automatically inherit the data and actions of the class on which they are based.

Derived classes and inheritance are powerful object-oriented features that you'll use in every C++ program you write. And in this chapter you'll discover exactly how to create derived classes.

Multiple Classes

In Chapter 5, you saw that a C++ program can have more than one object of a single class. That, in fact, is one of the advantages of the class type.

```
PartInfo  *the_bolt;        // first pointer to a
                            // PartInfo object
PartInfo  *the_washer;      // second pointer to a
                            // PartInfo object

the_bolt   = new PartInfo;  // allocate first object in
                            // memory
the_washer = new PartInfo;  // allocate second object in
                            // memory
```

Nor does a C++ program have restrictions on the number of *different* classes you can use. Although the examples to this point have consisted of a single class, you're free to create more than that.

The MultipleClasses program is a very simple example of a program that defines two classes—the PersonClass and the VehicleClass. Each class consists of nothing more than a constructor member function. When an object of either class is created, a single line is written to a window. Figure 6–1 shows the output of MultipleClasses.

```
// ***************  MultipleClasses.cp  ***************

class  PersonClass
{
   public:
      PersonClass( void );
};


class  VehicleClass
{
   public:
      VehicleClass( void );
};


PersonClass :: PersonClass( void )
{
   MoveTo( 20, 30 );
   DrawString( "\pPerson object created" );
}
```

```
VehicleClass :: VehicleClass( void )
{
   MoveTo( 20, 50 );
   DrawString( "\pVehicle object created" );
}


PersonClass   *the_man;       // declare object of type
                              // PersonClass
VehicleClass  *the_car;       // declare object of type
                              // VehicleClass


void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   the_man = new PersonClass;   // allocate memory
                                // PersonClass object
   the_car = new VehicleClass;  // allocate memory
                                // VehicleClass object

   while ( !Button() )
      ;
}
```

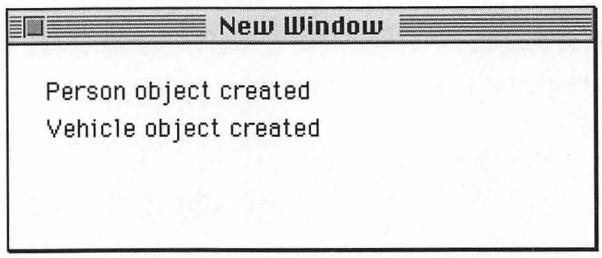


FIGURE 6–1 The output of the MultipleClasses program

Derived Classes

When two similar classes are needed, you may be tempted to jump right in and write them—perhaps copying and pasting information from one to form another. With derived classes, you don't have to do that. Instead, in just a few lines of code you can create a second class that has all the features of a first class—and more.

Why Create Derived Classes?

A pet store owner wants to write a program to keep track of the different types of animals in his shop. The owner just happens to be a Mac enthusiast who programs in C but doesn't know C++. He starts with a very simple data structure that consists of a string that will hold the type of one animal—dog, mouse, and so forth. Here's that structure:

```
struct  Animal
{
   Str255  type;
}
```

He then writes a few functions that allow him to access the Animal data, including a function to add a new animal and a function to write out information about an animal. After doing this, he realizes that it would be a good idea to be able to store and write out supplemental information about certain animals. For instance, he wants to mark certain animals as less suitable pets than others. Because he's programming in C, he modifies his data structure to look like this:

```
#define   NO_XTRA_INFO    0
#define   BAD_PET_INFO    1

typedef  struct
{
   Str255  type;
   short   misc_info;
} Animal, *AnimalPtr;
```

He then declares two struct pointer variables, as shown here:

```
AnimalPtr  the_dog;
AnimalPtr  the_snake;
```

After allocating memory, he can add the extra information:

```
the_dog->misc_info   = NO_XTRA_INFO;
the_snake->misc_info = BAD_PET_INFO;
```

When it comes time to write information to a window, he'll use a function that receives a pointer to a pet. Within that function he'll have to include code that looks something like this:

```
if ( the_pet->misc_info == BAD_PET_INFO )
{
   MoveTo( 20, 50 );
   DrawString( "\pDangerous pet!" );
}
```

While this method will work, there are a couple of drawbacks to it. First, the pet shop owner may think of new classifications of pets over time. When he does, he'll have to modify his program in several places. He'll have to add new #define directives and then search for the code that makes use of them. Second, the complexity of his program increases as he adds decision-making code:

```
if ( the_pet->misc_info == BAD_PET_INFO )
  // write message
else if ( the_pet->misc_info == EXOTIC_PET_INFO )
  // write message
else if ( the_pet->misc_info == EXPENSIVE_PET_INFO )
  // write message
```

```
else if ( the_pet->misc_info == INEXPENSIVE_PET_INFO )
  // write message
. . .
. . .
```

If the program eventually has a dozen functions that access the pet information, there will be at least a dozen such cascaded else-if sections of code. This is exactly the type of situation that object-oriented programming methods seek to avoid. And C++ avoids it through the use of *derived classes.* A derived class can also be called a *subclass.*

Figure 6–2 shows two classes—one a *base* class, the other a derived class. Any C++ class can serve as a base class. From the base class is derived a second class. Not surprisingly, that class is called a derived class. You can't distinguish a base class from a derived class by looking at their contents, because both consist of data members and member functions, as shown in Figure 6–2.

Figure 6–2 is simply a block diagram of two classes; it doesn't show off the special properties of a derived class, namely, that a derived class *inherits* the data members and member functions of the class on which it is based. The derived class then adds its own data members and functions to those inherited from the base class, as shown in Figure 6–3.

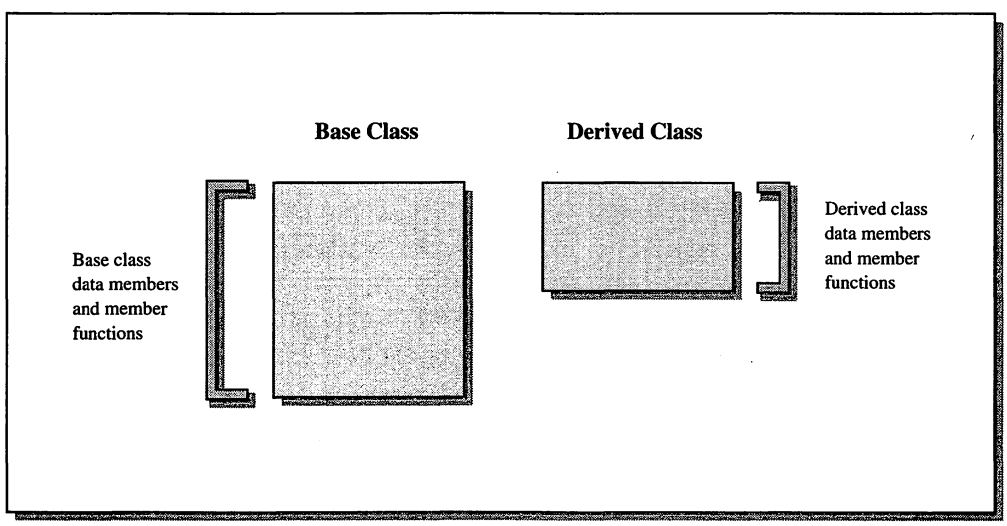


FIGURE 6–2 Base classes and derived classes contain data members, member functions, or both.

How would the pet shop owner write his program if he knew object-oriented programming? He could begin by creating a single base class called Animals. In that class would be the data members and member functions that would be common to *all* the pets he had in his shop. For instance, each animal would have a type, like dog or cat, so the base class would have a Str255 data member to hold this information. The base class would also have member functions to enter a new animal and write out information about an animal—actions common to all types of animals.

What about the derived class? In the case of the less suitable pet—the snake, for example—the derived class could consist of simply a single member function that would write out a warning message. Because the derived class inherits all of the data members and member functions of the base class, the derived class would still be able to store the type of animal and use the member functions that add the animal and write the animal information. Figure 6–4 shows how this base and derived class might look like.

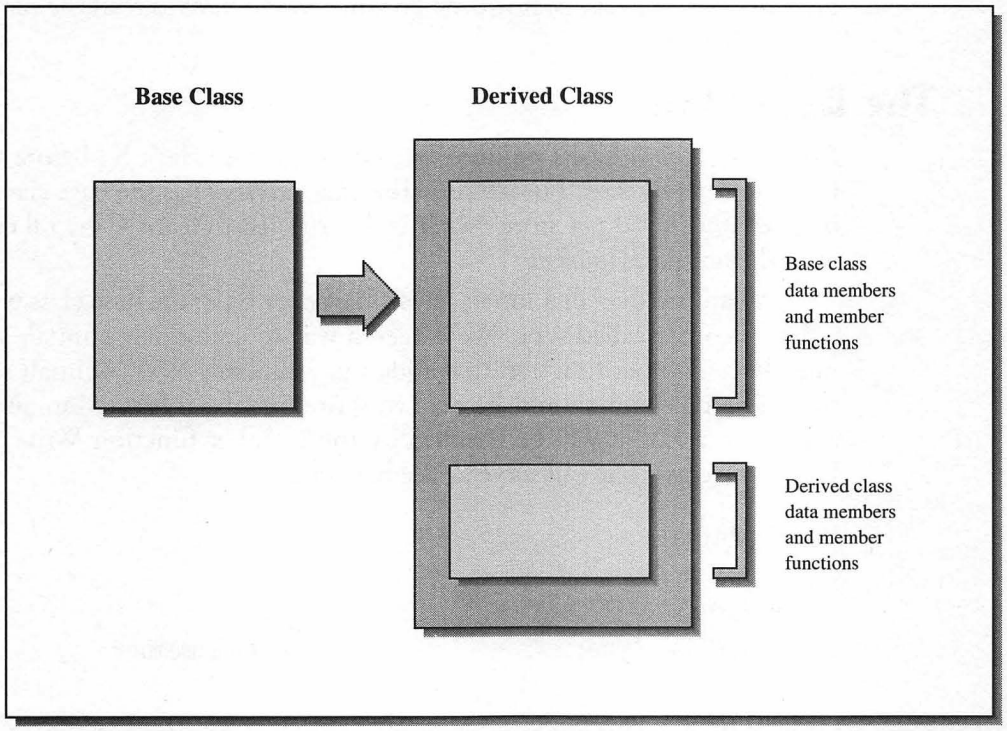


FIGURE 6–3 A derived class inherits the contents of the class on which it is based—the base class.

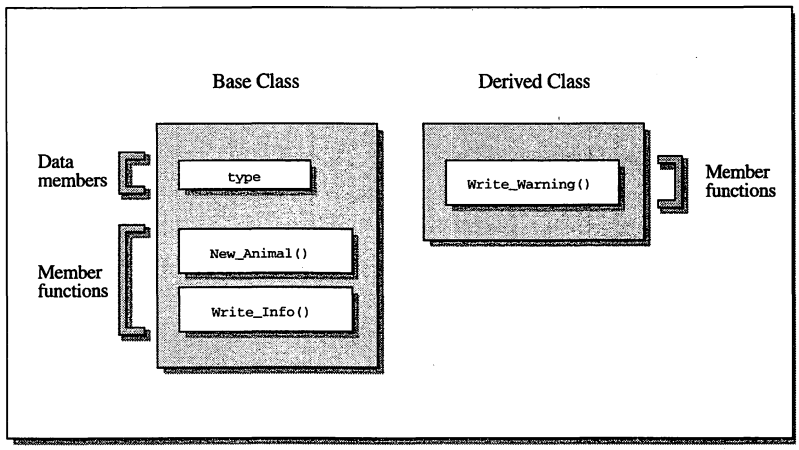


FIGURE 6–4 An example of a base class and a derived class

Figure 6–4 doesn't provide all the detail you need to write your own base and derived classes, however. For that, you'll have to read the remainder of this chapter.

The Base Class

A derived class is based on another class—the base class. So before delving into the derived class, I'll define a class that can serve as the base class. Since our neighborhood pet store owner is determined to learn C++, I'll carry on with the animal database.

Assuming that all animals have a name, or type, the base class will have a data member called type. We'll need a way to add a new animal, so there must be a member function that adds a new animal. New_Animal() will do that. The program should be able to write out the information about the animal, a task that will be handled by the member function Write_Info(). Here's the class that will serve as the base class:

```
class  Animal
{
   private:
      Str255  type;                    // data member

   public:
      void  New_Animal( Str255 );      // member function
      void  Write_Info( void );        // member function
};
```

Creating a new animal will consist of creating a new object and then setting the name, or type, of that animal using the New_Animal() function. That function will simply accept a string that serves as the animal type and then call my own Fill_Str255 function to place that string in the type data member. You can flip back to Chapter 3 if you don't remember what the Fill_Str255() function looks like.

```
void  Animal :: New_Animal( Str255 name )
{
   Fill_Str255( this->type, name );
}
```

Because the Animal class contains only one data member, writing the information requires only moving to the appropriate window location and calling DrawString() once. Here's a look at the Write_Info() member function:

```
void  Animal :: Write_Info( void )
{
   DrawString( this->type );
}
```

The way the Animal class was declared and the things that the class is composed of—data members and member functions—shouldn't look much different from the classes you encountered in the previous chapter. So what makes a base class different from any other class? Absolutely nothing. Any class can serve as a base class. The difference lies in the derived class—as you are about to see.

The Derived Class

To create a class that is derived from another class, you include the name of the base class on the first line of the derived class definition. But it's not enough to just give the class a name and then specify the class on which it will be based. You must also provide an *access specifier.* In almost all circumstances you'll want to use the public keyword as the access specifier. Here's an example:

```
class  BadPet : public  Animal
```

The colon between the class name and the access specifier lets the compiler know that this is to be a derived class. Figure 6–5 shows the syntax for the first line of a derived class declaration.

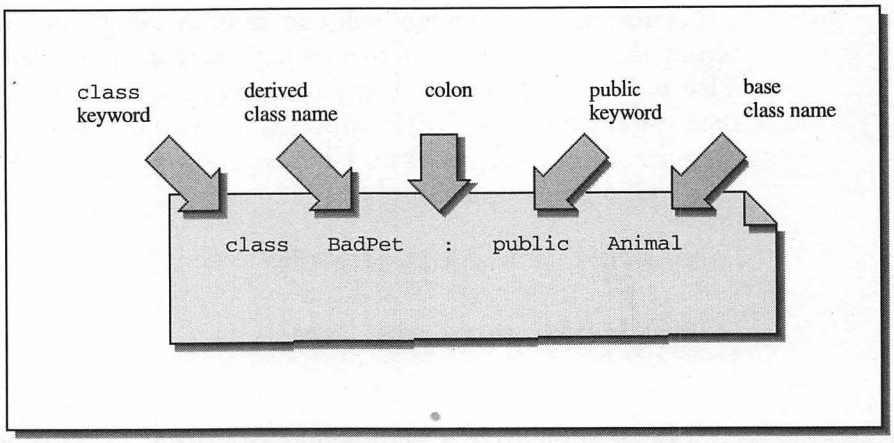


FIGURE 6–5 The format of the definition of a derived class

IMPORTANT

The public access specifier tells the compiler to include everything from the base class as is. That means that all data members and member functions inherited from the base class will retain the public and private keywords that are associated with them.

You can also use the private and protected keywords as access specifiers. They are used so infrequently, however, that I won't discuss them here. You will, however, see both of these keywords later in this chapter when I discuss the use of access specificiation *within* a class definition.

Since the derived class will include all the data members and member functions of class on which it is based, you need only define data and functions that are unique to the derived class. If the purpose of my derived class, which I've named BadPet, is to hold information about less desirable pets, then the only thing that BadPet might need is a member function that writes out a warning message. Here's how my derived class looks:

```
class  BadPet : public  Animal
{
   public:
      void Write_Warning( void );       // member function
};
```

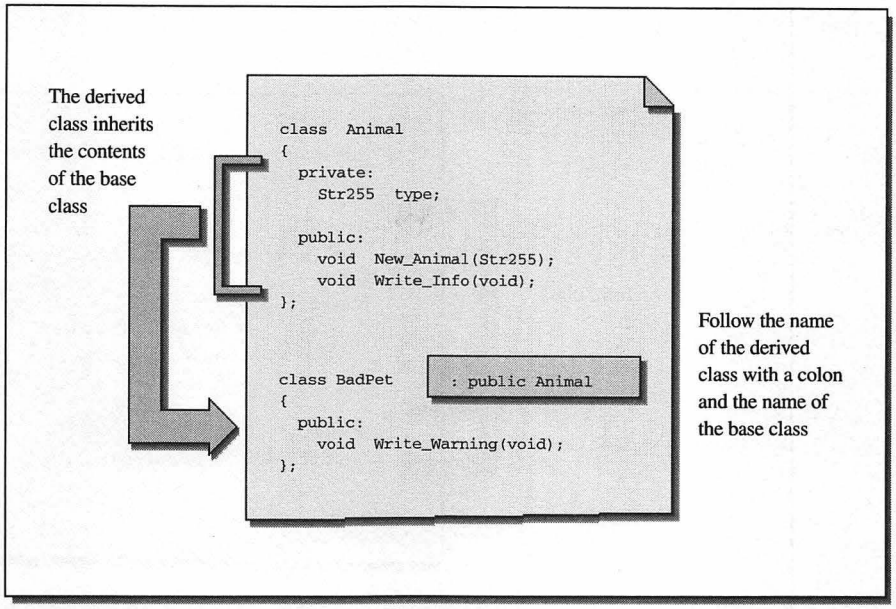


FIGURE 6–6 The BadPet derived class inherits the contents of the Animal base class.

Figure 6–6 shows that the BadPet class will inherit the data and functions of the Animal class. Figure 6–7 takes things a step further by illustrating how one could imagine that the BadPet derived class is the combination of both the base class and the derived class itself.

You write a member function of a derived class just as you would write a member function for any other class. List the return type, the class name, two colons, and the function name. Include any argument the function takes. Here's how I wrote the Write_Warning() member functions:

```
void  BadPet :: Write_Warning( void )
{
   DrawString( "\pDangerous pet!" );
}
```

Working with Derived Class Objects

Now that you can define base and derived classes, you're ready to create objects of both types. And you're ready to implement those objects in a C++ program that is truly object-oriented.

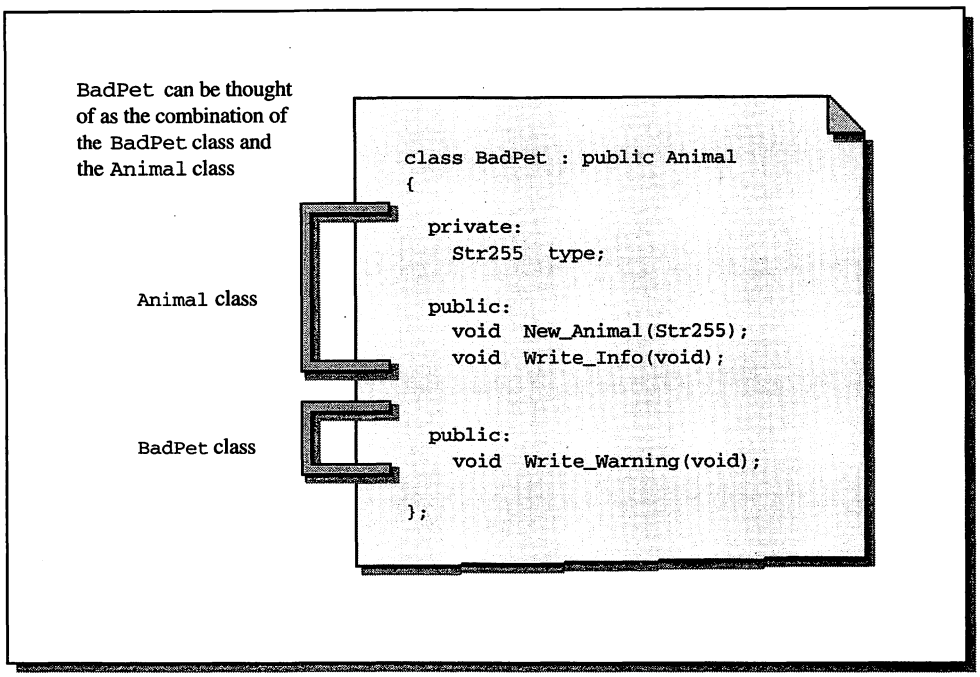


FIGURE 6–7 The BadPet derived class can be described as a combination of itself and the base class.

Creating Derived Objects

You declare an object to be of a derived class just as you would declare an object of any other class. Here's the declaration of an Animal object and a BadPet object:

```
Animal  *the_dog;          // declare object of type
                           // Animal
BadPet  *the_snake;        // declare object of type
                           // BadPet
```

Before working with an object, you must allocate memory for it. Use the new operator as I have done here to create an Animal object and a BadPet object.

```
the_dog   = new Animal;
the_snake = new BadPet;
```

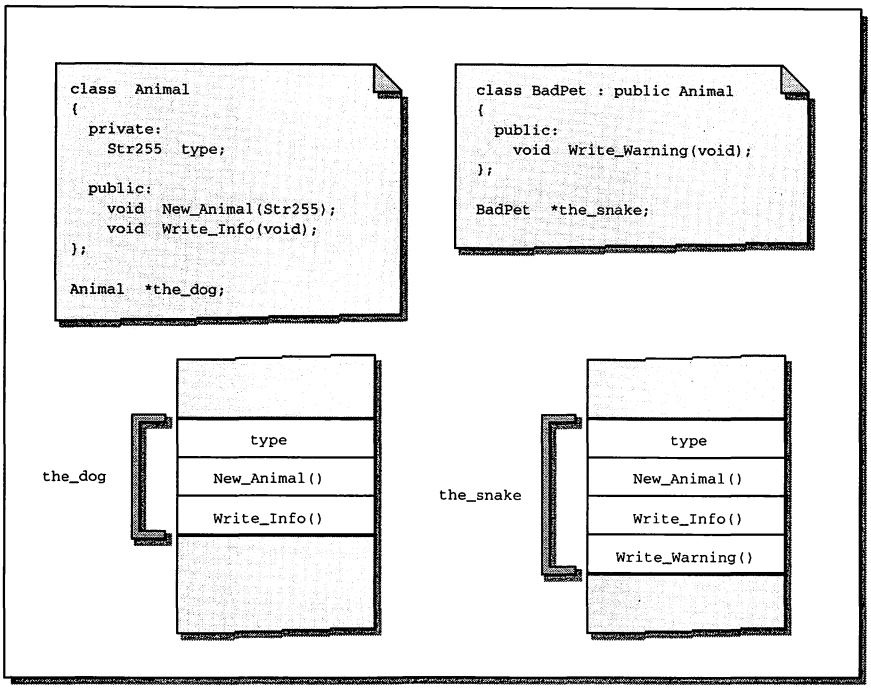


FIGURE 6–8 An Animal object and a BadPet object in memory

Figure 6–8 shows what memory looks like after the above objects have been created. As you examine the figure, keep in mind that each of the objects has its own *type* data member. Thus the `the_dog` object can call New_Animal() to set its type data to Labrador dog while the `the_snake` object can call New_Animal() to set its own type data to Python snake.

Derived objects always have their own versions of inherited data members. Inherited member functions are a little different. That's because member functions are pointers to the functions themselves. So the derived class inherits the list of member functions. Both the base class and the derived class point to the same functions. Figure 6–9 shows how the list of member functions for an Animal object and the list for a BadPet object both point to the same functions in memory.

Using Derived Objects

An object of a derived class inherits the data and functions of its base class. So an object of a derived class can call all the member functions listed in the base class. Even though the BadPet class doesn't explicitly define the func-

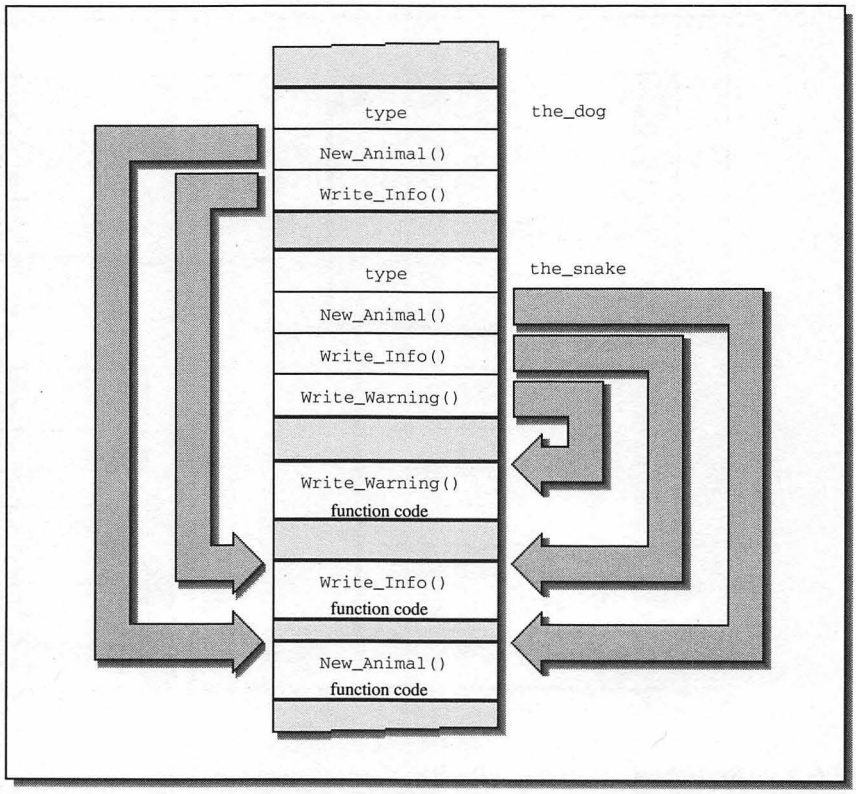


FIGURE 6–9 Because a derived class inherits the member functions of a base class, objects of both types have pointers to the same functions.

tions New_Animal() and Write_Info(), objects of this derived class can call these functions:

```
the_snake->New_Animal( "\pPython snake" );  // call base
                                            // member function
the_snake->Write_Info();                    // call base
                                            // member function
```

Because these two functions are defined in the Animal class, objects that are of the Animal type can, of course, call these functions as well:

```
the_dog->New_Animal( "\pLabrador dog" );    // call base
                                            // member function
```

```
the_dog->Write_Info();                    // call base

                                          // member function
```

What about the Write_Warning() function that was listed in the BadPet class? That function can be called only by a BadPet object:

```
the_snake ->Write_Warning();     // call derived member
                                 // function
```

An Animal object can't call Write_Warning(), because it has no knowledge of, or access to, data members or member functions that are created in derived classes.

Derived Objects and Data Member Access

The BadPet class has a single member function, the code for which I've repeated below.

```
void  BadPet :: Write_Warning( void )
{
   DrawString( "\pDangerous pet!" );
}
```

The Write_Warning() function doesn't attempt to access type, the data member that BadPet inherits from the Animal class. Could it? The answer is "no." For an explanation of why that's so, you'll have to look at the definition of the base class—the Animal class. Notice that the private keyword is used to define the level of access to the type data member:

```
class  Animal                             // base class
{
   private:
      Str255  type;                       // data member

   public:
      void  New_Animal( Str255 );         // member function
      void  Write_Info( void );           // member function
};
```

Recall from Chapter 5 that the only functions that can access a private data member of a class are the member functions of that same class. In the case of the type data member of the Animal class, only the New_Animal()

and Write_Info() member functions of the Animal class can be used to access type. While classes that are derived from the Animal base class (such as BadPet) inherit the type data member, these derived classes can access the type data member only via the member functions derived from the Animal base class, as shown in Figure 6–10.

Can this access limitation be overcome? Yes. There are two ways to go about it, but one method is much better than the other. Let's take a look at the less preferable way first, because it's the more obvious method.

The Animal class declares the type data member to be private. By declaring the type data member to be public, access to it is increased. If I do that, when BadPet inherits type it will inherit it as a public data member that can be accessed directly by any BadPet object; there will be no need to use an Animal member function—or any function at all, for that matter.

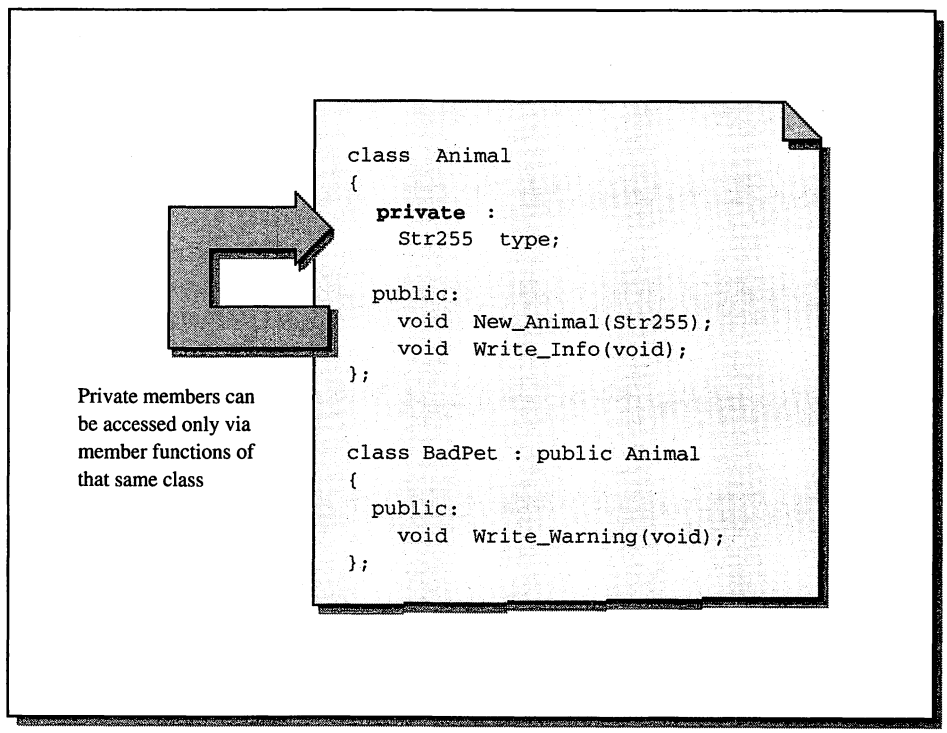


FIGURE 6–10 Only class member functions can access private class data members.

Assuming that type has been declared public, the following definition of Write_Warning() will work:

```
void  BadPet :: Write_Warning( void )
{
   DrawString( this->type );          // access to "type"
                                      // now okay
   DrawString( "\pDangerous pet!" );
}
```

Continuing on with the assumption that the Animal data member type is public, an object of BadPet type can then call the new version of Write_Warning() to access type. Because type is public, the same object can also access the type data member without the use of a member function. Examples of both of these cases are shown here:

```
BadPet  *the_snake;

the_snake = new BadPet;

the_snake->Write_Warning();
Fill_Str255( the_snake->type, "\pPython snake" );
```

Figure 6–11 shows how the public keyword affects data member access. Note how the Animal class data can be changed from anywhere in a program.

Declaring base class data members to be public isn't the preferred method of allowing derived class access, because it has a significant negative consequence. As pointed out in Chapter 5, restricting access to class data members is a very important part of object-oriented programming. Opening up access to a class data member makes that data member behave like a global variable. Any function or object can then modify its value, which is something that is best avoided.

The correct way to give derived classes access to data members of the base class is to declare those data members *protected.* The protected keyword is a compromise between private and public. When a base class data member is protected, it can be accessed by member functions of that class as well as member functions of any class derived from that class. But it cannot be accessed by any other function or be accessed directly by an object. Here's the ideal way to define the Animal class:

```
class  Animal                              // base class
{
   protected:
      Str255  type;                        // data member

   public:
      void  New_Animal( Str255 );          // member function
      void  Write_Info( void );            // member function
};
```

When written this way, a member function of a derived class can access the type data member. With that in mind, you could rewrite the BadPet

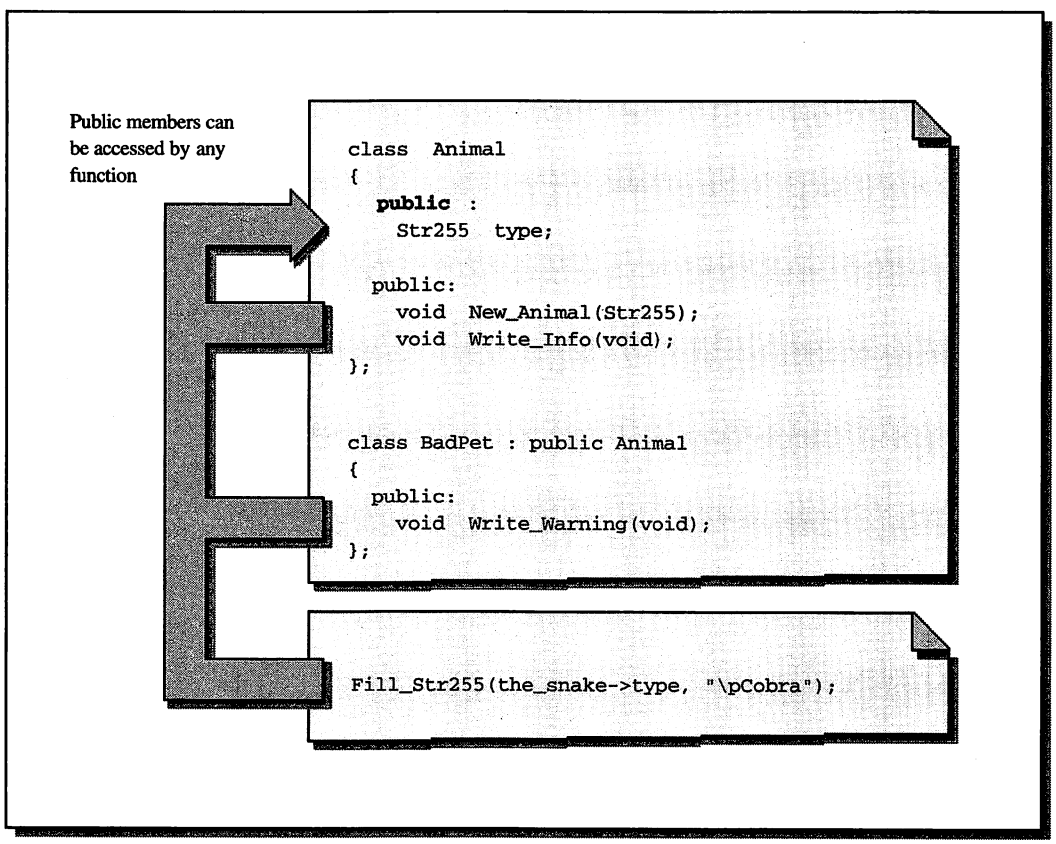


FIGURE 6–11 Public data members can be accessed from anywhere in a program.

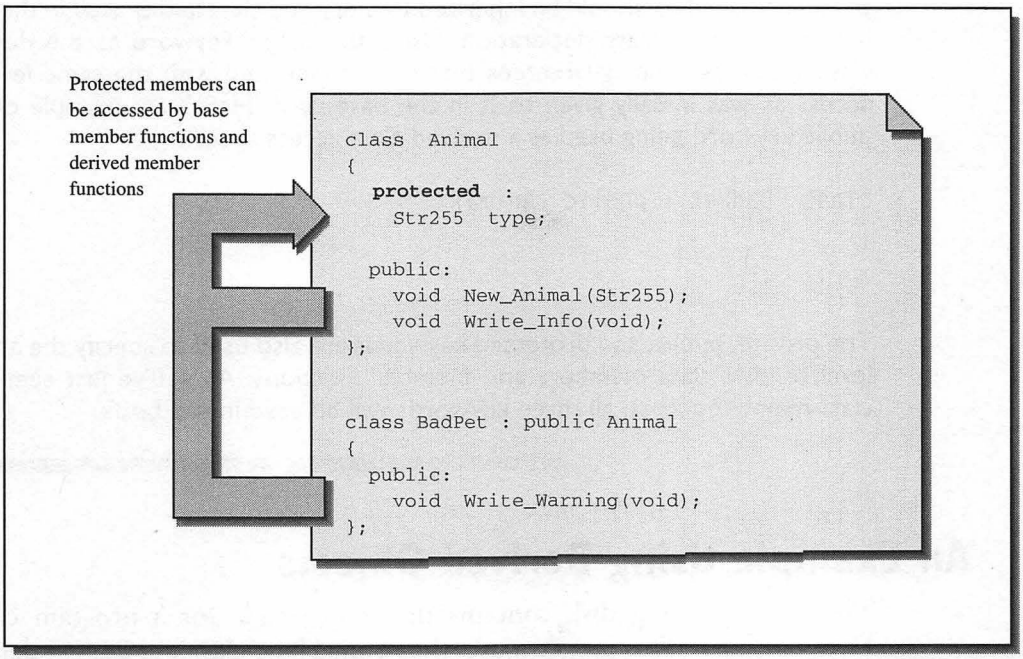


FIGURE 6–12 Protected data members can be accessed by class member functions and by derived classes.

member function Write_Warning() to write out the string stored in the type data member, followed by the warning string:

```
void  BadPet :: Write_Warning( void )
{
   DrawString( this->type );
   DrawString( "\p. . . is a dangerous pet!" );
}
```

Figure 6–12 shows how the protected keyword yields the perfect level of access for base data members.

IMPORTANT

Earlier in this chapter you read that the protected keyword would not be used as a derived class access specifier in this book. That use of the protected keyword is different from this use. The derived class access specifier tells the

compiler how data should be inherited by a derived class and is used in the first line of a derived class declaration. Using the public keyword as the derived class access specifier guarantees that data is inherited with the same level of access as was initially given to it in the base class. Here's an example of the public keyword being used as a derived class access specifier:

```
class  BadPet : public  Animal
{
   . . .
```

The private, public, and protected keywords are also used to specify the access level of class data members and member functions. As you've just seen, for class member access, all three keywords will be used in this book.

An Example Using Derived Objects

The accompanying disk contains the source code for a program called DerivedClass1. This program uses the Animal base class and BadPet derived class discussed in this chapter. Figure 6–13 shows what you'll see after you run DerivedClass1.

```
// ****************  DerivedClass1.cp  ****************

class  Animal                              // base class
{
   protected:
      Str255  type;                        // data member

   public:
      void  New_Animal( Str255 );          // member function
      void  Write_Info( void );            // member function
};

  class  BadPet : public  Animal             // derived class
{
   public:
      void Write_Warning( void );          // member function
};
```

```
void  Animal :: New_Animal( Str255 name )
{
   Fill_Str255( this->type, name );
}


void  Animal :: Write_Info( void )
{
   DrawString( this->type );
}


void  BadPet :: Write_Warning( void )
{
   DrawString( "\pDangerous pet!" );
}


Animal  *the_dog;          // declare object of type Animal
BadPet  *the_snake;        // declare object of type BadPet


void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );
```

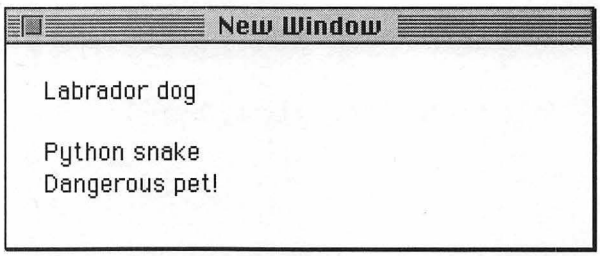


FIGURE 6–13 The output of the DerivedClass1 program

```
    the_dog   = new Animal;      // allocate memory for an
                                 // Animal object
    the_snake = new BadPet;      // allocate memory for a
                                 // BadPet object

    the_dog->New_Animal( "\pLabrador dog" );
    MoveTo( 20, 30 );
    the_dog->Write_Info();

    the_snake->New_Animal( "\pPython snake" );
    MoveTo( 20, 60 );
    the_snake->Write_Info();
    MoveTo( 20, 75 );
    the_snake->Write_Warning();

    while ( !Button() )
        ;
}
```

A Second Example Using Derived Objects

The BadPet derived class in the DerivedClass1 program inherited the data members and member functions of the Animal class. It also contained a single member function of its own. A C++ program can also contain a derived class that has its own data members. DerivedClass2 is such a program.

The DerivedClass2 program is DerivedClass1 with a few modifications—hence the clever name. The base class, Animal, is exactly the same as

you've seen in the past. The derived class, BadPet, has a couple of additions. Here's the new version:

```
class  BadPet : public  Animal          // derived class
{
   private:
      Str255  reason;                   // data member

   public:
      void  Add_Reason( Str255 );       // member function
      void  Write_Warning( void );      // member function
};
```

The BadPet class now has its own data member—a string named reason. This member will hold the reason that the pet is considered less desirable than others. The BadPet class has also gained a new member function—Add_Reason(). Since the new data member is private, the program has to have a way of accessing it from within the class. That's the purpose of Add_Reason().

```
void  BadPet :: Add_Reason( Str255 why )
{
   Fill_Str255( this->reason, why );
}
```

After declaring and creating a BadPet object, you can set the reason data member as follows:

```
the_snake->Add_Reason( "\pFirm grip!" );
```

Here's the complete source code listing for DerivedClass2. Figure 6–14 shows the output of the program. Figure 6–15 shows what memory might look like after an Animal object and a BadPet object have been created using the new operator.

```
// ****************  DerivedClass2.cp  ****************

class  Animal                           // base class
{
   protected:
      Str255  type;                     // data member
```

```
   public:
      void  New_Animal( Str255 );        // member function
      void  Write_Info( void );          // member function
};


class  BadPet : public  Animal           // derived class
{
   private:
      Str255  reason;                    // data member

   public:
      void  Add_Reason( Str255 );        // member function
      void  Write_Warning( void );       // member function
};


void  Animal :: New_Animal( Str255 name )
{
   Fill_Str255( this->type, name );
}

void  Animal :: Write_Info( void )
{
   DrawString( this->type );
}

void  BadPet :: Write_Warning( void )
{
   DrawString( "\pDangerous pet!" );
   Move( 10, 0 );
   DrawString( this->reason );
}

void  BadPet :: Add_Reason( Str255 why )
{
   Fill_Str255( this->reason, why );
}
```

```
Animal  *the_dog;          // declare object of type Animal
BadPet  *the_snake;        // declare object of type BadPet


void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   the_dog   = new Animal;  // allocate memory for an
                            // Animal object
   the_snake = new BadPet;  // allocate memory for a
                            // BadPet object

   the_dog->New_Animal( "\pLabrador dog" );
   MoveTo( 20, 30 );
   the_dog->Write_Info();

   the_snake->New_Animal( "\pPython snake" );
   MoveTo( 20, 60 );
   the_snake->Add_Reason( "\pFirm grip!" );
   the_snake->Write_Info();
   MoveTo( 20, 75 );
   the_snake->Write_Warning();

   while ( !Button() )
      ;
}
```

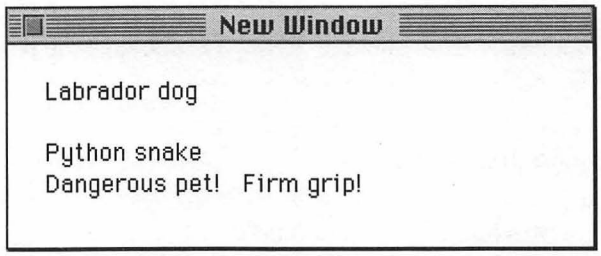


FIGURE 6–14 The output of the DerivedClass2 program

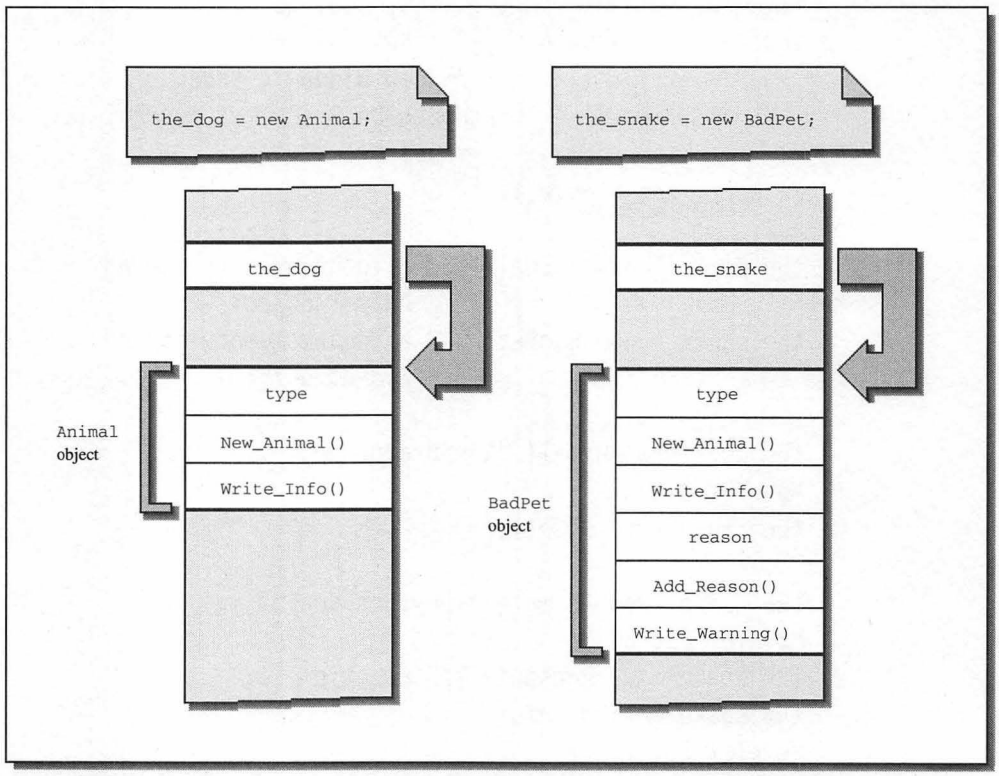


FIGURE 6–15 The Animal and BadPet objects in memory

Overriding Member Functions

When a class is derived from a base class, it inherits the member functions of the base class. But what if you want the derived object to perform an action that's just a little different from that performed by an inherited member function? Rather than require you to add a new function to the derived class, C++ provides a mechanism that allows you to alter the inherited function.

Why Override Member Functions?

Writing a warning message for bad pets is an idea that appeals to the pet shop owner. He wants to expand on it by including a text message for each type of animal in his shop—not just bad pets. Since most of his animals are good pets, he decides to modify the Write_Info() function of the Animal class so that it adds the message `Good pet!` after writing the animal type. This is how you've seen the Write_Info() routine in the past:

```
void  Animal :: Write_Info( void )
{
   DrawString( this->type);
}
```

Instead of just writing the value of the type data member, the new version of Write_Info() also moves over 10 pixels and writes the Good pet! message. Here's how the new version of Write_Info() looks:

```
void  Animal :: Write_Info( void )
{
   DrawString( this->type);
   Move( 10, 0 );
   DrawString( "\pGood pet!" );
}
```

The DerivedClass3 program is identical to the DerivedClass2 program, with the exception of the Write_Info() function. DerivedClass3 contains the new version shown above. Because of that, I won't provide the entire source code listing here. Instead, I'll just repeat the lines that work with the two objects:

```
the_dog->New_Animal( "\pLabrador dog" );
MoveTo( 20, 30 );
the_dog->Write_Info();

the_snake->New_Animal( "\pPython snake" );
the_snake->Add_Reason( "\pFirm grip!" );
MoveTo( 20, 60 );
the_snake->Write_Info();
MoveTo( 20, 75 );
the_snake->Write_Warning();
```

The first line in the above code assigns to the type data member of the Animal object the_dog the string Labrador dog. The next two lines move to the location to start drawing and then write out the animal information. At this point the output appears as shown in Figure 6–16.

So far, so good. The first task of the next lines of code, which are repeated below, is to assign to the type data member of the BadPet object the_snake a value of Python snake. Then the reason for giving this pet the bad pet status is assigned. Next, all the information is written out in a call to the inherited function Write_Info() and a call to BadPet's own member function Write_Warning(). Figure 6–17 shows how the program's output looks after the following code has run:

```
the_snake->New_Animal( "\pPython snake" );
the_snake->Add_Reason( "\pFirm grip!" );
MoveTo( 20, 60 );
the_snake->Write_Info();
MoveTo( 20, 75 );
the_snake->Write_Warning();
```

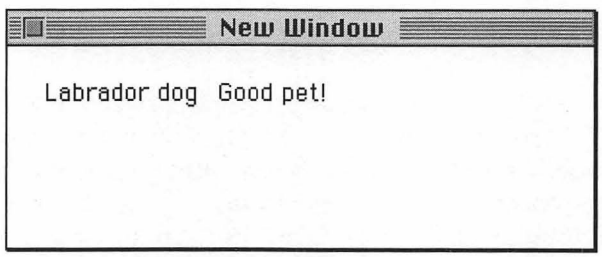


FIGURE 6–16 Part of the output of the DerivedClass3 program



FIGURE 6–17 The final output of the DerivedClass3 program

The output shown in the window in Figure 6–17 isn't quite what the pet shop owner had hoped for. The above code calls Write_Info() for the object `the_snake` so that the animal type can be written. But the owner has to remember that, when called, the Write_Info() function found in the Animal data class will also write the string Good pet! Since most of his pets are good pets, that's usually all right. But for exceptions—as in the case of the bad pet—the owner needs a way to write the animal type but not the Good pet! message. A slightly modified version of Write_Info() would be useful here. In fact, two versions of Write_Info()—one for the Animal class and one for the BadPet class—would be ideal. In C++, he can do that thanks to a concept called *member function overriding.*

Overriding a Function

Any function that appears in a base class can have an identically named function in a class derived from the base class. When an object of the base class calls the function, the base class function will execute. When an object of the derived class calls the same function, the version that appears in the derived class will execute instead. The derived class function overrides the base class function. Figure 6–18 clarifies this.

Previously, you saw that the pet shop owner wanted to come up with two separate Write_Info() routines—one for the Animal base class and one for the BadPet derived class. Member function overriding will provide a way to accomplish this. The following are new versions of the Animal and BadPet classes. In Animal, the Write_Info() function is now preceded by the C++ keyword virtual. And in BadPet, the Write_Warning() routine has been replaced by Write_Info(). I've used bold text to emphasize these changes:

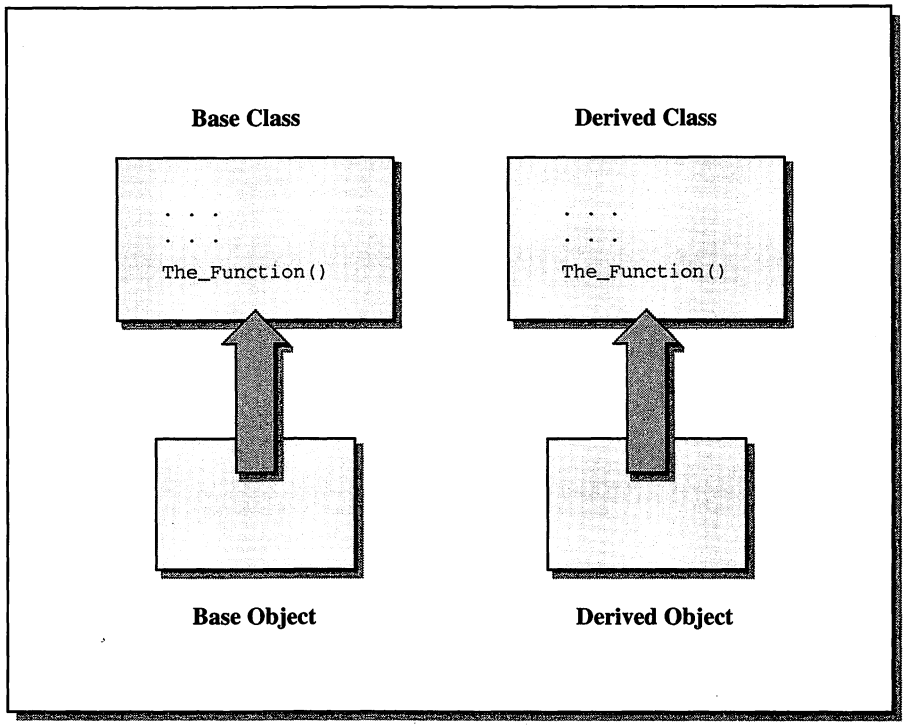


FIGURE 6–18 When function overriding is used, base and derived objects use different versions of a member function.

```
class  Animal                                  // base class
{
   protected:
      Str255  type;                            // data member

   public:
      void  New_Animal( Str255 );              // member function
      virtual  void  Write_Info( void );       // member function
};


class  BadPet : public  Animal                 // derived class
{
```

```
   private:
      Str255  reason;                    // data member

   public:
      void  Add_Reason( Str255 );        // member function
      void  Write_Info( void );          // member function
};
```

Before we discuss the virtual keyword, take a look at how I've written the two Write_Info() functions. The first version—the one associated with the Animal class—remains as it was written in the DerivedClass3 program. The new version—the one associated with the BadPet class—writes the type data member string and then a warning message.

```
void  Animal :: Write_Info( void )
{
   DrawString( this->type );
   Move( 10, 0 );
   DrawString( "\pGood pet!" );
}


void  BadPet :: Write_Info( void )
{
   DrawString( this->type );
   Move( 10, 0 );
   DrawString( "\pDangerous pet!" );
   Move( 10, 0 );
   DrawString( this->reason );
}
```

Overriding a member function has no impact on the way objects are declared and created. Thus, the following code, which allocates memory for an Animal base class object and a BadPet derived class object should look very familiar:

```
Animal  *the_dog;
BadPet  *the_snake;

the_dog   = new Animal;
the_snake = new BadPet;
```

To send a Write_Info() message to either object, just call the function as you've done in the past:

```
the_dog->Write_Info();
the_snake->Write_Info();
```

The above code will execute two different functions. Sending the same message to objects of two different classes is an object-oriented feature called *polymorphism.* How does the compiler know which Write_Info() should be called for each object? The virtual keyword provides the answer.

The virtual keyword lets the compiler know that there will be more than one version of a function. When creating a derived class function that is to override a base class function, precede the base class function with the word *virtual.* That's all there is to it. The compiler then keeps track of all derived classes that have a function of the same name. When an object receives a message to invoke a function, the compiler knows which version of the function to execute. Figure 6–19 illustrates this.

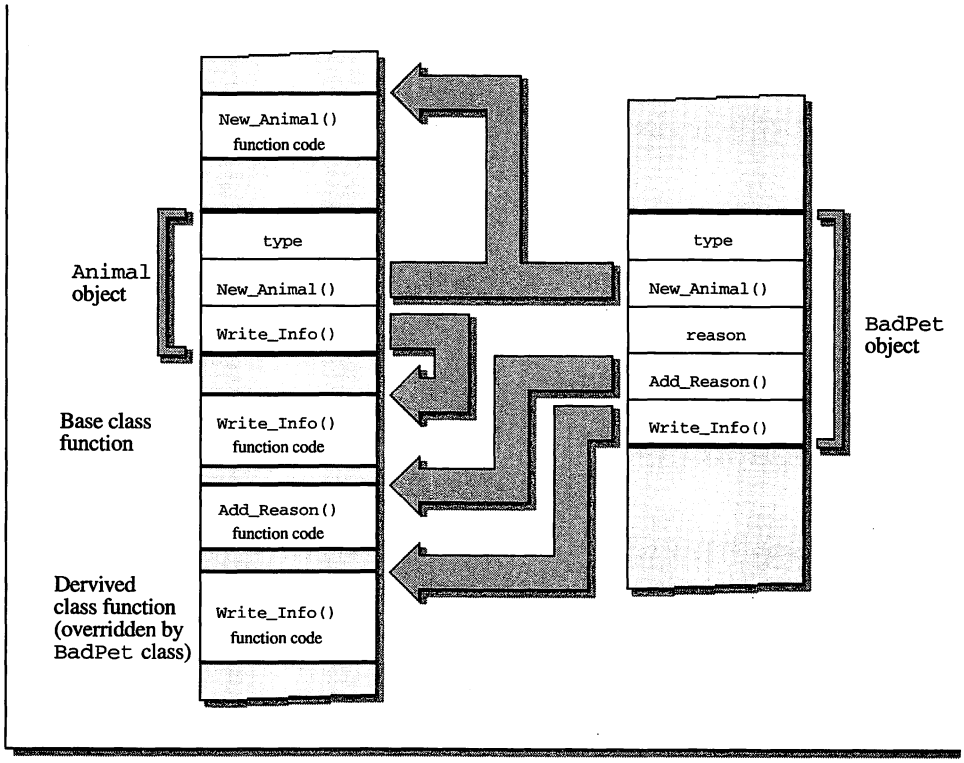


FIGURE 6–19 A base object, a derived object, and the functions they use

NOTE

Member function overriding is different from function overloading, which was discussed in Chapter 4.

Function overloading is a C++ feature that allows you to give two functions an identical name as long as the functions have a different number of arguments or different types of arguments. Function overloading can be used in object-oriented programs, but it isn't necessarily an object-oriented technique.

Member function overriding applies only to object-oriented programming. It allows a derived class to declare a function that overrides, or replaces, a like-named function in the base class. A derived class member function that overrides a base class member function can have both the same number of arguments and same type of arguments.

An Overriding Example

The Overriding program uses the Animal and BadPet classes with which you are already familiar. The derived class BadPet has a Write_Info() function that overrides the base class Write_Info() function.

The Overriding program produces the results the pet shop owner was looking for. Before looking over the Overriding source code listing, be sure that you understand why the program works. First take another look at the Animal base class; you'll want to refer to it as you walk through these lines of code from the Overriding program.

```
class  Animal                              // base class
{
   protected:
      Str255  type;                        // data member

   public:
      void  New_Animal( Str255 );          // member function
      virtual  void  Write_Info( void );   // member function
};
```

The following lines from main() work with the Animal object the_dog. The first line invokes the New_Animal() function. This routine simply copies the string Labrador dog into the type data member. After preparing to write out the Animal information, Write_Info() is called:

```
the_dog->New_Animal( "\pLabrador dog" );
MoveTo( 20, 30 );
the_dog->Write_Info();
```

Which of the two Write_Info() functions is called? The one associated with the Animal class. The virtual keyword in the Animal class definition doesn't affect this function in any way. It only tells the compiler that classes derived from the Animal class may override it. Here's the Write_Info() routine that is called:

```
void  Animal :: Write_Info( void )
{
   DrawString( this->type );
   Move( 10, 0 );
   DrawString( "\pGood pet!" );
}
```

Recall that this, when used in a member function, refers to the object that invoked the function. Here, the object is the_dog, and the value of type for the_dog is the string Labrador dog. After this string is written, the words Good pet! are drawn to the window. At this point, the program output appears as shown in Figure 6–20.

Next, the program works with the BadPet object the_snake. Here's another look at the derived class BadPet:

```
class  BadPet : public  Animal          // derived class
{
   private:
      Str255  reason;                   // data member
```

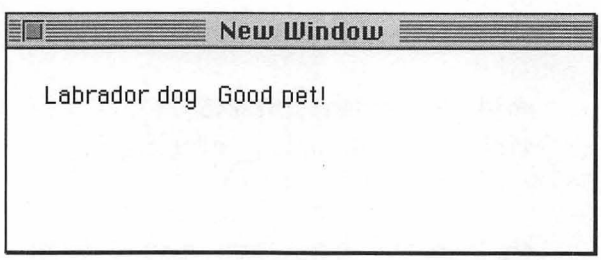


FIGURE 6–20 Part of the output of the Overriding program

```
   public:
      void  Add_Reason( Str255 );          // member function
      void  Write_Info( void );            // member function
};
```

First, New_Animal() is called to set the type data member belonging to the the_snake object to the string Python snake. The program has only one New_Animal() function, which is defined in the Animal class. The derived BadPet class inherits it:

```
the_snake->New_Animal( "\pPython snake" );
```

Next, the Add_Reason() function is called to set the reason data member of the_snake. Both the data member reason and the member function Add_Reason() are a part of the BadPet class; they are not inherited from the Animal base class.

```
the_snake->Add_Reason( "\pFirm grip!" );
```

Finally, a message is sent to the the_snake object to invoke the Write_Info() function:

```
MoveTo( 20, 60 );
the_snake->Write_Info();
```

Because Write_Info() was declared with the virtual keyword in the base class, the compiler knows to check to see if the BadPet derived class has its own copy of Write_Info(). It does, so that version of Write_Info() is used:

```
void  BadPet :: Write_Info( void )
{
   DrawString( this->type );
   Move( 10, 0 );
   DrawString( "\pDangerous pet!" );
   Move( 10, 0 );
   DrawString( this->reason );
}
```

Instead of writing the type data member and the Good pet! string as the Animal class Write_Info() does, this version of Write_Info() writes the type data member—a string that says Dangerous pet!—and the value in the

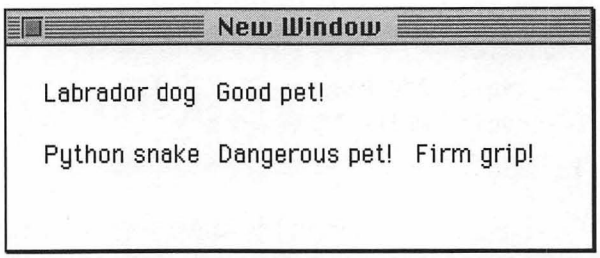


FIGURE 6–21 The final output of the Overriding program

object's reason data member. Figure 6–21 shows how the program's window now looks:

Here, in its entirety, is the Overriding source code listing. You'll also find the source code on the accompanying disk.

```
// ******************  Overriding.cp  ******************

class  Animal                                  // base class
{
   protected:
      Str255  type;                            // data member


   public:
      void  New_Animal( Str255 );              // member function
      virtual  void  Write_Info( void );       // member function
};


class  BadPet : public  Animal                 // derived class
{
   private:
      Str255  reason;                          // data member

   public:
      void  Add_Reason( Str255 );              // member function
      void  Write_Info( void );                // member function
};
```

```
void  Animal :: New_Animal( Str255 name )
{
   Fill_Str255( this->type, name );
}


void  Animal :: Write_Info( void )
{
   DrawString( this->type );
   Move( 10, 0 );
   DrawString( "\pGood pet!" );
}


void  BadPet :: Write_Info( void )
{
   DrawString( this->type );
   Move( 10, 0 );
   DrawString( "\pDangerous pet!" );
   Move( 10, 0 );
   DrawString( this->reason );
}


void  BadPet :: Add_Reason( Str255 why )
{
   Fill_Str255( this->reason, why );
}


Animal  *the_dog;        // declare object of type Animal
BadPet  *the_snake;      // declare object of type BadPet


void  main( void )
{
   WindowPtr   the_window;
```

```
    Rect        window_rect;


    InitGraf( &thePort );
    InitFonts();
    InitWindows();


    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    the_dog   = new Animal;   // allocate memory for an
                              // Animal object
    the_snake = new BadPet;   // allocate memory for a
                              // BadPet object


    the_dog->New_Animal( "\pLabrador dog" );
    MoveTo( 20, 30 );
    the_dog->Write_Info();


    the_snake->New_Animal( "\pPython snake" );
    the_snake->Add_Reason( "\pFirm grip!" );
    MoveTo( 20, 60 );
    the_snake->Write_Info();


    while ( !Button() )
       ;
}
```

Chapter Summary

Object-oriented programs usually consist of more than one class, and often those classes are related to one another. To simplify the process of defining new classes and to imply a relationship between classes, C++ allows you to name one class as a base from which other classes are derived. These *derived classes* automatically inherit the data members and member functions of the *base class* on which they are based.

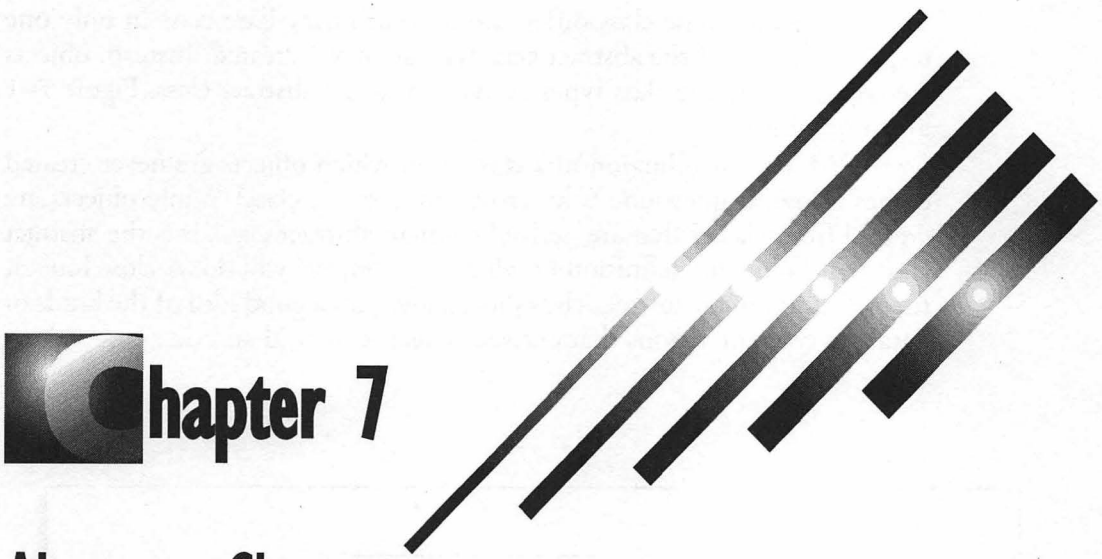
You need do nothing special to a class to make it a base class—any class will work. To define a class as a class derived from a base class, however, you must add a few things to the class header. First, add the name of the base class. Then add a colon followed by the public keyword. The following is the definition of a derived class named BadPet that is derived from a base class named Animal:

```
class  BadPet : public  Animal
{
   // BadPet data members and member functions
};
```

Pointers to objects—whether the objects are of a base class or a derived class—are declared by listing the class name followed by the * operator and the object name. Objects of a derived class are then created just as are objects of a base class—by using the new operator.

An object of a derived class inherits the data and functions of its base class. So an object derived from a base class can call all the member functions listed in the base class. If the base member function doesn't have the exact functionality that the derived class needs, the derived class can *override* the base class function and supply a new function with the same name. To give derived classes permission to override its member functions, the base class must use the virtual keyword before the member function's name in the base class definition. Here a base class named Animal makes its member function Write_Info() available for overriding by derived classes:

```
class  Animal
{
   public:
      virtual  void  Write_Info( void );
};
```



Chapter 7

Abstract Classes

Good object-oriented programming technique involves the careful planning of what classes should be created to represent the data and actions that program objects will need. Past chapters introduced you to classes and objects but didn't dwell on how to choose the data members and member functions of the classes. Now that you understand the basics of classes, it's time to learn how to define a program's classes.

In Chapter 6, you saw that one or more classes can be derived from a single base class. These derived classes inherit the data and functions of the class from which they are derived. So it would seem that a wise choice of a base class is an important part of object-oriented programming. This chapter takes a close look at base classes. In particular, the focus will be on a special kind of base class called an *abstract class.*

Why Abstract Classes?

One very common use of a base class is to make it an abstract class. An abstract class can be thought of as a common ancestor of a family of derived classes.

An abstract base class differs from an ordinary base class in only one respect—objects of the abstract class type are never created. Instead, objects are created using the class types derived from the abstract class. Figure 7–1 illustrates this idea.

With the introduction of a class from which objects are never created comes a logical question: Why create an abstract class? While objects are created from classes that are derived from an abstract class, it is the abstract class that holds the definition of what those objects will do. A close look at the definition of an abstract class should give you a good idea of the kinds of data and types of actions that derived objects can hold and do.

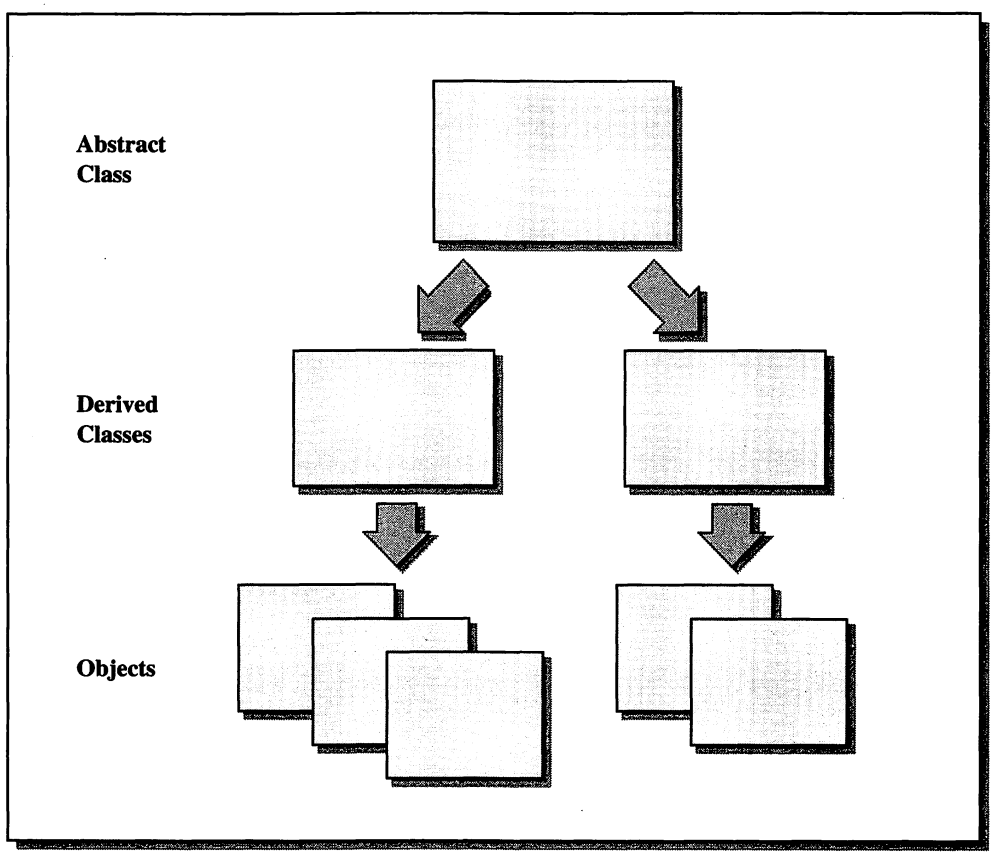


FIGURE 7–1 Objects are created from classes derived from the abstract class—not from the abstract class itself.

Creating an Abstract Class

The pet shop owner, not having read a C++ book, became overwhelmed by object-oriented programming and allowed me to take over his database program. Since you're familiar with the pet shop example, I'll carry on with it throughout the remainder of this book.

Like any good programmer, I gave some serious thought to what the program should do. I came up with a single base class that will serve as an abstract class. The abstract base class is the Animal class—with a few additions. Here's how the Animal class definition now looks:

```
class  Animal
{
   protected:
      Str255  type;
      Str255  desc;
      long    cost;

   public:
      virtual void  Set_Type( Str255 );
      virtual void  Get_Type( Str255 );
      virtual void  Set_Desc( Str255 );
      virtual void  Get_Desc( Str255 );
      virtual void  Set_Cost( long );
      virtual long  Get_Cost( void );
      virtual void  Write_Info( void );
};
```

Remember, an abstract class defines what its derived classes can do. I've filled out the Animal class so that it can store and work with all the animal information the pet shop owner will need to keep track of.

The Abstract Class Data Members

The Animal class now has three data members. You're familiar with the first member, `type`. The `type` member keeps track of the type of animal in the form of a string—White rat, for example.

The second data member, `desc`, is a string that holds a description of the animal—not popular, for example. In previous versions of the Animal class the description was passed to Write_Info(). It makes more sense,

however, to store the description as a part of the object. Then it can be saved, changed, or written to a window at any time.

Keeping track of names is a very important part of many data structures. That's why I've used string data members in the class examples. But numbers are just as important, so Animal now has a long data member that can be used in future examples. The cost data member is a long integer that holds the dollar amount that the pet shop charges for an animal of this type.

The Abstract Class Member Functions

The Animal class has seven member functions, each of which is preceded by the virtual keyword. That means that any of the member functions can be overridden by any class that is derived from Animal. Will each derived class in fact override each member function? That's not important. What is important is that you give derived classes the power to do so. The virtual keyword does just that.

You'll notice the words *Get* and *Set* in most of the member function names. A program needs to be able to *set* a data member's value and *get* that member's value. Because data members are accessed through the class member functions, there are usually two member functions for each data member—one to set a data member and one to get, or return, a member. As an example, the type data member has the Set_Type() function to provide it with a value and the Get_Type() function to return its value to the program. Here are those functions:

```
void  Animal :: Set_Type( Str255 name )
{
   Fill_Str255( this->type, name );
}


void  Animal :: Get_Type( Str255 name )
{
   Fill_Str255( name, this->type );
}
```

Set_Type(), which you've seen before as New_Animal(), assigns, or sets, the type member by calling Fill_Str255() to assign to type the value of the passed-in string. Get_Type() simply copies the string in the type member to the passed-in string variable name. After a Get_Type() message is passed to an object, the variable name will hold the value of the object's type data member.

Just as the type data member has Set and Get functions, so does the desc data member. Because desc is a Str255, the member functions that set and get its value work in the same way that the ones for the type Str255 data member work:

```
void  Animal :: Set_Desc( Str255 desc )
{
   Fill_Str255( this->desc, desc );
}


void  Animal :: Get_Desc( Str255 comment )
{
   Fill_Str255( comment, this->desc );
}
```

To set the cost data member, Set_Cost() simply accepts a single passed-in parameter that holds the price of the pet. To get the value of the cost data member, Get_Cost() uses the return keyword to pass the cost back to the program. Get_Cost() is the first example of a member function that has a return type other than void.

```
void  Animal :: Set_Cost( long price )
{
  this->cost = price;
}


long  Animal :: Get_Cost( void )
{
  return ( this->cost );
}
```

NOTE

Remember your C: the return keyword passes back the value of whatever lies after it. Passes back to where? To the calling function.

The last member function in the Animal abstract class is Write_Info(). This routine writes all the information about one animal to a window.

```
void  Animal :: Write_Info( void )
{
   Str255  str;

   MoveTo( 20, 30 );
   DrawString( this->type );
   MoveTo( 20, 45 );
   DrawString( this->desc );
   NumToString( this->cost, str );
   MoveTo( 20, 60 );
   DrawString( "\p$");
   DrawString( str );
}
```

Creating a Family of Classes

This chapter has an example program that uses the Animal class. But the program does not declare any Animal objects. Instead, the example creates objects only from classes derived from the Animal class. That's what makes the Animal class an abstract class. The abstract class, along with the class derived from it, can be thought of as a family of classes.

The Derived Classes

At this point I've identified two different general types of animals in the pet shop—animals that are considered good pets and animals that might be considered bad pets. With that in mind, I've created two classes—the BadPet class and the GoodPet class, both of which are derived from the Animal class.

```
class  BadPet : public  Animal
{
   public:
      void  Write_Info( void );
};
```

```
class  GoodPet : public  Animal
{
   public:
      void  Write_Info( void );
};
```

Both the BadPet class and the GoodPet class look trivial. But keep in mind that each inherits every data member and member function of its base class—the Animal class. An object of the BadPet class, for example, would look like an object of the Animal class. The only difference would be that the pointer to the Write_Info() function would point to a different version of Write_Info() for each of the two objects. That's because the BadPet class overrides the Animal Write_Info() member function. Figure 7–2 reinforces this idea.

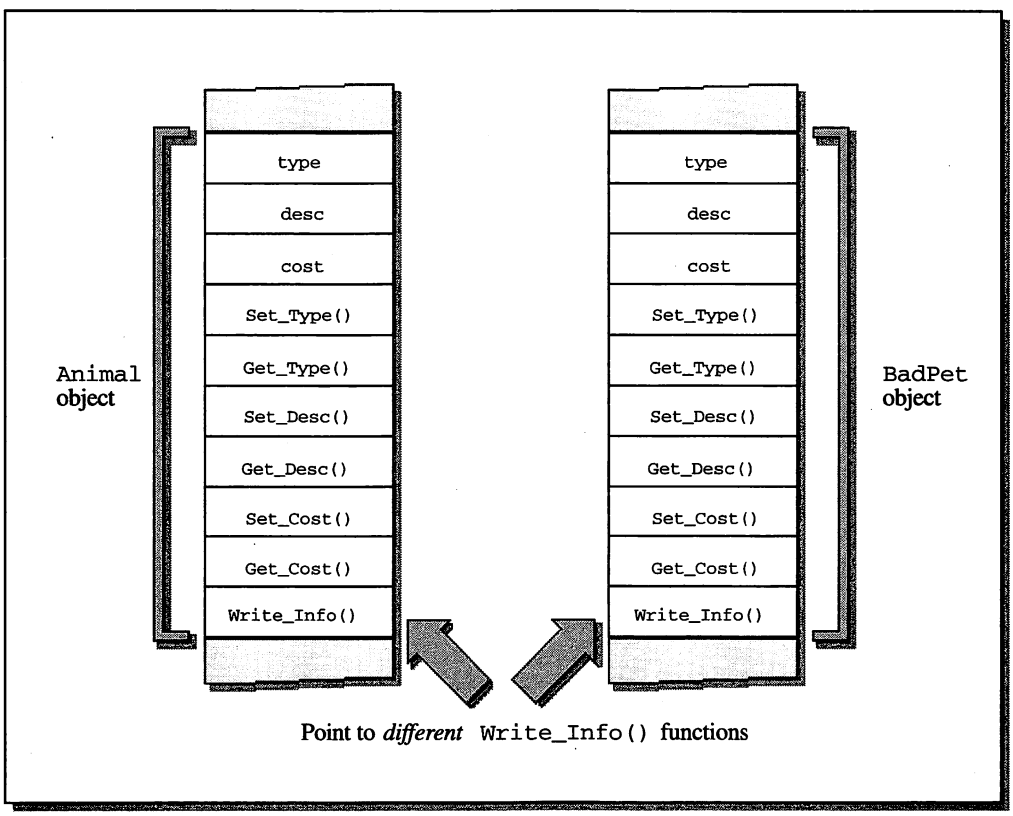


FIGURE 7–2 A derived class inherits the elements of the base class but may override one or more base class functions.

NOTE

I said that objects are not created from the abstract class, yet Figure 7–2 shows what an Animal object would look like. There's nothing about an abstract class definition that prevents objects being created from it. It's the programmer's choice to not do so.

The Member Functions of the Derived Classes

Both the BadPet and the GoodPet class are content to use the six Get and Set functions as defined by the Animal base class. Only the Write_Info() class will be overridden. Here's how BadPet overrides WriteInfo():

```
void  BadPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** OMIT? **");
}
```

The first line in the body of Write_Info() should be of interest to you:

```
Animal :: Write_Info();
```

Recall from Chapter 4 that the scope resolution operator can be used to tell the compiler which variable to use when confronted with two like-named variables:

```
long  days = 365;          // global to entire program

void  main( void )
{
   long  days = 31;        // local to main()
   long  days_in_year;

   days_in_year = ::days;  // uses global version of days
}
```

More recently, you've seen the scope resolution operator used in the headers of class member functions. For the Write_Info() function of the BadPet class, the header looks like this:

```
void  BadPet :: Write_Info( void )
```

Here the :: operator tells the compiler that this version of Write_Info() should be associated with the BadPet class. The :: operator can also be used when making a call to a class member function. A member function can be invoked via an object, as is being done here:

```
the_dog->Write_Info();
```

NOTE

In this example, all member functions but the Write_Info() member function of the Animal abstract class work as is for the derived class. This isn't always the case, however. While an abstract class should define the actions common to its derived classes, it does not necessarily have to provide the functionality of those actions. Thus, an abstract class may list several member functions, but many of the functions may be overridden by the member functions of a derived class.

A member function can also be called from within a different member function. That's what the first line in the Write_Info() function is doing. Here's that function again:

```
void  BadPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** OMIT? **");
}
```

The first line in the function body calls the Write_Info() function. But which one? All three classes—Animal, BadPet, and GoodPet—have defined a Write_Info() function. By prefacing the function call with the name of the class associated with the function, you're telling the compiler which version to use.

Because Write_Info() is listed as a virtual function in the Animal class and is also listed in the BadPet class, the function is overridden in BadPet. Thus, a

call to Write_Info() by a BadPet object will result in the BadPet version of the function being called. But within the BadPet version, the Animal version is called! What is the net effect of a call to the BadPet version of Write_Info()? Both it and the Animal version are called, as shown in Figure 7–3.

Calling a base member function from within a derived member function is a common action. It allows a class to both override a base class member function and make use of the overridden function. With this option of calling a base member function, it turns out that a member function can override a base member function in order to replace it or override the function in order to supplement it, as shown in Figure 7–4.

In Figure 7–4, a base class called BClass is defined along with one derived class named DClass. DClass inherits The_Func()—the one member function of BClass. It then overrides it by listing it as one of its own functions. When the DClass version of The_Func() is created, it can be written in one of two ways. By invoking the base class version of the function, you can supplement the base class version. The function can also be written so that it completely replaces the base class version. To do that, you would not invoke the BClass version. Figure 7–4 shows both options for writing an overriding function.

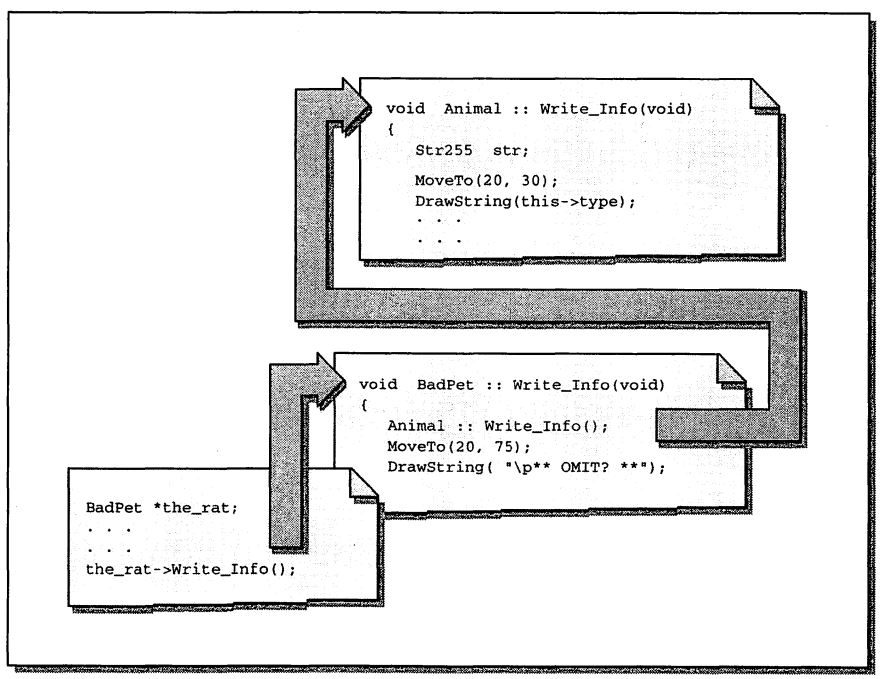


FIGURE 7–3 Member functions can invoke, or call, other member functions.

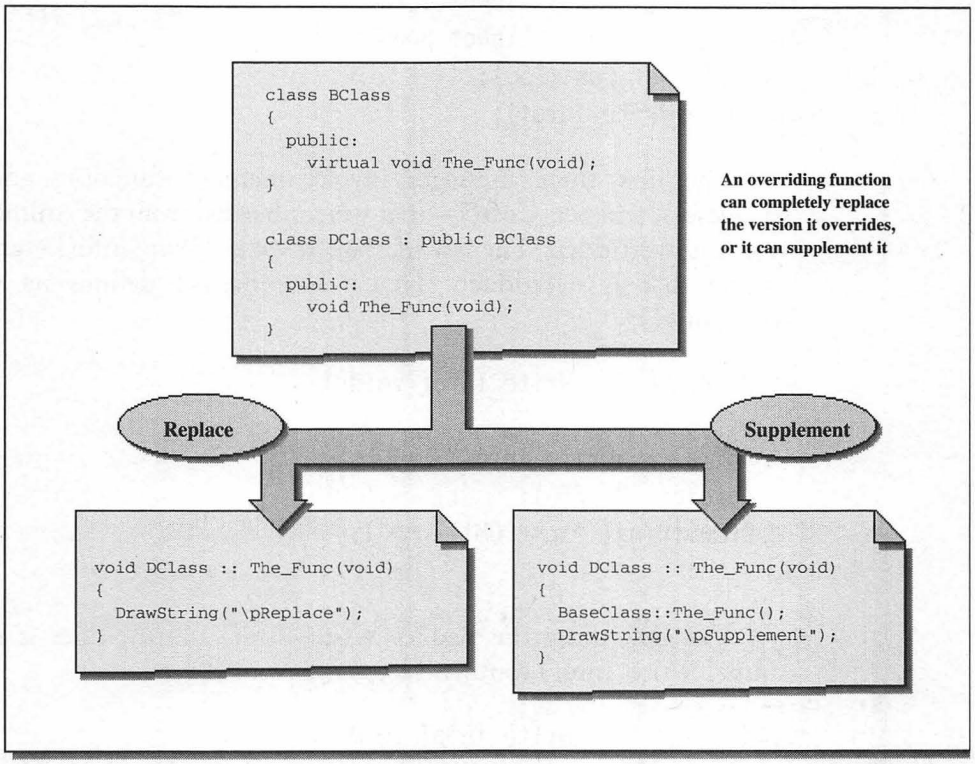


FIGURE 7–4 A function can override another function so that it replaces or supplements the base function.

An Abstract Example

AbstractClass is a program that puts to use the Animal abstract base class and the GoodPet and BadPet derived classes that you've seen throughout this chapter. Since this example is meant to illustrate how an abstract class works, it declares no Animal objects. Instead, the program declares one GoodPet object and one BadPet object:

```
GoodPet  *the_dog;     // declare object of type GoodPet
BadPet   *the_rat;     // declare object of type BadPet
```

The program opens a window and allocates memory for the BadPet object. Then four messages are sent to the object:

```
the_dog->Set_Type( "\pWhite rat" );
the_dog->Set_Desc( "\pNot popular");
the_dog->Set_Cost( 5 );
the_dog->Write_Info();
```

The first three messages invoke member functions—Set_Type(), Set_Desc(), and Set_Cost()—that were inherited from the Animal class but were not overridden. The last message invokes Write_Info()—an inherited routine that is overridden. Here's how BadPet defines its version of Write_Info():

```
void  BadPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** OMIT? **");
}
```

The first thing the BadPet Write_Info() routine does is invoke the Animal Write_Info() routine. Here's the Animal version:

```
void  Animal :: Write_Info( void )
{
   Str255  str;


   MoveTo( 20, 30 );
   DrawString( this->type );
   MoveTo( 20, 45 );
   DrawString( this->desc );
   NumToString( this->cost, str );
   MoveTo( 20, 60 );
   DrawString( "\p$");
   DrawString( str );
}
```

Because the BadPet version of Write_Info() calls the Animal version of Write_Info(), you should expect to see the output generated from the Animal version in a window, as shown in Figure 7–5.

Invoking the Animal version of Write_Info() isn't the only thing that the BadPet Write_Info() routine does. It also writes the string ** OMIT? **.

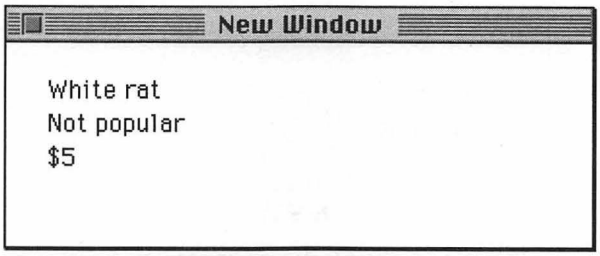


FIGURE 7–5 The results window after one Write_Info() routine invokes the Animal :: Write_Info() routine

So after the execution of the entire BadPet Write_Info() routine, you'll see results like those pictured in Figure 7–6.

To demonstrate that the GoodPet class works, the AbstractClass program opens a second window and sends to a GoodPet object the following four messages:

```
the_dog->Set_Type( "\pLabrador dog" );
the_dog->Set_Desc( "\pFriendly");
the_dog->Set_Cost( 100 );
the_dog->Write_Info();
```

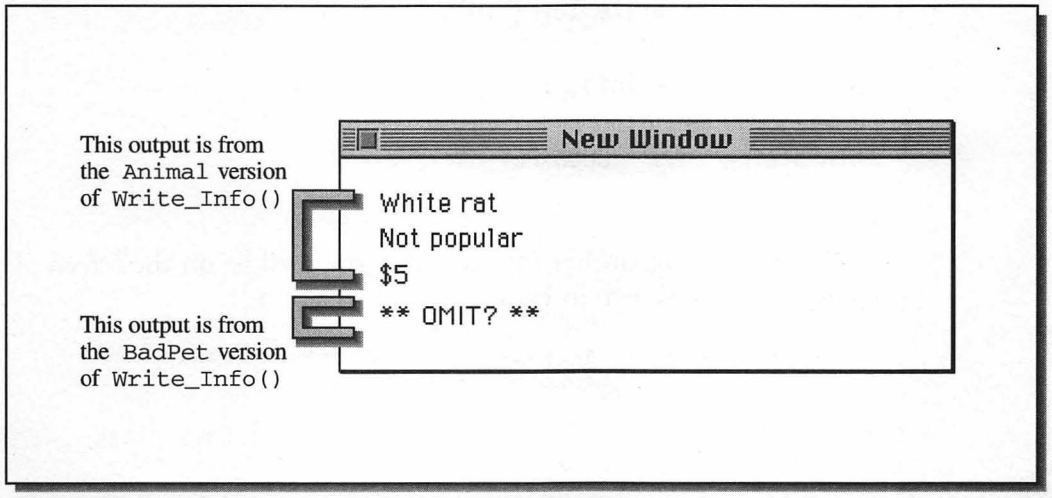


FIGURE 7–6 Program output comes from two separate Write_Info() functions.

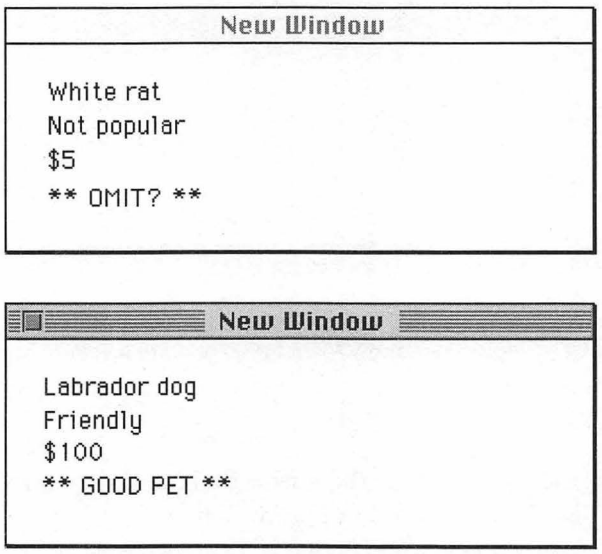


FIGURE 7–7 The final output of the AbstractClass program is drawn in two separate windows.

The GoodPet derived class has its own version of the Write_Info() function. Here's a look at it:

```
void  GoodPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** GOOD PET **");
}
```

When the program has run, two windows will be on the screen. The program output is shown in Figure 7–7.

```
// *****************  AbstractClass.cp  *****************

class  Animal                          // abstract base class
{
   protected:
      Str255  type;
```

```
      Str255  desc;
      long    cost;

   public:
      virtual void  Set_Type( Str255 );
      virtual void  Get_Type( Str255 );
      virtual void  Set_Desc( Str255 );
      virtual void  Get_Desc( Str255 );
      virtual void  Set_Cost( long );
      virtual long  Get_Cost( void );
      virtual void  Write_Info( void );
};


class  BadPet : public  Animal      // derived class
{
   public:
      void  Write_Info( void );
};


class  GoodPet : public  Animal     // derived class
{
   public:
      void  Write_Info( void );
};


void  Animal :: Set_Type( Str255 name )
{
   Fill_Str255( this->type, name );
}


void  Animal :: Get_Type( Str255 name )
{
   Fill_Str255( name, this->type );
}
```

```
void  Animal :: Set_Desc( Str255 desc )
{
   Fill_Str255( this->desc, desc );
}


void  Animal :: Get_Desc( Str255 comment )
{
   Fill_Str255( comment, this->desc );
}


void  Animal :: Set_Cost( long price )
{
  this->cost = price;
}


long  Animal :: Get_Cost( void )
{
  return ( this->cost );
}


void  Animal :: Write_Info( void )
{
   Str255  str;

   MoveTo( 20, 30 );
   DrawString( this->type );
   MoveTo( 20, 45 );
   DrawString( this->desc );
   NumToString( this->cost, str );
   MoveTo( 20, 60 );
   DrawString( "\p$");
   DrawString( str );
}
```

```
void  BadPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** OMIT? **");
}


void  GoodPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** GOOD PET **");
}


GoodPet  *the_dog;      // declare object of type GoodPet
BadPet   *the_rat;      // declare object of type BadPet


void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   the_rat = new BadPet;
```

```
    the_rat->Set_Type( "\pWhite rat" );
    the_rat->Set_Desc( "\pNot popular");
    the_rat->Set_Cost( 5 );
    the_rat->Write_Info();

    SetRect( &window_rect, 50, 200, 350, 300 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    the_dog = new GoodPet;

    the_dog->Set_Type( "\pLabrador dog" );
    the_dog->Set_Desc( "\pFriendly");
    the_dog->Set_Cost( 100 );
    the_dog->Write_Info();

    while ( !Button() )
        ;
}
```

The Class Hierarchy

A base class and the classes that are derived from it form a class hierarchy. For simple programs, like this chapter's AbstractClass example, it's easy to visualize this hierarchy, as shown in Figure 7–8.

It probably seems like overkill to draw a class hierarchy for a program as simple as AbstractClass, because you can easily visualize in your head just how the three classes are related. But for larger programs with several base classes and many more derived classes, a pictorial class hierarchy does help you keep track of things.

Symantec C++ makes it easy to obtain a view of your program's class hierarchy. When you're in the THINK Project Manager, the class hierarchy is just a mouse click away. With a project open and compiled, select the Browser menu item from the Source menu, which is shown in Figure 7–9.

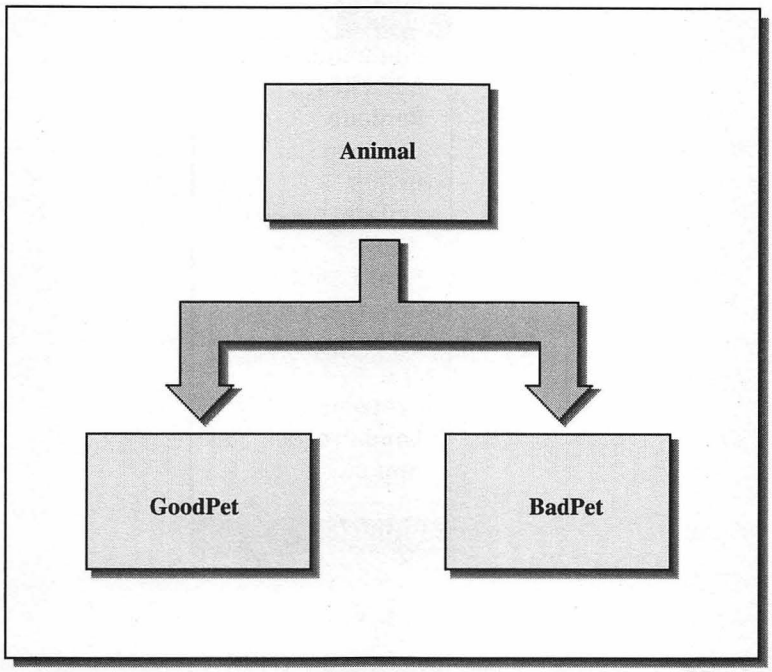


FIGURE 7–8 The class hierarchy for the AbstractClass program

When you select Browser, a THINK Class Browser window will open. The Browser is a tool that displays the class hierarchy of the classes in any THINK project. Figure 7–10 shows the window that opens when Browser is selected for this chapter's AbstractClass project.

The Browser displays the *root* class—the class that has no base class—on the far left side of the window. For AbstractClass, the root class is the Animal class. Lines run from the root class to each class that is derived from it. For AbstractClass, those lines run from Animal to BadPet and from Animal to GoodPet, as shown in Figure 7–10.

Clicking on a class name in the Class Browser window displays a pop-up menu that lists the member functions of the class. In Figure 7–11, I've clicked on the GoodPet class. The pop-up menu displays the one member function that is defined in GoodPet—the Write_Info() function. Note that only member functions that are defined in the class—not inherited member functions—are displayed. To see all the inherited member functions, click on the Animal class.

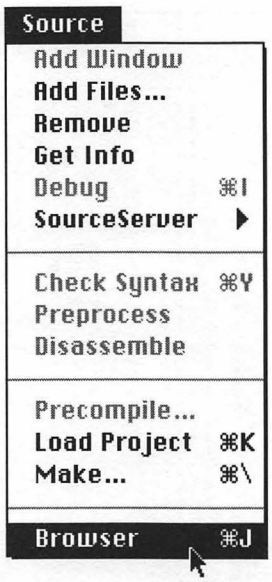


FIGURE 7–9 Selecting the Browser item from the THINK Project Manager's Source menu

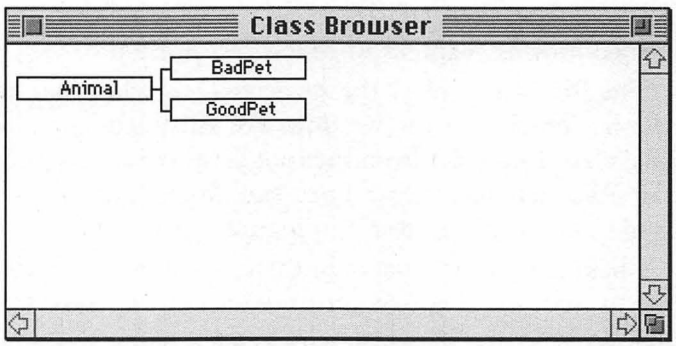


FIGURE 7–10 The class hierarchy for the AbstractClass program as displayed in the Class Browser

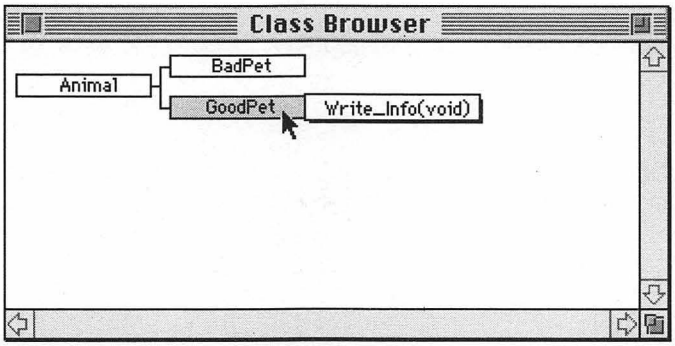


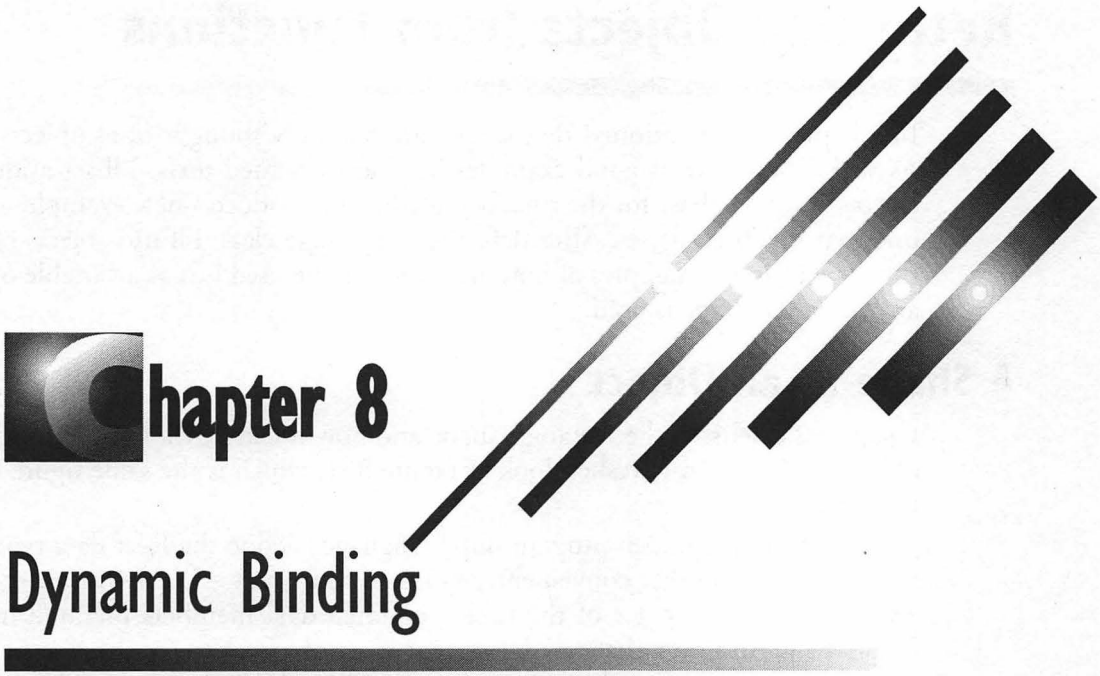
FIGURE 7–11 Using the Browser to view the member functions of the GoodPet class

At this point, the Browser may seem to be of limited use. But as your programs grow in size and complexity, the Browser will become a much more valuable resource. The C++ language allows you to derive classes from other derived classes. In a program that does that, the class hierarchy can become quite complex.

Chapter Summary

Derived classes inherit the data members and member functions of the class from which they are derived. When a program declares object variables, it can declare them to be of the base class type or of any one of the derived class types. But in most programs that use derived classes, objects are never declared to be of the base class. Instead, the base class is used only as the pattern from which a family of derived classes is defined. When this situation occurs, the base class is said to be an *abstract* class.

A base class and the classes that are derived from it form a *class hierarchy.* For simple programs that consist of only a few classes, it's very easy to visualize what this hierarchy looks like For larger programs that contain several classes, however, a pictorial class hierarchy helps to keep track of things. The THINK Project Manager makes it easy to obtain a view of your program's class hierarchy. To do so, first compile your project's source code. Then select the Browser menu item from the Source menu. A THINK Class Browser window will open, and in it will be displayed the class hierarchy of the classes in the currently opened project.



Chapter 8

Dynamic Binding

Objects that are derived from a base class often have member functions that override those found in the base class. That means that a program may have several different types of objects, each making calls to like-named functions. Although the functions have the same name, they are represented by different code. To add to this complexity, programs often create objects at runtime—or, "on the fly." That means that at the time the source code is compiled, the compiler might not know what type of object is being created. Yet the compiler will still allow this object of unknown type to make calls to member functions. How the compiler allows this to happen and how the compiled application handles these unresolved situations is through *dynamic binding.*

On your way to discovering just what dynamic binding is, you'll see several example programs that introduce a set of new classes—classes that work with shapes. In these examples, you'll see how objects can be used in functions—functions that are associated with a class and functions that aren't.

Returning Objects from Functions

In Chapter 2, I mentioned that shapes are naturally thought of as objects. As such, they serve as good examples in object-oriented texts. I'll set aside the pet shop database for the time being while I introduce a new example—one that involves shapes. After defining a rectangle class, I'll use objects of this type in some examples of how an object can be used just as a variable of any other data type is used.

A Shape as an Object

Chapter 2 discussed the rectangle shape and how it can be thought of as an object in C++. As a refresher, look at Figure 8–1, which is the same figure I used back in Chapter 2.

Because Macintosh programming languages define the Rect data type as a data structure that conveniently holds all four values that define a rectangle, I'll use it in place of the four individual data members pictured in Figure 8–1. And now that you know that most objects have member functions to set and get the values of the data members, I'll include a function to set the Rect data member. For the sake of simplicity I'll forego the "get" member function, as well as the member functions to grow and erase the rectangle object. To give the object something to do, I'll just include a member function to draw the rectangle. Figure 8–2 shows what my simple rectangle object will look like.

The Rectangle class has two member functions, both of which I'll define to be virtual functions. If in the future I decide to create any derived classes from the Rectangle class, the derived classes can override any of the Rectangle member functions. Here's the Rectangle class:

```
class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short, short );
      virtual void  Draw_Rectangle( void );
};
```

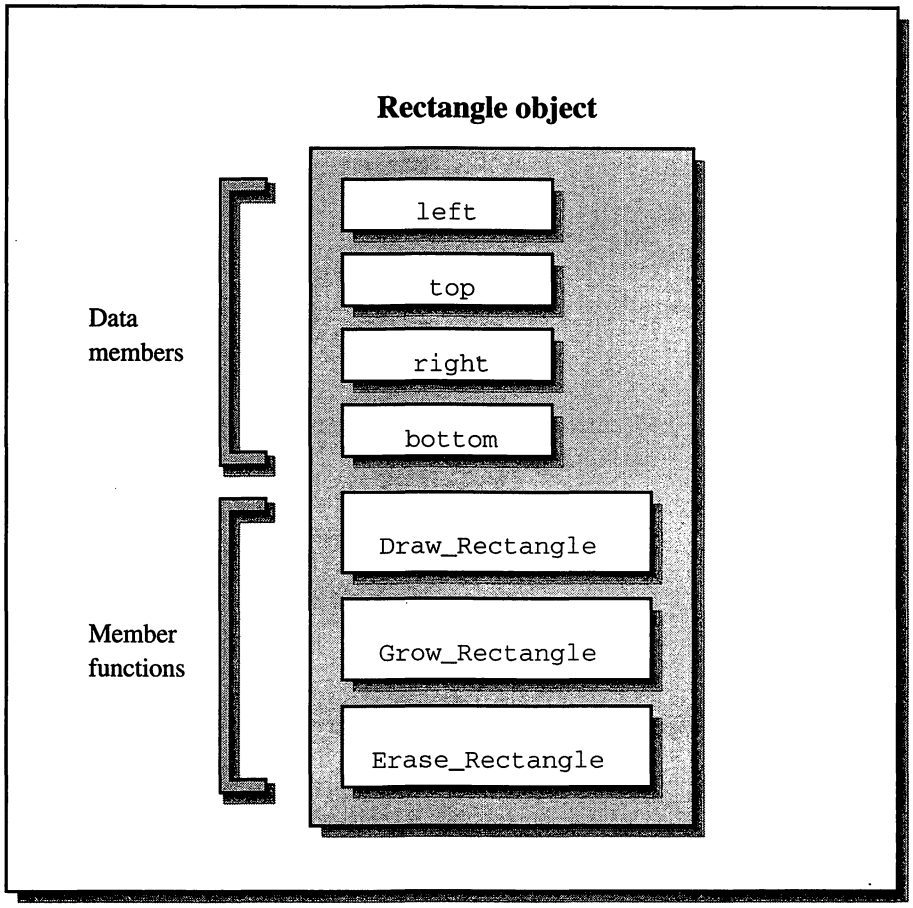


FIGURE 8–1 Representation of a rectangle as an object

To establish the coordinates of the aRect member, Set_Rectangle() will be called. Pass the four coordinates to Set_Rectangle(), and it will call the Toolbox routine SetRect(). Because SetRect() accepts a pointer to a Rect as its first argument, I'll preface this->aRect with the & operator. Here's the "set" member function for the Rectangle class:

```
void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}
```

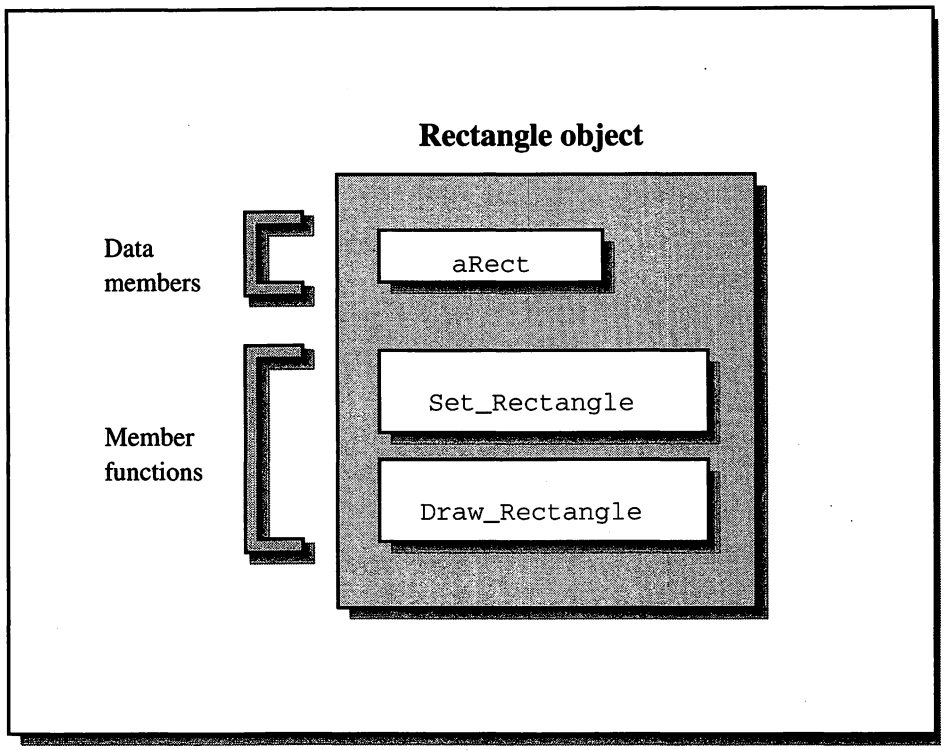


FIGURE 8–2 Another way of representing a rectangle as an object

I'll define drawing a rectangle to mean framing it. I could make things more complex, and more interesting, by having Draw_Rectangle() accept a Pattern as an argument and then fill and frame the rectangle. But again, simplicity wins out. Here's the Draw_Rectangle() member function:

```
void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}
```

Before seeing how an object is used as the return value of a function, let's test the new class by taking a look at a short program called RectangleClass. Three lines of RectangleClass do the work of creating a new Rectangle object, setting its boundaries, and then drawing it to a window:

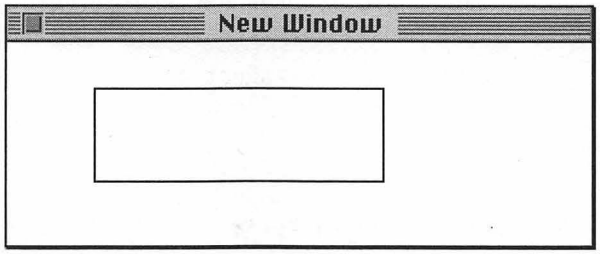


FIGURE 8–3 The output of the RectangleClass program

```
the_rect = new Rectangle;

the_rect->Set_Rectangle( 50, 25, 200, 75 );
the_rect->Draw_Rectangle();
```

Figure 8–3 shows what the window opened by RectangleClass will look like.

Now, the code for RectangleClass. You'll also find the code and project on the accompanying disk.

```
// *****************  RectangleClass.cp  ****************

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}
```

```
void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


Rectangle  *the_rect;          // declare object of type
                               // Rectangle


void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;


   InitGraf( &thePort );
   InitFonts();
   InitWindows();


   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );


   SetPort( the_window );

   the_rect = new Rectangle;

   the_rect->Set_Rectangle( 50, 25, 200, 75 );
   the_rect->Draw_Rectangle();

   while ( !Button() )
      ;
}
```

Returning an Object

As discussed in Chapter 5, once a class is defined, an instance, or object, of that class is created using the new operator:

```
Rectangle  *the_rect;  // declare object of type Rectangle

the_rect = new Rectangle;
```

So far, you've only seen the new operator used from within main(); all new objects have been created in main(). As the heading of this section indicates, an object can be created outside of main(). And if it is, it can be returned to main() or any other function using the *return* keyword. The New_Rectangle() function listed below is an example.

```
Rectangle  *New_Rectangle( void )
{
   Rectangle  *temp;

   temp = new Rectangle;

   return ( temp );
}
```

The return type of New_Rectangle() is an pointer to a Rectangle object. The function itself declares a variable named `temp` to be a pointer to a rectangle. The new operator sets aside memory for the Rectangle, and the return keyword passes this pointer back to the calling routine. Let's see how a call to New_Rectangle() looks:

```
Rectangle  *New_Rectangle( void )
{
   Rectangle  *temp;

   temp = new Rectangle;

   return ( temp );
}


Rectangle  *the_rect;  // declare object of type Rectangle
```

```
void  main( void )
{
   // initialize Mac, open a window

   the_rect = New_Rectangle();

   // remainder of the program
}
```

New_Rectangle() uses a variable named temp to allocate the memory for the Rectangle. Because temp is a local variable, its scope is confined to the New_Rectangle() function. On the left side of Figure 8–4 you can see how memory might look after the new operator is executed in New_Rectangle(). There a block of memory has been allocated, and temp points to it. The global Rectangle pointer variable the_rect hasn't yet received a value. When the call to New_Rectangle() has been completed, memory appear as pictured on the right side of Figure 8–4. There, the_rect has been assigned the

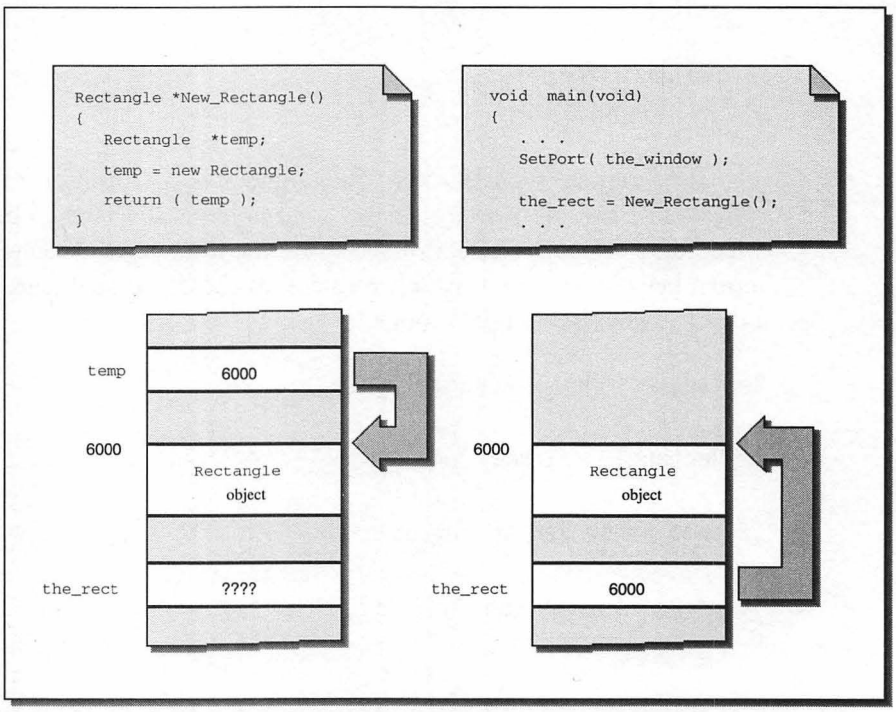


FIGURE 8–4 Creating an object via a function

value that temp had—the address of the block of memory that holds the Rectangle object. Variable temp isn't shown on the right side of the figure, because at this point the program is back in main(), and the local variable temp is no longer accessible.

You may wonder about the purpose of defining and calling a function whose only effect is to replace the single line of code that uses the new operator. In this situation, there is no advantage. When a program uses several classes that are derived from the same abstract class, however, the advantages of this technique become readily apparent—as you'll see later in this chapter.

Function Prototypes and Forward References

Function prototypes are listed at the start of a source code listing so that the compiler knows the properties of each function it will encounter. The function properties are its name, return type, and arguments. Here is the function prototype for the New_Rectangle() function, followed by the function itself:

```
Rectangle  *New_Rectangle( void );        // prototype

Rectangle  *New_Rectangle( void )         // function
{
   Rectangle  *temp;

   temp = new Rectangle;

   return ( temp );
}
```

A function that has C or C++ data types as arguments and a C or C++ data type as the return type requires only a single-line prototype. But for a function that has either an argument or a return type that isn't a C or C++ keyword, you'll have to add an additional statement. The New_Rectangle() function is just such as routine. Its return type is Rectangle—a type that isn't a keyword, and thus a type the compiler is not familiar with. Remember, function prototypes are traditionally listed at the top of the source code—before variable declarations and class definitions. When the compiler encounters the New_Rectangle() prototype, it hasn't yet seen the Rectangle class definition. Figure 8–5 shows the error messages the Symantec compiler gave me when I attempted to compile a program with the New_Rectangle() prototype. The error messages you would see might vary depending on the version of Symantec C++ you're using.

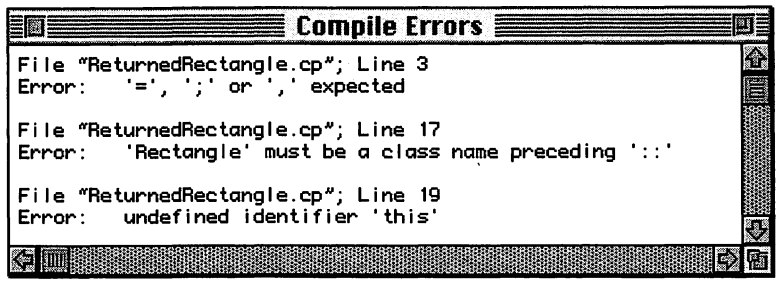


FIGURE 8–5 Error message when a class name is used in a function prototype

The first error in the Compile Errors window in Figure 8–5 is referring to the prototype. Because the compiler is completely baffled by the word *Rectangle,* it doesn't know what to make of the prototype statement. In fact, it doesn't even recognize it as a statement. To overcome this dilemma you can use the class keyword in a *forward reference.* A forward reference is used to name a class before defining it. In a sense, you're prototyping the class. That is, you're making the class known to the compiler before defining it—just as a function prototype makes a function known to the compiler before it is defined. Now, using the class keyword, let's take a look at the order of things in the source code:

```
class    Rectangle;                    // forward reference

Rectangle  *New_Rectangle( void );  // function prototype

// Rectangle class definition here

// New_Rectangle() function definition here
```

The forward reference informs the compiler that Rectangle is a class. From that point on, the compiler will accept statements—including function prototypes—that make use of Rectangle. You'll see examples of forward references in some of the example programs later in this chapter.

A Returned Object Example

As is normally the practice in this book, I'll end the section with the source code for a brief example that demonstrates the main concept discussed in

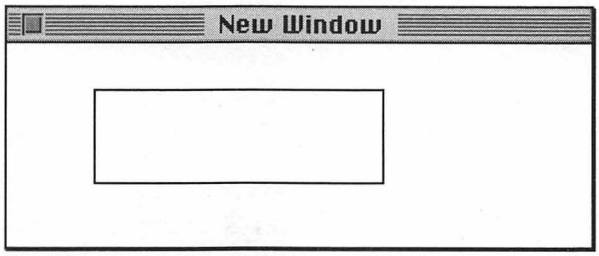


FIGURE 8–6 The output of the ReturnedRectangle program

the section. The program ReturnedRectangle does just that; it returns a rectangle to the calling function. You'll see the output of the program in Figure 8–6. Notice that the results of running ReturnedRectangle are the same as those produced by the previous example, RectangleClass. The only difference between the two programs is how the Rectangle variable `the_rect` is assigned its value. In RectangleClass, `the_rect` receives its value directly using the new operator:

```
the_rect = new Rectangle;
```

In ReturnedRectangle, `the_rect` receives its value indirectly through a call to a function:

```
the_rect = New_Rectangle();
```

In either case, once `the_rect` has a value—that is, once it points to memory allocated for a Rectangle object—its member functions can be accessed.

```
// ***************  ReturnedRectangle.cp  ***************

class    Rectangle;   // at this point the compiler hasn't
                      // seen the Rectangle class definition
                      // - use as a forward reference here

Rectangle  *New_Rectangle( void );   // function prototype
```

```
class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}


void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


Rectangle  *New_Rectangle( void )
{
   Rectangle  *temp;

   temp = new Rectangle;

   return ( temp );
}


Rectangle  *the_rect;   // declare object of type Rectangle


void  main( void )
```

```
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   the_rect = New_Rectangle();

   the_rect->Set_Rectangle( 50, 25, 200, 75 );
   the_rect->Draw_Rectangle();

   while ( !Button() )
      ;
}
```

Returned Objects and Derived Classes

An object that is of a derived class can be created and returned by a function just as an object of a base class can be. So why devote a whole section of this chapter to the topic? Because when objects are created in this way, dynamic binding occurs—and dynamic binding is the topic that all the preceding pages have been leading to.

Rectangles and Derived Classes

DerivedRectangles is a program that will be used in the next section when dynamic binding is finally discussed. DerivedRectangles defines two classes derived from the Rectangle class—FatRect and FancyRect. Each of these

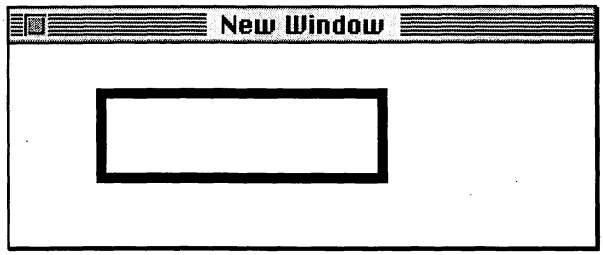


FIGURE 8–7 Result of drawing an object of the FatRect class

derived classes consists of a single member function—Draw_Rectangle()—which it inherits from the Rectangle class and then overrides:

```
class  FatRect : public  Rectangle        // derived class
{
   public:
      void  Draw_Rectangle( void );
};


class  FancyRect : public  Rectangle       // derived class
{
   public:
      void  Draw_Rectangle( void );
};
```

The FatRect version of Draw_Rectangle() changes the pen size and then frames the class data member rectangle aRect. The routine ends by returning the pen to its normal state—one pixel by one pixel. When a FatRect object sends a message to Draw_Rectangle(), the result is a rectangle like the one pictured in Figure 8–7.

```
void  FatRect :: Draw_Rectangle( void )
{
   PenSize( 5, 5 );
   FrameRect( &this->aRect );
   PenNormal();
}
```

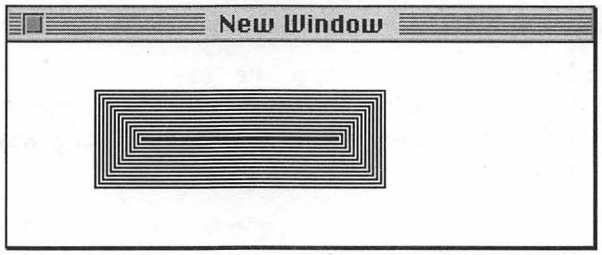


FIGURE 8–8 Result of drawing an object of the FancyRect class

The FancyRect version of Draw_Rectangle() is more involved than the FatRect version. This routine uses a loop to continually inset the object's rectangle, framing it each time. The number of times the loop runs is dependent on the height of the object's rectangle. Draw_Rectangle() first determines the height of the rectangle and saves that value in the local variable `height`. This number is divided by four to give the number of inset rectangles to draw. Because the rectangle will be inset repeatedly, a temporary rectangle is used. That way the size of the rectangle that is a part of the object won't be altered by the routine. Figure 8–8 shows what a rectangle with a height of 50 pixels would look like.

```
void  FancyRect :: Draw_Rectangle( void )
{
   short  i;              // loop counter
   short  height;         // height of the rectangle
   short  num_rects;      // number of inset rects to draw
   Rect   temp;           // work with a temporary rectangle

   height = this->aRect.bottom - this->aRect.top;
   num_rects = ( height/4 );

   temp = this->aRect;

   for ( i = 0; i <= num_rects; i++ )
   {
      FrameRect( &temp );
      InsetRect( &temp, 2, 2 );
   }
}
```

NOTE

Note that aside from the use of the *this* keyword, the body of the Draw_Rectangle() function looks like a Macintosh C program—a reminder that your knowledge of C will go a long way in your C++ programming endeavors.

DerivedRectangle creates a FancyRect object and draws it to the window using these lines:

```
the_rect = new FancyRect;

the_rect->Set_Rectangle( 50, 25, 200, 75 );
the_rect->Draw_Rectangle();
```

If you want to see a FatRect object, you'll have to modify the program or continue reading this chapter.

```
// ***************  DerivedRectangle.cp  ***************

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


class  FatRect : public  Rectangle       // derived class
{
   public:
      void  Draw_Rectangle( void );
};
```

```
class  FancyRect : public  Rectangle     // derived class
{
   public:
      void  Draw_Rectangle( void );
};


void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}


void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}



void  FatRect :: Draw_Rectangle( void )
{
   PenSize( 5, 5 );
   FrameRect( &this->aRect );
   PenNormal();
}


void  FancyRect :: Draw_Rectangle( void )
{
   short  i;             // loop counter
   short  height;        // height of the rectangle
   short  num_rects;     // number of inset rects to draw
   Rect   temp;          // work with a temporary rectangle

   height = this->aRect.bottom - this->aRect.top;
   num_rects = ( height/4 );
```

```
    temp = this->aRect;

    for ( i = 0; i <= num_rects; i++ )
    {
        FrameRect( &temp );
        InsetRect( &temp, 2, 2 );
    }
}


FancyRect  *the_rect;   // declare object of type FancyRect


void  main( void )
{
    WindowPtr   the_window;
    Rect        window_rect;

    InitGraf( &thePort );
    InitFonts();
    InitWindows();

    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                             noGrowDocProc, (WindowPtr)-1L,
                             true, 0 );
    SetPort( the_window );

    the_rect = new FancyRect;

    the_rect->Set_Rectangle( 50, 25, 200, 75 );
    the_rect->Draw_Rectangle();

    while ( !Button() )
        ;
}
```

Dynamic Binding

The DerivedRectangle program has a serious drawback in that there is no decision-making code that provides for a choice of which type of object should be created. If my C++ programs are to evolve into true Macintosh applications, they must have a Macintosh interface with menus and dialog boxes, and they must give the user more control over what happens. All of these faults will be remedied in the remainder of this book, but I'll begin to improve on the decision-making abilities of my programs right now.

DerivedRectangle defines a single base class, Rectangle, and two derived classes, FatRect and FancyRect. What I would ideally like to include in my programs is the ability to let the user choose from which of the two classes an object should be created—something like this:

```
if ( choice == 1 )
   temp = new  FatRect;
else
   temp = new  FancyRect;
```

The problem—or what appears to be the problem—with the above code is that the variable `temp` is being used with two different classes. The question is, should `temp` be declared as a pointer to a FatRect or a pointer to a FancyRect? The answer is, neither! Variable `temp` should be declared to be a pointer to a Rectangle—the base class. Examine the following New_Rectangle() function; then read on to discover how the function works.

```
Rectangle  *New_Rectangle( short choice )
{
   Rectangle  *temp;

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;

   return ( temp );
}
```

Object-oriented programming allows a programmer to assign an object pointer of a derived class to an object pointer of its base class. In the above example, `temp`—declared to be a Rectangle pointer—could, of course,

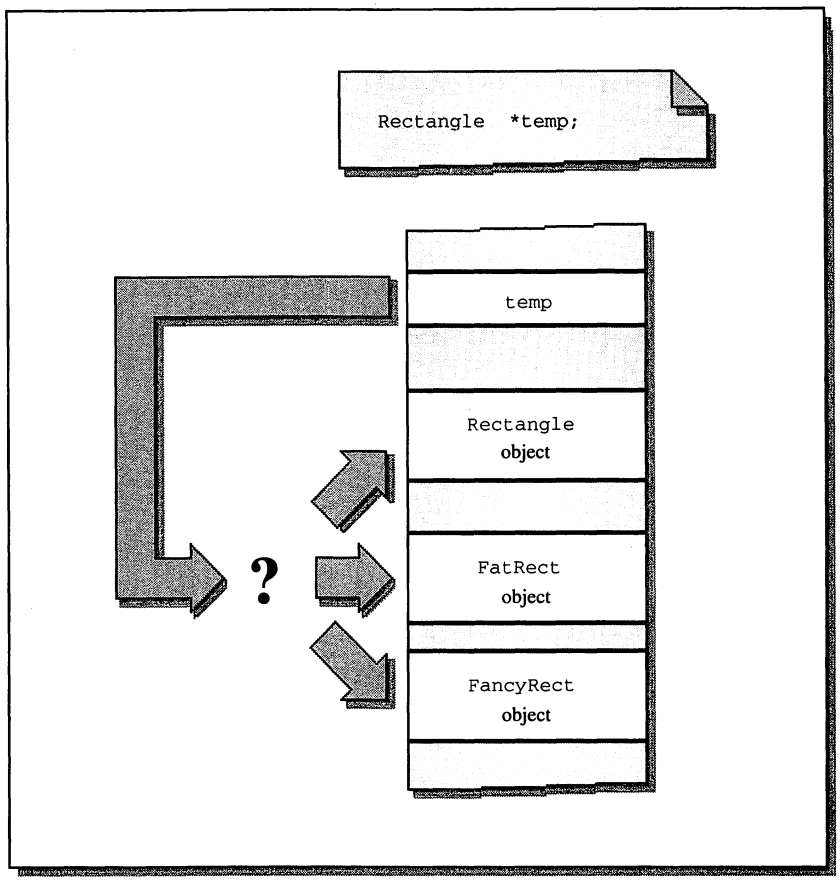


FIGURE 8–9 A variable that is a pointer to a base class may point to a base class or a derived class object.

point to a Rectangle object. Because I'm using Rectangle as an abstract class, I'll have `temp` point to one of the derived class objects instead. Knowing that I can have a base class pointer point to a derived class object, I can use the same `temp` variable on the receiving end of the two statements that create new objects.

Figure 8–9 shows that if I had a base class object and two derived class objects in memory, the rules of object-oriented programming would allow me to have `temp` point to any one of the three objects.

Figure 8–9 shows the general concept that a base class pointer variable is allowed to point to an object of the base class or an object of any class

derived from the base class. Figure 8–10 highlights my particular case. The `temp` variable will point to an object of one of two types—either a FatRect object or a FancyRect object. Figure 8–10 shows that `temp` will point to a location in memory—it just isn't clear at this time what *type* of object will be at that location.

The New_Rectangle() function will set a Rectangle pointer to point to either a FatRect or FancyRect object. It will then return that pointer to the line of code that invoked the function. Here's a look at New_Rectangle(), and a call to it:

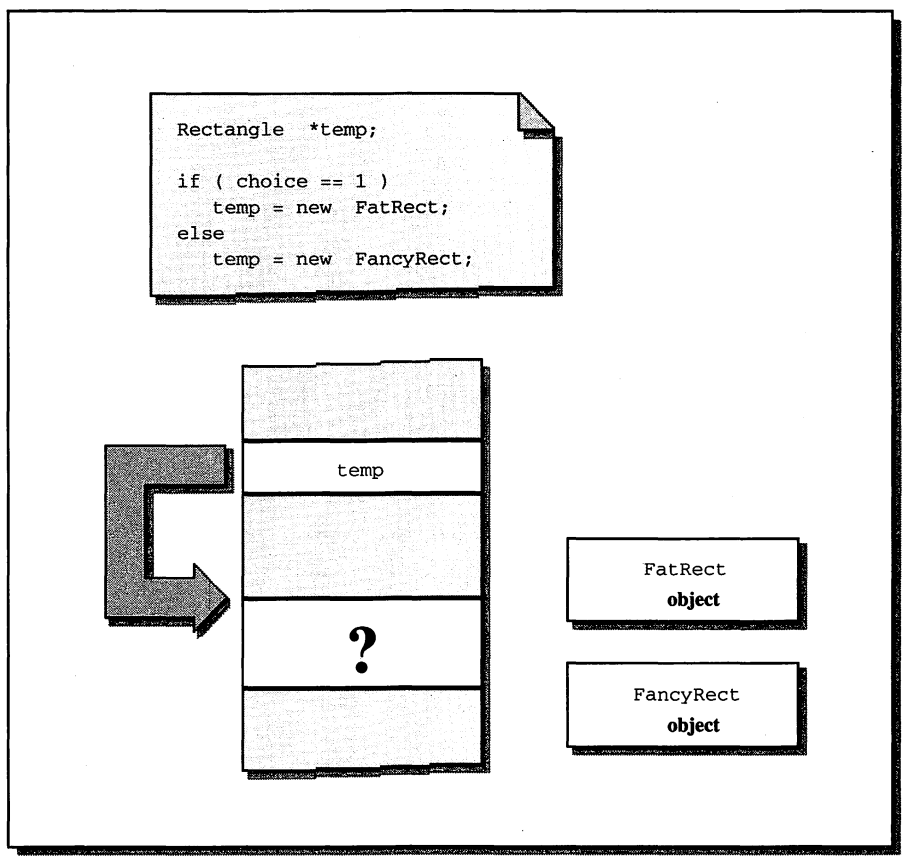


FIGURE 8–10 Until the code is executed, it is unclear which type of object temp will point to.

```
Rectangle  *New_Rectangle( short choice )
{
   Rectangle  *temp;

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;

   return ( temp );
}

Rectangle  *the_rect;  // declare object of type Rectangle

the_rect = New_Rectangle( 2 );
```

After the above code executes, the_rect will be pointing to a FancyRect object. Because I passed a value of 2 to New_Rectangle(), the else portion of the if-else executed, creating a new FancyRect object.

The variable the_rect is declared to be a pointer to a Rectangle object. Yet it points to a FancyRect object. While the type of variable the_rect is a Rectangle pointer, the class of the object that the_rect points to is FancyRect. Again, this is permissible in C++.

This situation leads to one very interesting question: When a program that includes the above code executes, and a Draw_Rectangle message is sent, how does the program know which Draw_Rectangle() routine to execute? Variable the_rect points to a FancyRect object, yet the_rect was declared to be of type Rectangle. Remember, both object types—Rectangle and FancyRect—have a Draw_Rectangle() member function. The answer to the question is that the FancyRect Draw_Rectangle() function will execute.

When code is compiled, certain things may be unclear. Consider this situation:

```
Rectangle  *the_rect;   // declare object of type Rectangle
short       rect_type;  // type of rect: FatRect or
                        // FancyRect

// prompt the user to enter a value for rect_type
```

```
the_rect = New_Rectangle( rect_type );

the_rect->Set_Rectangle( 50, 25, 200, 75 );
the_rect->Draw_Rectangle();
```

In the above code, the type of object that will be returned by New_Rectangle() isn't known, and can't be known, until runtime. The object's type won't be known until the user chooses the type of object—after the code is compiled and executed. Yet the code goes on to send the the_rect object a Draw_Rectangle message. When the compiler reaches this section of code, how does it mark, or indicate in any way, which Draw_Rectangle() function is to execute? The answer is, it doesn't. That decision is made each time the compiled program, the executable, runs. The determination of which class an object belongs to is reserved for the actual running of the application. There's a name for this object-oriented feature, and, if you haven't already guessed, the name is *dynamic binding.*

The ability to send the same message to objects of different classes is called *polymorphism.* The implementation of polymorphism is done through dynamic binding. Dynamic binding involves the building of tables—lookup tables that are used by the program—by the compiler. Thankfully, the details of these tables remain transparent to the program user and to the programmer who writes the program. You won't ever have to concern yourself with *how* dynamic binding works—just that it *does* work.

Dynamic binding is a very powerful—and sometimes difficult to grasp—programming concept. Figure 8–11 serves to further clarify this important feature of object-oriented programming. It shows that when a program sends a message to an object, that message is used by the lookup table to determine which of the functions with that name should be executed. As promised, I've spared you the details of precisely what takes place in the lookup table and memory.

A Dynamic Binding Example

In the pages of this chapter, you have been presented with all the code you need to write a short program that uses dynamic binding—you just haven't seen it neatly packaged. The program that appears at the end of this section does just that.

This section's program includes the Rectangle abstract class and the FatRect and FancyRect derived classes with which you're already familiar. The program uses the New_Rectangle() function to return either a FatRect object pointer or a FancyRect object pointer—in the form of a Rectangle pointer. Here's one more look at that important function:

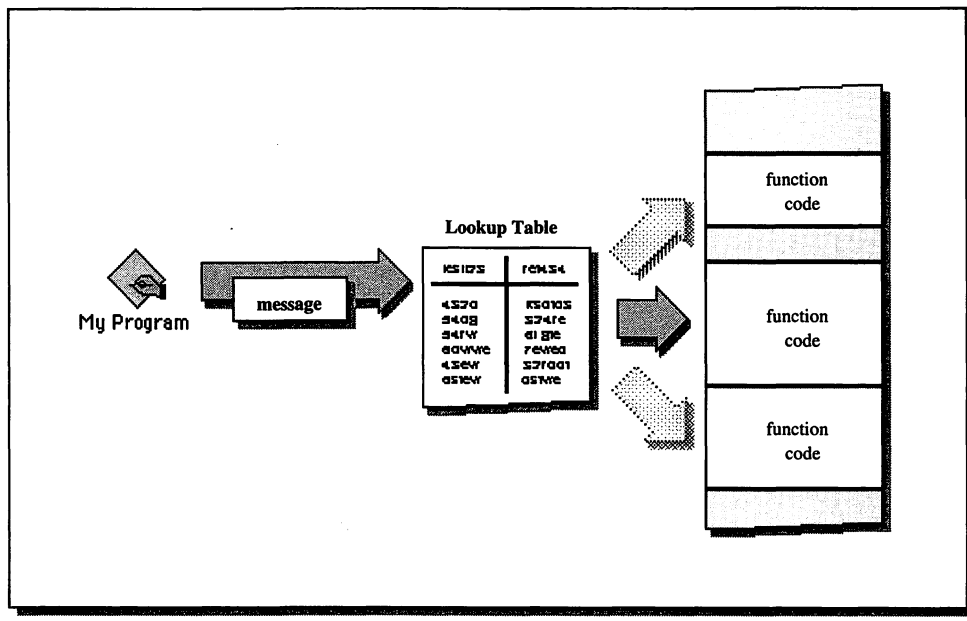


FIGURE 8–11 When an application is running, a lookup table is used to determine which function to execute.

```
Rectangle  *New_Rectangle( short choice )
{
   Rectangle  *temp;

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;

   return ( temp );
}
```

The program passes a value of 2 to New_Rectangle() to cause that routine to return a pointer to an object of the FancyRect class. You can change this parameter value to a 1 if you want to get a pointer to a FatRect object instead:

```
the_rect = New_Rectangle( 1 );
```

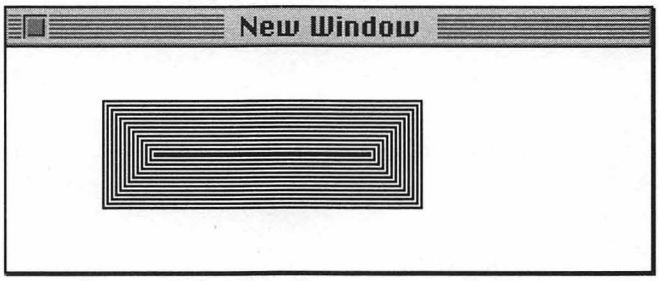


FIGURE 8–12 The output of the ReturnDerivedRect program

Regardless of the type of object created, the program goes on to send two messages to the object:

```
the_rect->Set_Rectangle( 50, 25, 200, 75 );
the_rect->Draw_Rectangle();
```

Variable `the_rect`, declared to be a pointer to a Rectangle, will execute the proper Draw_Rectangle() routine—regardless of the class of the object that `the_rect` ends up pointing to. That's dynamic binding in action.

The program that appears next is named ReturnDerivedRect. If you run it as is, you'll see a window like the one shown in Figure 8–12. If you change the parameter to New_Rectangle() to a 1, you'll see the window pictured in Figure 8–13.

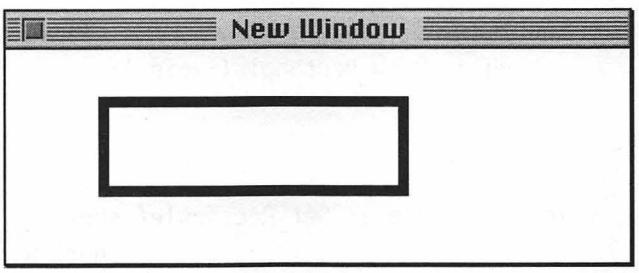


FIGURE 8–13 The output of ReturnDerivedRect if the parameter to New_Rectangle is changed to a value of 1

```
// *****************  ReturnDerivedRect.cp  **************

class  Rectangle;                        // forward reference

Rectangle  *New_Rectangle( short );   // function prototype


class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


class  FatRect : public  Rectangle       // derived class
{
   public:
      void  Draw_Rectangle( void );
};


class  FancyRect : public  Rectangle     // derived class
{
   public:
      void  Draw_Rectangle( void );
};


void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}
```

```
void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


void  FatRect :: Draw_Rectangle( void )
{
   PenSize( 5, 5 );
   FrameRect( &this->aRect );
   PenNormal();
}


void  FancyRect :: Draw_Rectangle( void )
{
   short  i;              // loop counter
   short  height;         // height of the rectangle
   short  num_rects;      // number of inset rects to draw
   Rect   temp;           // work with a temporary rectangle

   height = this->aRect.bottom - this->aRect.top;
   num_rects = ( height/4 );

   temp = this->aRect;

   for ( i = 0; i <= num_rects; i++ )
   {
      FrameRect( &temp );
      InsetRect( &temp, 2, 2 );
   }
}


Rectangle  *New_Rectangle( short choice )
{
   Rectangle  *temp;
```

```
    if ( choice == 1 )
       temp = new  FatRect;
    else
       temp = new  FancyRect;

    return ( temp );
}


Rectangle  *the_rect;   // declare object of type Rectangle


void  main( void )
{
    WindowPtr   the_window;
    Rect        window_rect;

    InitGraf( &thePort );
    InitFonts();
    InitWindows();

    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    the_rect = New_Rectangle( 2 );

    the_rect->Set_Rectangle( 50, 25, 200, 75 );
    the_rect->Draw_Rectangle();

    while ( !Button() )
       ;
}
```

The Rectangle Class—Still an Abstract Class?

Chapter 7 defined an abstract class as a class that serves as a common ancestor of a family of derived classes. It was further defined as a class from which no objects are defined. Instead, objects are defined from the various classes that are derived from the abstract class. The previous example, for instance, used the Rectangle class as an abstract class. In that program, objects are always of either the FatRect derived class or the FancyRect derived class. Notice that in the previous example, however, I declared a variable to point to the Rectangle base class:

```
Rectangle  *the_rect;  // declare object of type Rectangle
```

So, is the Rectangle class still considered an abstract class? Yes. Although a pointer to a Rectangle variable is declared, it never actually points to a Rectangle object. You won't find the new operator used with the Rectangle class anywhere in the source code of the previous program. Instead, `the_rect` is always assigned to point to one of the two classes derived from the Rectangle abstract class.

Passing Objects to Functions

This chapter has demonstrated that objects can be returned by functions—just as other data types such as shorts and floats are returned. The New_Rectangle() function developed in this chapter returns an object (or, more accurately, a pointer to an object). This may have led you to wonder if objects can also be used as parameters to functions. From the title of this section, you've probably figured out that objects can indeed be used as parameters.

A Rectangle Object as a Parameter

An object can be used as a parameter to a function—but not to just any function. Recall that, when working with objects, the general access strategy is to declare class data members to be protected. That means that only objects of that class—or of a derived class—can access said members. Thus functions that aren't member functions of a class are unable to access or manipulate objects. Since you're familiar with shapes as objects, I'll elaborate by using an example that includes the Rectangle class.

First, let's see what we *can't* do. If I want a function to do something with a Rectangle object (whether an object of the base class or a derived

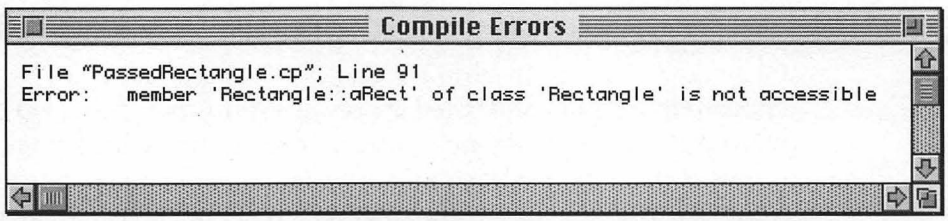


FIGURE 8–14 The error message reported when an object is passed to a function

class), I can't just write a function and pass it an object or a pointer to an object. For instance, let's say I want to write a function that changes the size of a rectangle. The Change_Size() routine I wrote to do this task didn't work; it gave me the error message shown in Figure 8–14:

```
void  Change_Size( Rectangle *from_rect )
{
   short  L, T, R, B;

   L = from_rect->aRect.left;

   // rest of function here
}
```

What doesn't the compiler like about my Change_Size() function? The function tries to work with the Rectangle class data member `aRect`, which is declared to be protected:

```
class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short, short );
      virtual void  Draw_Rectangle( void );
};
```

Only member functions of the Rectangle class or a class derived from the Rectangle class are aware of the aRect member. When Change_Size() attempts to look at one of the four coordinates of the Rect data structure, the compiler complains that Change_Size() has no right to do so. While at times like this data hiding may appear to get in my way, I'll be sure to keep in mind that encapsulation is one of the *strengths* of object-oriented programming.

So, how *does* one pass an object to a function in order to access the object's data? By making the function a part of the object. In the code below I've modified the Rectangle class to include Change_Size() as one of its member functions:

```
class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
      virtual void  Change_Size( Rectangle * );
};
```

Next, I wrote the Change_Size() function—being sure to use the scope resolution operator along with the Rectangle class name:

```
void  Rectangle :: Change_Size( Rectangle *from_rect )
{
   short  L, T, R, B;

   L = from_rect->aRect.left;
   T = from_rect->aRect.top;
   R = from_rect->aRect.right;
   B = from_rect->aRect.bottom;

   this->Set_Rectangle( L, T, R, B );
}
```

The Change_Size() function is invoked by a Rectangle object. When Change_Size() is called, a pointer to a different Rectangle object is passed.

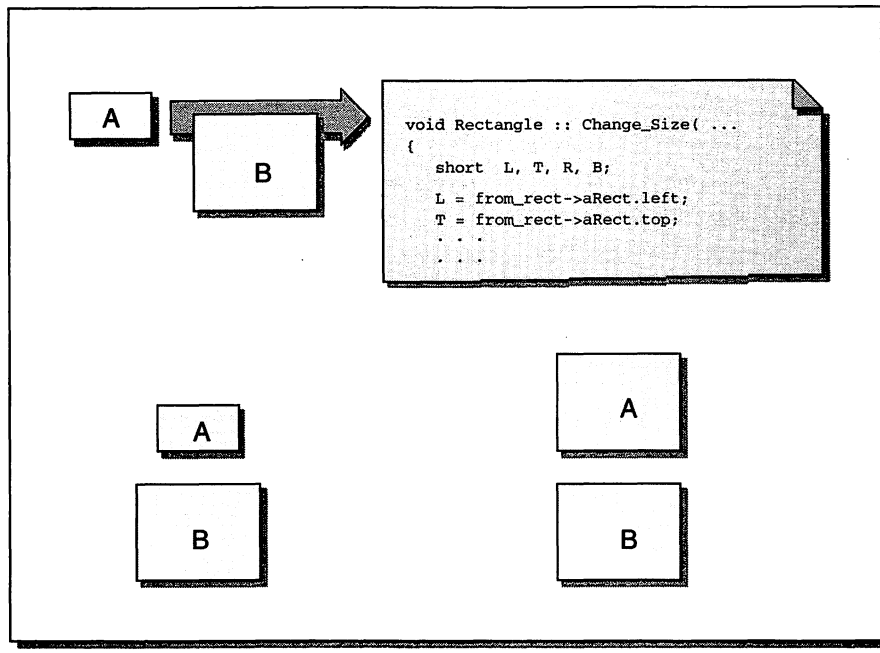


FIGURE 8–15 The Change_Size() function changes the size of the aRect data member of an object.

The function accesses the aRect member of the passed-in object to get each of its four coordinates. Then it sets the aRect member of the object that received the message to these new coordinates. Figure 8–15 illustrates what the function is doing. On the left side of the figure, you can see that Rectangle object A is calling Change_Size() and passing the larger Rectangle object B. At the bottom left of the figure are the two rectangles before the call. After the call, object A is the same size as object B—as shown in the bottom right portion of Figure 8–15.

To make use of the Change_Size() function I'll have to declare two objects:

```
Rectangle  *the_rect;        // declare object of type
                             // Rectangle
Rectangle  *another_rect;    // declare object of type
                             // Rectangle
```

I then allocate memory for the first object and set its size. This will be the object that is passed to the Change_Size(). I'll send the object a draw message to see what it looks like. The results are shown in Figure 8–16.

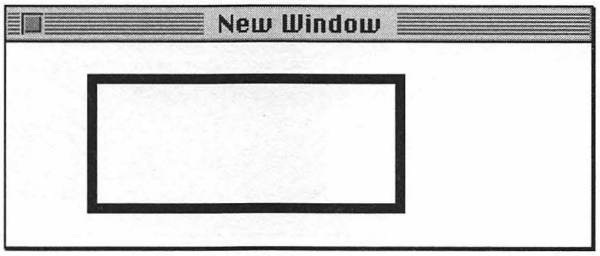


FIGURE 8–16 The result of drawing the first object—`the_rect`.

```
the_rect = new FatRect;
the_rect->Set_Rectangle( 40, 20, 200, 90 );
the_rect->Draw_Rectangle();
```

Next I allocate memory for the second object and set its size. I've made this object a FancyRect, and I've positioned it in the upper left corner of the window. This object will be the one that invokes Change_Size(). Again, I send the new object a draw message to see what's going on. The results are shown in Figure 8–17.

```
another_rect = new FancyRect;
another_rect->Set_Rectangle( 10, 10, 35, 35 );
another_rect->Draw_Rectangle();
```

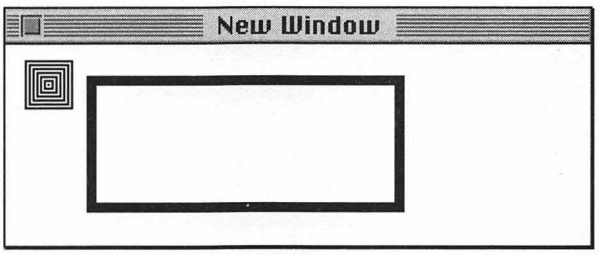


FIGURE 8–17 The result of drawing the second object—`another_rect`

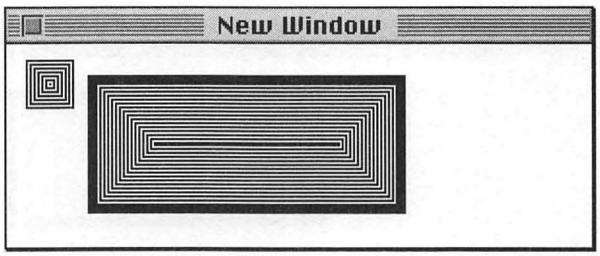


FIGURE 8–18 The result of drawing the another_rect—after its size has been changed

Now, to test the Change_Size() function, I'll invoke it through the FancyRect object, passing the FatRect object as the parameter. When Change_Size() has executed, the calling object, another_rect, should be the size of the passed object, the_rect. I'll send another_rect a draw message and, if all has gone well, another_rect should draw right over the rectangle that the_rect drew. Figure 8–18 shows that that's exactly what happened.

```
another_rect->Change_Size( the_rect );
another_rect->Draw_Rectangle();
```

The PassedRectangle Example Program

PassedRectangle is a program that brings together the code used in this section. You'll find the source code and project for PassedRectangle on the accompanying disk.

If you'd like to see exactly how PassedRectangle works, select Use Debugger from the THINK Project Manager's Project menu before running the program. Step through the code to see each of the three rectangles drawn to the program's one window.

```
// ***************  PassedRectangle.cp  ***************

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
```

```
      virtual void  Draw_Rectangle( void );
      virtual void  Change_Size( Rectangle * );
};


class  FatRect : public  Rectangle       // derived class
{
   public:
      void  Draw_Rectangle( void );
};


class  FancyRect : public  Rectangle     // derived class
{
   public:
      void  Draw_Rectangle( void );
};


void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}


void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


void  Rectangle :: Change_Size( Rectangle *from_rect )
{
   short  L, T, R, B;

   L = from_rect->aRect.left;
   T = from_rect->aRect.top;
   R = from_rect->aRect.right;
```

```
    B = from_rect->aRect.bottom;

    this->Set_Rectangle( L, T, R, B );
}


void  FatRect :: Draw_Rectangle( void )
{
    PenSize( 5, 5 );
    FrameRect( &this->aRect );
    PenNormal();
}


void  FancyRect :: Draw_Rectangle( void )
{
    short  i;              // loop counter
    short  height;         // height of the rectangle
    short  num_rects;      // number of inset rects to draw
    Rect   temp;

    height = this->aRect.bottom - this->aRect.top;
    num_rects = ( height/4 );

    temp = this->aRect;

    for ( i = 0; i <= num_rects; i++ )
    {
        FrameRect( &temp );
        InsetRect( &temp, 2, 2 );
    }
}


Rectangle  *the_rect;         // declare object of type
                              // Rectangle
Rectangle  *another_rect;     // declare object of type
                              // Rectangle
```

```
void  main( void )
{
   WindowPtr   the_window;
   Rect        window_rect;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();

   SetRect( &window_rect, 50, 50, 350, 150 );
   the_window = NewWindow( 0L, &window_rect,
                           "\pNew Window", true,
                           noGrowDocProc, (WindowPtr)-1L,
                           true, 0 );
   SetPort( the_window );

   the_rect = new FatRect;
   the_rect->Set_Rectangle( 40, 20, 200, 90 );
   the_rect->Draw_Rectangle();

   another_rect = new FancyRect;
   another_rect->Set_Rectangle( 10, 10, 35, 35 );
   another_rect->Draw_Rectangle();

   another_rect->Change_Size( the_rect );
   another_rect->Draw_Rectangle();

   while ( !Button() )
      ;
}
```

Returned Objects and the Animal Class

So as not to leave the pet shop owner in the dark about the exciting programming technique called dynamic binding, I've updated his Animal

database to include a function that returns an Animal pointer that points to one of the two Animal derived classes—GoodPet or BadPet. I stole the New_Rectangle() routine and modified it to work with the Animal class. Here's a look at New_Animal():

```
Animal  *New_Animal( short  choice )
{
   Animal  *temp;

   if ( choice == 1 )
      temp = new  GoodPet;
   else
      temp = new  BadPet;

   return ( temp );
}
```

Because the compiler won't know what an Animal is at the very top of the source code, I use a forward reference before writing the New_Animal() prototype:

```
class  Animal;                          // forward reference

Animal  *New_Animal( short );           // function prototype
```

Creating a new GoodPet object or a BadPet object is done using the same technique as was used in the rectangle examples: declare a pointer to an object of the base class; then assign that pointer to point to an object of either of the derived classes by invoking a function. Here's how that's done:

```
Animal  *the_pet;       // declare object of type Animal

the_pet = New_Animal( 1 );
```

Once the_pet is pointing to an object, I can send the object messages. Since I'm passing to New_Animal() a value that I know will create a GoodPet object, I'll include appropriate parameters for a popular pet:

```
the_pet->Set_Type( "\pTropical fish" );
the_pet->Set_Desc( "\pContainable!" );
the_pet->Set_Cost( 20 );
the_pet->Write_Info();
```

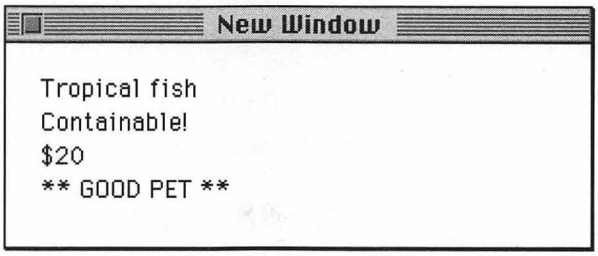


FIGURE 8–19 The output of the ReturnDerivedPet program

All of the above code appears in the program that follows—ReturnDerivedPet. When you run ReturnDerivedPet, you'll see a window like the one shown in Figure 8–19.

If you want to create a BadPet object, simply pass New_Animal() a value other than 1. You'll also want to include message parameters for less desirable pets. Here's an example:

```
the_pet = New_Animal( 2 );

the_pet->Set_Type( "\pStriped skunk" );
the_pet->Set_Desc( "\pDistinct odor" );
the_pet->Set_Cost( 75 );
the_pet->Write_Info();
```

If you use the above code in ReturnDerivedPet, you'll see a window like the one pictured in Figure 8–20.

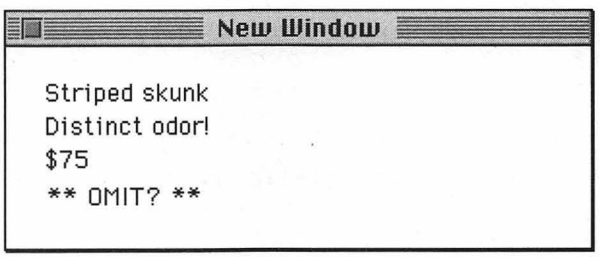


FIGURE 8–20 The output of ReturnDerivedPet when code to create a BadPet object is used

```
// ***************  ReturnDerivedPet.cp  ***************

class  Animal;                          // forward reference

Animal  *New_Animal( short );           // function prototype

void  Fill_Str255( Str255, Str255 );


class  Animal                           // base class
{
   protected:
      Str255  type;
      Str255  desc;
      long    cost;

   public:
      virtual void  Set_Type( Str255 );
      virtual void  Get_Type( Str255 );
      virtual void  Set_Desc( Str255 );
      virtual void  Get_Desc( Str255 );
      virtual void  Set_Cost( long );
      virtual long  Get_Cost( void );
      virtual void  Write_Info( void );
};


class  BadPet : public  Animal          // derived class
{
   public:
      void  Write_Info( void );
};


class  GoodPet : public  Animal         // derived class
{
   public:
      void  Write_Info( void );
};
```

```
void  Animal :: Set_Type( Str255 name )
{
   Fill_Str255( this->type, name );
}


void  Animal :: Get_Type( Str255 name )
{
   Fill_Str255( name, this->type );
}


void  Animal :: Set_Desc( Str255 desc )
{
   Fill_Str255( this->desc, desc );
}


void  Animal :: Get_Desc( Str255 comment )
{
   Fill_Str255( comment, this->desc );
}



void  Animal :: Set_Cost( long price )
{
  this->cost = price;
}



long  Animal :: Get_Cost( void )
{
  return ( this->cost );
}
```

```
  void  Animal :: Write_Info( void )
{
   Str255  str;

   MoveTo( 20, 30 );
   DrawString( this->type );
   MoveTo( 20, 45 );
   DrawString( this->desc );
   NumToString( this->cost, str );
   MoveTo( 20, 60 );
   DrawString( "\p$");
   DrawString( str );
}

  void  BadPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** OMIT? **");
}

void  GoodPet :: Write_Info( void )
{
   Animal :: Write_Info();
   MoveTo( 20, 75 );
   DrawString( "\p** GOOD PET **");
}


Animal  *New_Animal( short  choice )
{
   Animal  *temp;

   if ( choice == 1 )
      temp = new  GoodPet;
   else
      temp = new  BadPet;
```

```
    return ( temp );
}

Animal  *the_pet;          // declare object of type Animal

void  main( void )
{
    WindowPtr   the_window;
    Rect        window_rect;

    InitGraf( &thePort );
    InitFonts();
    InitWindows();

    SetRect( &window_rect, 50, 50, 350, 150 );
    the_window = NewWindow( 0L, &window_rect,
                            "\pNew Window", true,
                            noGrowDocProc, (WindowPtr)-1L,
                            true, 0 );
    SetPort( the_window );

    the_pet = New_Animal( 1 );

    the_pet->Set_Type( "\pTropical fish" );
    the_pet->Set_Desc( "\pContainable!" );
    the_pet->Set_Cost( 20 );
    the_pet->Write_Info();

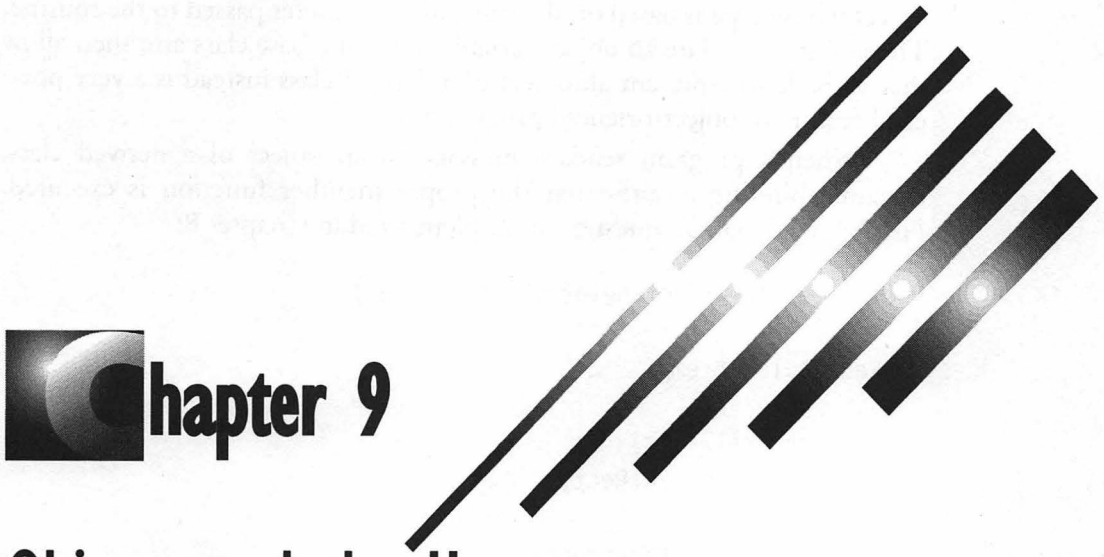
    while ( !Button() )
        ;
}
```

Chapter Summary

A base class may have any number of derived classes, each with its own set of member functions that override those defined in the base class. When that

happens, a program may have several objects of different class types, each calling like-named member functions. While these functions have the same name, they represent different code. Additionally, an object pointer can be declared to be a base class pointer, yet end up pointing to a derived object once the program is running. To resolve these issues, an object-oriented program uses dynamic binding. An object pointer declared to be a base class pointer is not bound to that base class until after the program is compiled and running. Only then, when the object that the pointer will point to is created, does the pointer become bound to either the base class or one of the classes derived from the base class.

Objects act as other standard C or C++ variables in that they can be passed to functions and returned from functions. This applies to normal functions as well as class member functions.



Chapter 9

Objects and the User

Windows are one of the most important elements of a graphical user interface; they're the vehicle for displaying information to the user. A program's text and graphics are always drawn to a window, as all of this book's example programs have demonstrated. While windows are great for output, they don't allow input—for that, you must use alerts and dialog boxes.

In this chapter, you'll see how to use an alert to let the user decide which type of object your program should create. The alert, however, will be just a stepping stone to the dialog box. Dialog boxes put users in complete control by letting them specify not only the type of object to create but all of the individual features of each object.

Using an Alert to Create a New Object

The source code examples in Chapter 8 show how your programs can work with objects without knowing which type of objects they are. In particular, the New_Rectangle() function developed in the previous chapter creates an

object whose type is based on the value of a parameter passed to the routine. The ability to declare an object variable to be of a base class and then allow that variable to represent an object of a derived class instead is a very powerful feature of object-oriented programming.

When a program sends a message to an object of a derived class, dynamic binding ensures that the proper member function is executed. Here's how dynamic binding was implemented in Chapter 8:

```
Rectangle  *New_Rectangle( short choice )
{
   Rectangle  *temp;

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;

   return ( temp );
}

Rectangle  *the_rect;

the_rect = New_Rectangle( 2 );

the_rect->Set_Rectangle( 50, 25, 200, 75 );
the_rect->Draw_Rectangle();
```

Macintosh programs place the user in charge—that's what makes them user-friendly. The New_Rectangle() function attempts to be user-friendly by not limiting itself to the generation of a single type of object. While the function is a step in the right direction, a program that uses it will have certain limitations. There's still a lot of information hard-coded into the above code—information in the form of unvarying numbers rather than variables that can take on different values. Any truly useful program will give the user the opportunity to input most or all of the values. To do that, I'll have to start including more elements of the Macintosh user interface.

So far, I've included a window in each program so that I could view the program's output. Now, I'll have to start including a way for a user of my programs to enter information. Menus, alerts, and dialog boxes are three methods. Since the alert is the easiest to create and maintain, I'll start with it.

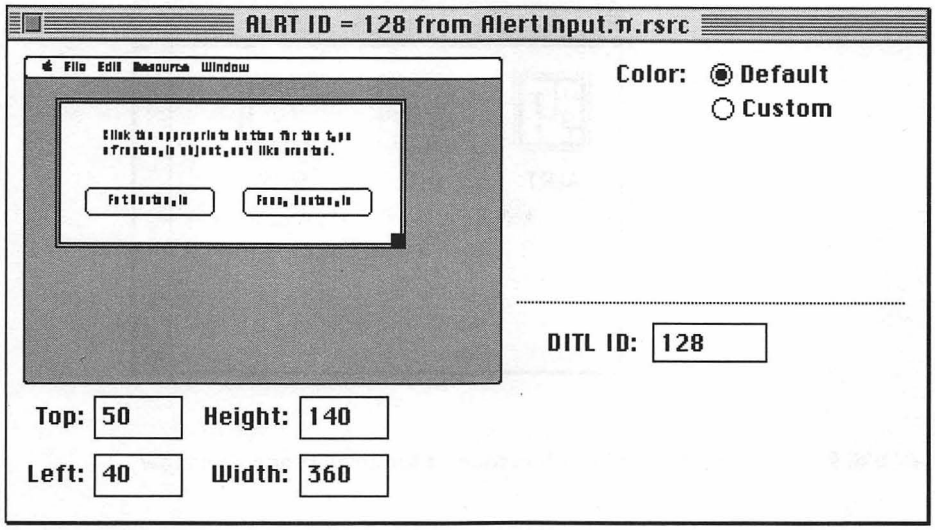


FIGURE 9–1 The ALRT resource for a program that uses an alert

Alert Resources

The disk that accompanies this book includes an example program named AlertInput. This is the first example that makes use of a resource file. The resource file—named AlertInput.π.rsrc—consists of three resources. Two of these resources—an ALRT and a DITL—will be used by the program to post an alert. Figure 9–1 shows how the ALRT resource looks in the resource editor ResEdit, while Figure 9–2 shows the DITL that is used by the ALRT.

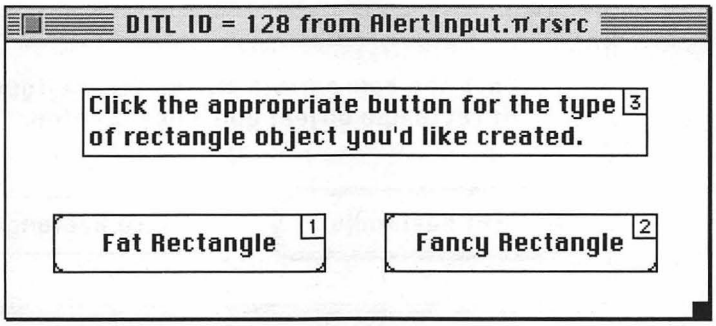


FIGURE 9–2 The DITL resource for a program that uses an alert

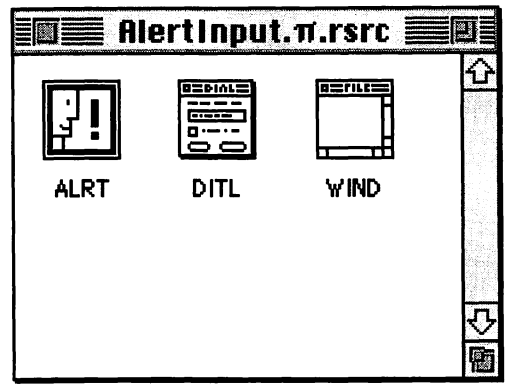


FIGURE 9–3 A resource file with resources for an alert and a window

The third resource in the file is a WIND. Since none of my previous examples used a resource file, I've used the routine NewWindow() to supply window-defining information to the Toolbox. Now that I'll be using a resource file, I'll define window attributes in a WIND resource and use the Toolbox function GetNewWindow() to load this information into memory. Figure 9–3 shows the resource types present in the AlertInput.π.rsrc file.

Using the Alert to Select an Object Type

Items 1 and 2 in the DITL of the alert are push buttons that will give the user the ability to choose the object type. Figure 9–4 shows what the user will see when running a program that posts the alert.

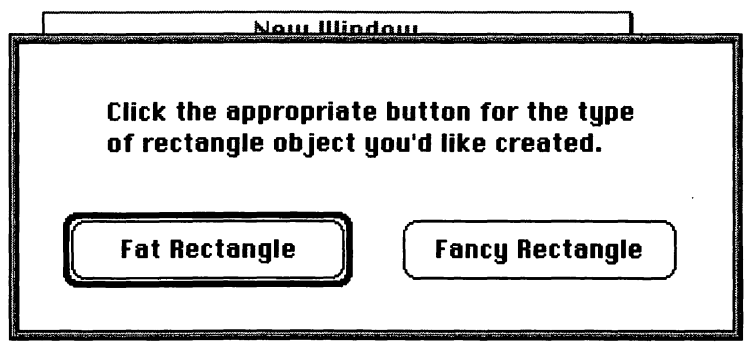


FIGURE 9–4 The alert that results from the ALRT and DITL resources

The Toolbox routine Alert() does all the work of posting an alert and monitoring the user's actions while the alert is on the screen. When the user dismisses the alert by clicking the mouse on one of the alert's buttons, Alert() returns the item number of the clicked-on item. I've added a call to Alert() to the New_Rectangle() function. Here's how that routine now looks:

```
Rectangle  *New_Rectangle( void )
{
   Rectangle  *temp;
   short       choice;

   choice = Alert( 128, nil );

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;

   return ( temp );
}
```

When the Alert() is dismissed, variable choice will hold the item number of the clicked-on button. New_Rectangle() uses that value in an if-else statement to determine which type of object to create. Using the alert eliminates the need to pass a parameter to New_Rectangle(). Now the routine is called as shown here:

```
Rectangle  *the_rect;

the_rect = New_Rectangle();

the_rect->Set_Rectangle( 50, 25, 200, 75 );
the_rect->Draw_Rectangle();
```

An Alert Example

AlertInput is a program very similar to the ReturnDerivedRect program found in Chapter 8. The main difference is in the way AlertInput uses an alert to let the user be in charge of selecting the type of object to create. The program also differs from ReturnDerivedRect in that it uses a WIND

resource to display a window. The last change is purely cosmetic; because the example programs are getting larger, I've added some fancy comments to separate the program into sections.

IMPORTANT

Don't forget about the Symantec resource file naming convention. A project's resource file must have the same name as the project, with .rsrc appended to it. Thus for my AlertInput.π project, the resource file is named AlertInput.π.rsrc. You can verify this by checking out the AlertInput example on the accompanying disk.

```
// ******************  AlertInput.cp  ******************

//_____________________________________________________________
//                                          forward references

class    Rectangle;


//_____________________________________________________________
//                                          function prototypes

Rectangle  *New_Rectangle( void );


//_____________________________________________________________
//                                          #define directives

#define     WIND_ID      128
#define     ALRT_ID      128


//_____________________________________________________________
//                                            global variables

Rectangle  *the_rect;
```

```
//_____________________________________________________
//                                        class definitions

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


class  FatRect : public  Rectangle
{
   public:
      void  Draw_Rectangle( void );
};


class  FancyRect : public  Rectangle
{
   public:
      void  Draw_Rectangle( void );
};


//_____________________________________________________
//                                 member function definitions

void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}
```

```
void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


void  FatRect :: Draw_Rectangle( void )
{
   PenSize( 5, 5 );
   FrameRect( &this->aRect );
   PenNormal();
}


void  FancyRect :: Draw_Rectangle( void )
{
   short  i;              // loop counter
   short  height;         // height of the rectangle
   short  num_rects;      // number of inset rects to draw
   Rect   temp;           // work with a temporary rectangle

   height = this->aRect.bottom - this->aRect.top;
   num_rects = ( height/4 );

   temp = this->aRect;

   for ( i = 0; i <= num_rects; i++ )
   {
      FrameRect( &temp );
      InsetRect( &temp, 2, 2 );
   }
}


//____________________________________________________________
//                                                      main()

void  main( void )
```

```
{
   WindowPtr   the_window;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitCursor();

   the_window = GetNewWindow( WIND_ID, nil, (WindowPtr)-1L );
   SetPort( the_window );

   the_rect = New_Rectangle();

   the_rect->Set_Rectangle( 50, 25, 200, 75 );
   the_rect->Draw_Rectangle();

   while ( !Button() )
      ;
}


//_____________________________________________________________
//                          open alert and create new object

Rectangle  *New_Rectangle( void )
{
   Rectangle  *temp;
   short       choice;

   choice = Alert( ALRT_ID, nil );

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;
     return ( temp );
}
```

Updating an Object

While objects can be defined to do nothing more than hold information, they usually also have the ability to output at least some information to a window. For example, a Rectangle object stores the coordinates of a rectangle and has a member function that draws that rectangle to a window. An Animal object holds a pet's name, cost, and suitability and has a member function that writes all of this information to a window.

When you write a program that make use of windows—as all of mine do—you should give the program the ability to update, or redraw, the contents of a window—whether you're using an object-oriented language or a procedural one. Up to this point I haven't been doing that. That's because, to keep the programs as simple as possible, I've kept windows "frozen" on the screen; the user could take no action to move or obscure a window. Now that the examples are becoming more Mac-like, I'll have to handle windows in a more appropriate manner.

The Need to Redraw a Window's Contents

Since you've just finished with the AlertInput program, I'll use it as an example of why updating windows is so important. I can easily modify the AlertInput example to create two objects instead of one. I just declare a second Rectangle variable and call New_Rectangle() a second time:

```
Rectangle  *the_rect;
Rectangle  *another_rect;

the_rect = New_Rectangle();        // create first object

the_rect->Set_Rectangle( 20, 20, 150, 80 );
the_rect->Draw_Rectangle();

another_rect = New_Rectangle();    // create second object

another_rect->Set_Rectangle( 200, 40, 275, 70 );
another_rect->Draw_Rectangle();
```

If I designate the first object a FatRect object and the second object a FancyRect object, the program's window should look like the one pictured in Figure 9–5. Right?

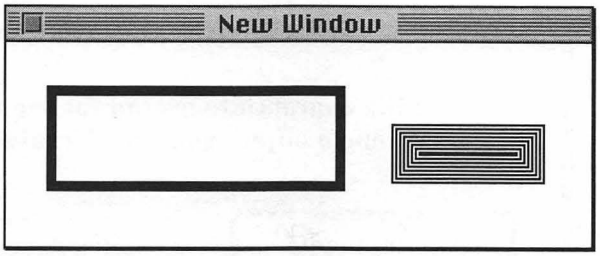


FIGURE 9–5 The expected output after creating two objects

When I ran a program that contained the above code snippet, I didn't see the results shown in Figure 9–5. Instead, I saw only the second object in the window, as shown in Figure 9–6.

What happened to the first object? It was drawn, but when New_Rectangle() was called a second time, the alert obscured the window, as shown in Figure 9–7.

When the alert was dismissed, the window's contents were empty. Then the second object was sent one message to establish the coordinates of its rectangle and another message to draw its rectangle:

```
another_rect->Set_Rectangle( 200, 40, 275, 70 );
another_rect->Draw_Rectangle();
```

What should have happened before the above code ran? When the alert was dismissed, my program should have determined that the contents of the obscured window needed to be redrawn, or updated. What my

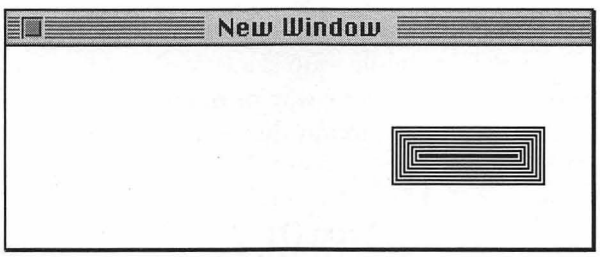


FIGURE 9–6 The actual output after creating two objects

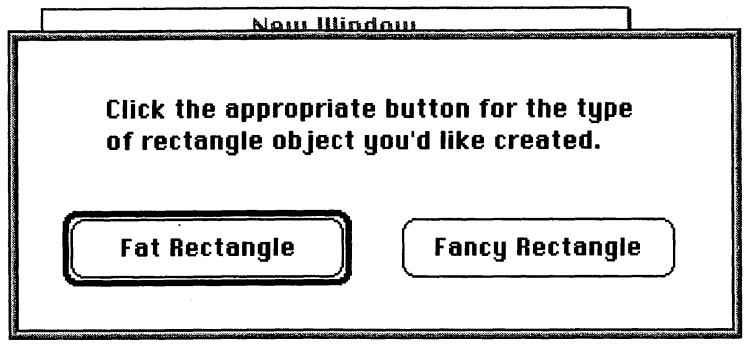


FIGURE 9–7 An alert obscuring part of a window

program needs is a means of determining when an update is necessary. That means that it's time for the addition of a real event loop to my program.

Update Events and Redrawing a Window's Contents

A Mac program should be event-driven; it should be aware of, and respond to, events such as mouse clicks and keystrokes. For the sake of simplicity, my examples haven't been event-driven. This chapter and the ones that follow will be discussing Macintosh-specific topics rather than generic object-oriented topics. Programs that include interface features, such as alerts and dialog boxes for entering object information and windows for outputting information, require a knowledge of events. My event loop, shown below, doesn't provide this information.

```
while ( !Button() )
   ;
```

The above while loop is aware of only one type of event—a click of the mouse button. A better way of doing things is to have an infinite loop that repeatedly calls a function that examines a single event and acts accordingly:

```
for ( ; ; )
   Handle_One_Event();
```

If you've programmed the Mac using C, the Handle_One_Event() routine will look very familiar. It calls the Toolbox function

WaitNextEvent() to grab hold of the most recent event. Then a switch statement is used to determine how to best handle this particular event.

```
void  Handle_One_Event( void )
{
   EventRecord  the_event;
   WindowPtr    window;
   GrafPtr      old_port;

   WaitNextEvent( everyEvent, &the_event, 15L, 0L );

   switch ( the_event.what )
   {
      case mouseDown:
         if ( the_rect != nil )
            delete the_rect;
         ExitToShell();
         break;

      case updateEvt:
         window = (WindowPtr)the_event.message;
         GetPort( &old_port );
         SetPort( window );
         BeginUpdate( window );
            EraseRgn( window->visRgn );
            if ( the_rect != nil )
               the_rect->Draw_Rectangle();
         EndUpdate( window );
         SetPort( old_port );
         break;
   }
}
```

In keeping with my philosophy of providing examples that have a minimum of code, I'll continue to simply use a click of the mouse button to end the program. A mouse click is an event, and WaitNextEvent() will respond by placing a value of mouseDown in the what field of the EventRecord variable the_event. Handle_One_Event() will then delete the the_rect object, if present, and then call the Toolbox function ExitToShell() to exit the program.

NOTE

A *nil pointer* is a pointer that hasn't been assigned to point to anything. When `the_rect` is declared, it has a value of nil. After the new operator is used to assign the pointer to point to an object, it will have a value other than nil.

Is it important that I delete the `the_rect` object before quitting the program? That depends. If the object has a destructor member function, then it is important, because the deletion of an object is what triggers the object's destructor function. The Toolbox function ExitToShell() doesn't delete objects, so no destructors will be called as the program exits. The examples in the last several chapters haven't had objects with destructors, nor do the ones in this chapter. Since I'm starting to fill in my source code to make it more like a full-featured program, however, I'll get in the habit of properly cleaning up before exiting—whether I use destructors or not.

After a mouseDown event, the second type of event the Handle_One_Event() function responds to is an updateEvt. When any part of an obscured window comes back into view, the system generates an update event. That's what happens when an alert that is covering (or partially covering) a window is dismissed. If a program includes an event loop with a call to WaitNextEvent() in it, the update event will be captured, and the loop can respond to it.

If more than one window is on the screen, your program must know which window needs updating. That information is held in the message field of the EventRecord returned by WaitNextEvent(). Before attempting to update the window, extract this information and cast it to a WindowPtr. Then call GetPort() to save the current port and SetPort() to set the active port to the window that needs updating.

```
window = (WindowPtr)the_event.message;
GetPort( &old_port );
SetPort( window );
```

IMPORTANT

Skipping the steps that save and set ports *may* work in a single-window program. Then again, it may not. And if you ever change your program to include more than one window, a failure to keep track of graphics ports will almost always lead to problems drawing to windows and updating them. Save those ports!

```
BeginUpdate( window );
   EraseRgn( window->visRgn );
   if ( the_rect != nil )
      the_rect->Draw_Rectangle();
EndUpdate( window );
```

Once you've told the Mac which port—or window—it will be working with, go ahead and begin the update. Window updating is something that you, the programmer, are responsible for. The Mac doesn't know exactly when you're going to take care of the updating, so you have to tell it. That's exactly what the calls to BeginUpdate() and EndUpdate() do. Events are stored in an event queue, or holding area. When you call the BeginUpdate() and EndUpdate() pair of Toolbox routines, the Mac knows that this one event has been handled and that it's time to remove it from the queue.

Between the calls to BeginUpdate() and EndUpdate() lies the code that does the actual updating. EraseRgn() lets QuickDraw know which part of the window needs redrawing. Then, it's up to you to draw the window contents. For my example, simply redrawing the `the_rect` object does the trick. As I did for the mouseDown event, I first make sure that the object exists. I know that it was declared, but I don't know at exactly what point in the execution of the program it was created. If the object has any value other than nil, memory has been allocated for it. That means it's safe to draw its rectangle data member.

Before I consider my handling of the update event complete, I have to reset the graphics port to whatever port was active before the update began. I don't have to attempt to figure out which port that was; I saved that information at the start of the update when I called GetPort(). Now I call SetPort() to restore the saved port to the active port:

```
SetPort( old_port );
```

Testing the Object Update

To test the updating of a window to which object information has been drawn, I'll write a simple program that opens a window, creates a Rectangle object, and then draws the object's rectangle to the window. Next, I'll open an alert. The only purpose of the alert will be to obscure the window. When the alert is dismissed, the rectangle that was in the window should be redrawn—if the updateEvt is handled properly by my event loop.

The resource file for my test program will need resources for two alerts. The first alert you've already seen—it's the one posted by New_Rectangle()

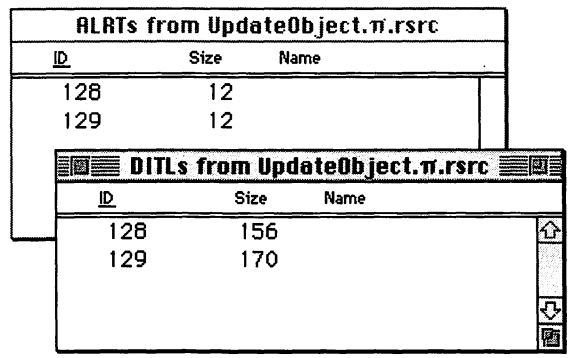


FIGURE 9–8 A program that has two alerts requires two ALRT resources and two DITL resources.

to allow the user to select an object type. The second alert will exist only to obscure the program's window. Figure 9–8 shows the resource IDs of the two ALRT resources and the two DITL resources my program will need.

Figure 9–9 is a look at the DITL resource for the second alert. It consists of a button to dismiss the alert and a text item that tells what the alert is doing.

In my source code I'll include code that opens a window, sets the port, creates a new object, and draws the object's rectangle. Then I'll post the alert

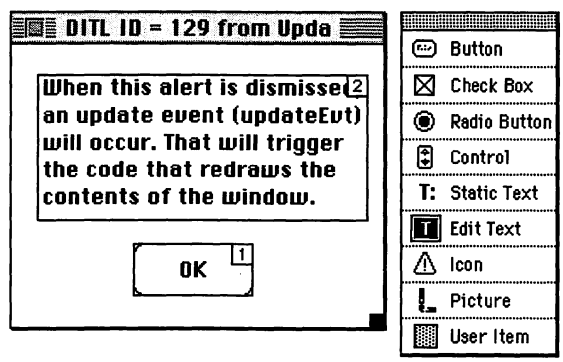


FIGURE 9–9 The DITL resource for the second alert

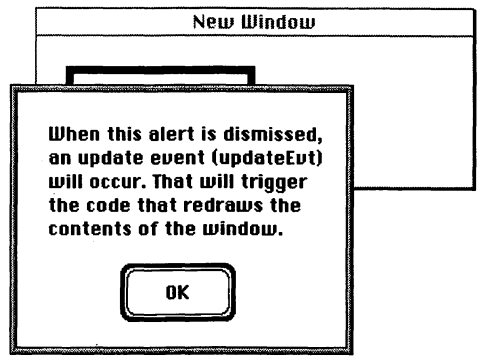


FIGURE 9–10 The second alert will obscure part of the window.

that obscures the window. Figure 9–10 shows what the user will see after the call to Alert().

```
the_window = GetNewWindow( WIND_ID, nil, (WindowPtr)-1L );
SetPort( the_window );

the_rect = New_Rectangle();
the_rect->Set_Rectangle( 20, 20, 150, 80 );
the_rect->Draw_Rectangle();

Alert( 129, nil );
```

Let's see what happens when the user dismisses the alert. I ran my test program with the Use Debugger feature turned on so that I could step through it. I set a break point at the first line in the updateEvt section of my Handle_One_Event() routine. Sure enough, when the alert was dismissed, the program stopped at this breakpoint. Figure 9–11 shows what my screen looked like. Note that the lower part of the program window is blank—that's the part that was covered by the alert. Also notice the value of the object pointer variable `the_rect`, which is shown in the Data window in Figure 9–11. Its exact value isn't important. What is significant is that it doesn't have a value of 0x00000000. A pointer that has a value of all zeros is a nil pointer. Because `the_rect` was assigned to point at an object earlier in the program, it shouldn't be—and isn't—a nil pointer.

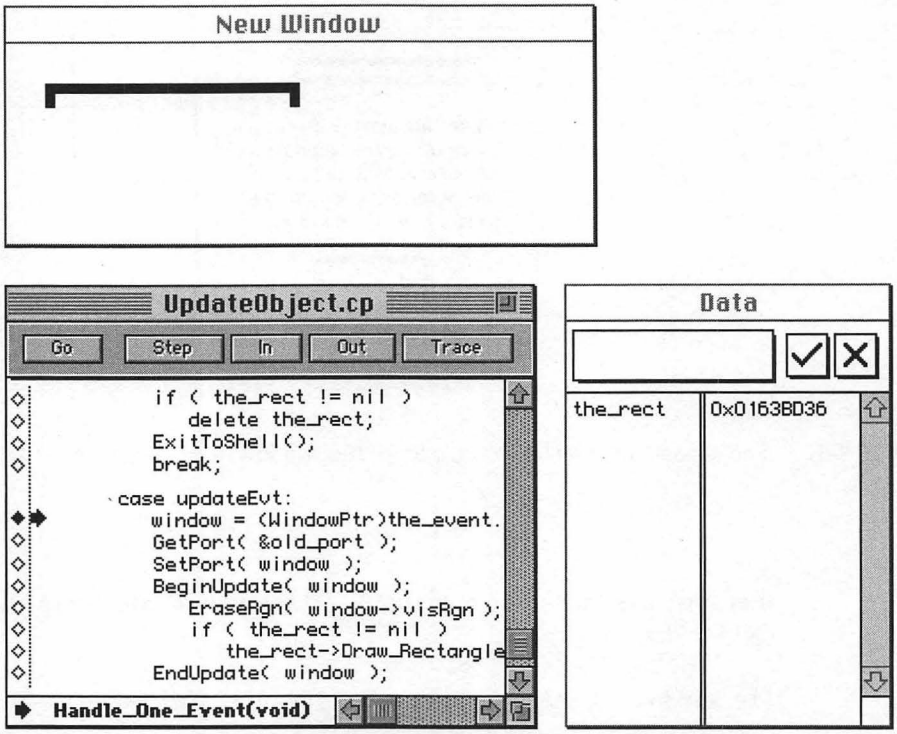


FIGURE 9–11 The test program paused at a breakpoint by the update event

To verify that my update code is working, I stepped through the updateEvt case. Figure 9–12 shows that, because the object pointer isn't nil, the code under the if statement is executed. The `the_rect` object gets a draw message sent to it, and the object's rectangle is again drawn to the window.

A nil pointer is one that doesn't point to any valid data. That's represented by a value of 0x00000000. To verify my assumptions about `the_rect`, I reran my test program. This time, before clicking on the Go button, I set a breakpoint at the line of code that gives `the_rect` its value. In Figure 9–13, you can see that the program is stopped at this point. The arrow pointing to the New_Rectangle() line means this line of code is the next line to execute. That means that `the_rect` shouldn't have a value at this point; it doesn't get a value until this line executes. Notice in the Data window in Figure 9–13 that `the_rect` does indeed have a nil value.

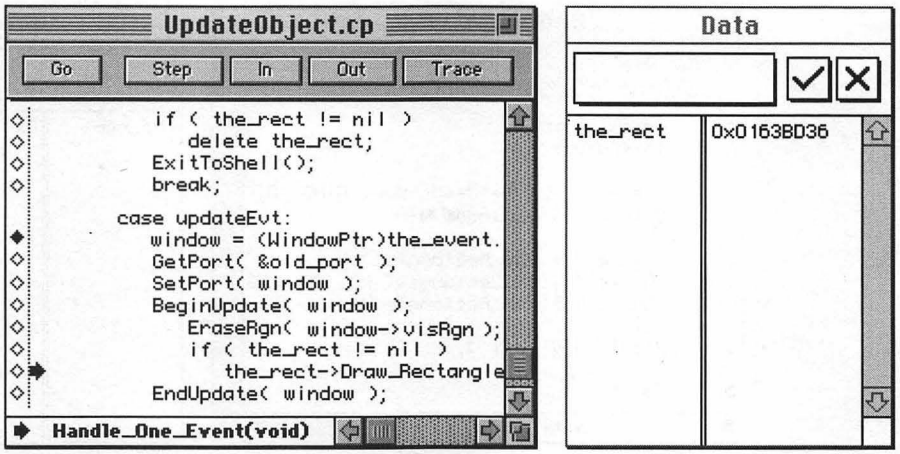


FIGURE 9–12 The test program paused at a breakpoint just before the window is updated

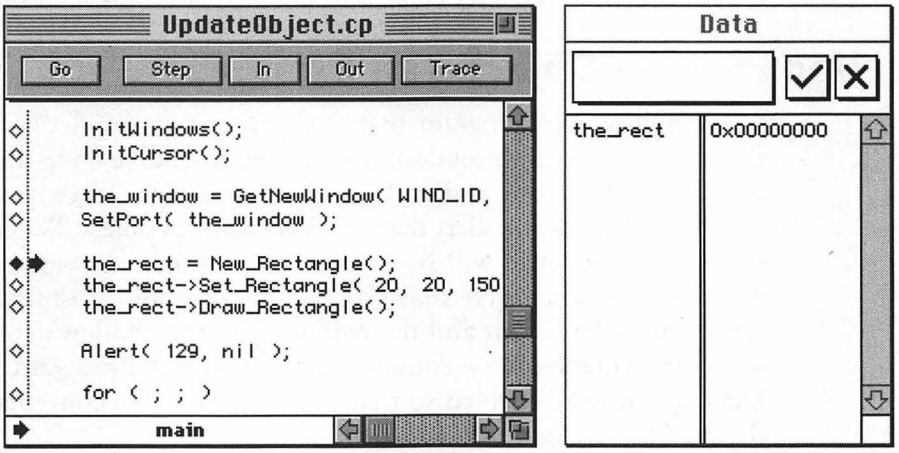


FIGURE 9–13 The object pointer has a value of nil before New_Rectangle() is called.

Clicking on the Step button once executes the line of code that holds the call to the New_Rectangle() routine. When the call is completed, the_rect should be pointing to an object, and the black arrow should move down a line. Figure 9–14 shows that this is exactly what has happened. You can see in the Data window that the_rect is no longer a nil pointer. It is now

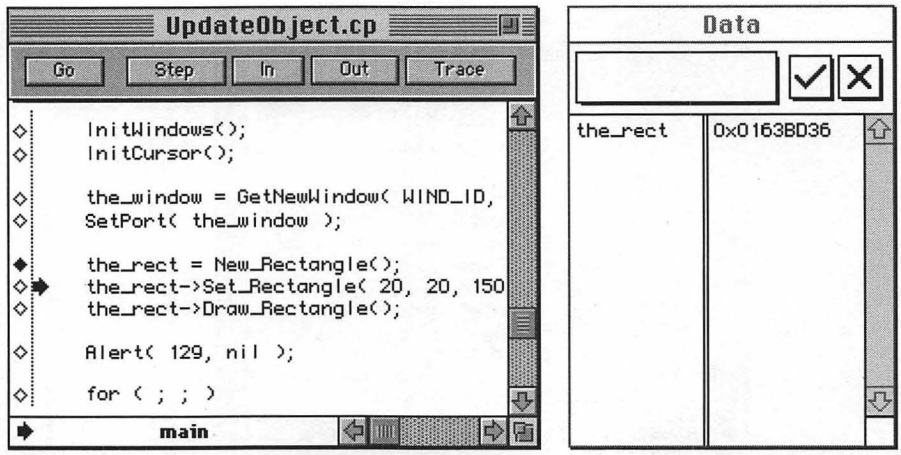


FIGURE 9–14 After the object is created, the object pointer no longer has a value of nil.

pointing to the start of the data that describes what the_rect is—the data that holds the new object created in the New_Rectangle() function.

Updating an Object—an Example

UpdateObject is a program that demonstrates the updating of a window that holds object information. You've seen its source code throughout this chapter, so I'll spare you a detailed explanation. When you run UpdateObject, you'll see an alert that lets you create an object. When you dismiss the alert, a rectangle will be drawn to the program's window. After that, you'll see a second alert that obscures the window. Dismissing that alert causes an update event and the redrawing of the window's contents. If you have the Symantec C++ compiler, try running the program with the Use Debugger feature enabled so that you can verify the concepts discussed in the previous section.

```
// ****************  UpdateObject.cp  ****************

//_____________________________________________________
//                                        forward references

class    Rectangle;
```

```
//_____________________________________________________________
//                                          function prototypes

Rectangle  *New_Rectangle( void );
void        Handle_One_Event( void );


//_____________________________________________________________
//                                          #define directives

#define     WIND_ID             128
#define     ALRT_ID             128
#define     OBSCURE_ALRT_ID     129


//_____________________________________________________________
//                                            global variables

Rectangle  *the_rect;


//_____________________________________________________________
//                                           class definitions

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


class  FatRect : public  Rectangle
{
   public:
```

```
      void  Draw_Rectangle( void );
};


class  FancyRect : public  Rectangle
{
   public:
      void  Draw_Rectangle( void );
};


//_____________________________________________________________
//                                  member function definitions

void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}


void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


void  FatRect :: Draw_Rectangle( void )
{
   PenSize( 5, 5 );
   FrameRect( &this->aRect );
   PenNormal();
}


void  FancyRect :: Draw_Rectangle( void )
{
   short  i;              // loop counter
   short  height;         // height of the rectangle
```

```
    short  num_rects;      // number of inset rects to draw
    Rect   temp;           // work with a temporary rectangle

    height = this->aRect.bottom - this->aRect.top;
    num_rects = ( height/4 );

    temp = this->aRect;

    for ( i = 0; i <= num_rects; i++ )
    {
       FrameRect( &temp );
       InsetRect( &temp, 2, 2 );
    }
}


//_____________________________________________________________________
//                                                              main()

void  main( void )
{
    WindowPtr   the_window;

    InitGraf( &thePort );
    InitFonts();
    InitWindows();
    InitCursor();

    the_window = GetNewWindow( WIND_ID, nil,
                               (WindowPtr)-1L );
    SetPort( the_window );

    the_rect = New_Rectangle();
    the_rect->Set_Rectangle( 20, 20, 150, 80 );
    the_rect->Draw_Rectangle();

    Alert( OBSCURE_ALRT_ID, nil );

    for ( ; ; )
```

```
        Handle_One_Event();
}


//_____________________________________________________________
//                          open alert and create new object

Rectangle  *New_Rectangle( void )
{
   Rectangle  *temp;
   short       choice;

   choice = Alert( ALRT_ID, nil );

   if ( choice == 1 )
      temp = new  FatRect;
   else
      temp = new  FancyRect;

   return ( temp );
}


//_____________________________________________________________
//                                          handle single event

void  Handle_One_Event( void )
{
   EventRecord  the_event;
   WindowPtr    window;
   GrafPtr      old_port;

   WaitNextEvent( everyEvent, &the_event, 15L, 0L );

   switch ( the_event.what )
   {
      case mouseDown:
         if ( the_rect != nil )
            delete the_rect;
```

```
            ExitToShell();
            break;

        case updateEvt:
          window = (WindowPtr)the_event.message;
          GetPort( &old_port );
           SetPort( window );
           BeginUpdate( window );
              EraseRgn( window->visRgn );
              if ( the_rect != nil )
                 the_rect->Draw_Rectangle();
           EndUpdate( window );
           SetPort( old_port );
           break;
    }
}
```

Using a Dialog Box to Create a New Object

From the programmer's vantage point, an alert is very easy to handle—a call to the Toolbox routine Alert() does all the work. Ease-of-use was, in fact, my reason for using an alert to introduce the idea of an object-oriented program receiving object information from the user. But an alert has several limitations that make it impractical when your program requires interaction that is anything more than a simple choice. To move to the next level, you'll want to use a dialog box.

Dialog Box Resources

My earliest attempts at using a function to vary the type of object created required no intervention on the part of the user. I made the decision by passing a particular value to the function:

```
the_rect = New_Rectangle( 2 );
```

Using an alert to place this decision in the hands of the user is a step in the right direction, but it doesn't go far enough. The user doesn't have any

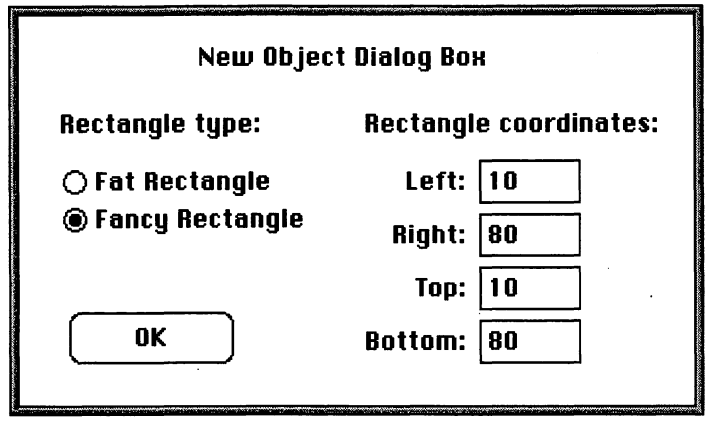


FIGURE 9–15 A dialog box that lets the user provide the specifications for an object

say in establishing the size of the object's rectangle. That is still decided by me when I set the parameters for a call to Set_Rectangle():

```
the_rect->Set_Rectangle( 20, 20, 150, 80 );
```

A better approach would be to use a dialog box. With a little planning, a single dialog box will allow the user to supply all the information necessary to define an object. For a program that uses the Rectangle class, a dialog box like the one pictured in Figure 9–15 would be perfect.

A dialog box requires a DLOG resource and a DITL resource. Figure 9–16 shows the DITL I created for a dialog box like the one pictured in Figure 9–15. From the name of the resource file, you can see that this DITL will be used in a program named DialogInput. You'll see the complete source code listing for that program later in this chapter

In Figure 9–16, item 1 is a push button. Items 2 and 3 are radio buttons, and items 4 through 7 are edit text boxes. The remaining items are static text items. Figure 9–16 shows how the DITL looks in ResEdit when the Show Item Numbers menu option is selected. I'll need these item numbers when it comes time to write my source code. If you have enough memory on your Mac, you might want to run your resource editor and the Symantec C++ compiler at the same time. Then you can simply switch from the resource editor to the compiler and type in some #define directives as you work on the resource file—just as I've done here:

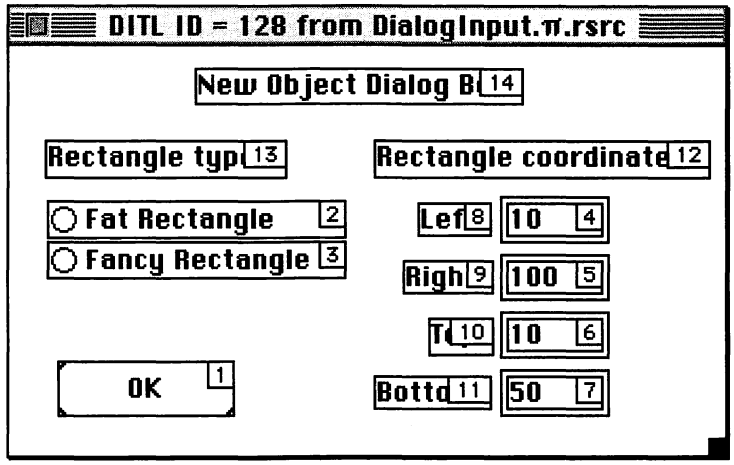


FIGURE 9–16 The DITL resource used to create the dialog box

```
#define     WIND_ID             128
#define     DLOG_ID             128
#define     OK_BUTTON             1
#define     FAT_OBJECT_TYPE       2
#define     FANCY_OBJECT_TYPE     3
#define     LEFT_ITEM             4
#define     RIGHT_ITEM            5
#define     TOP_ITEM              6
#define     BOTTOM_ITEM           7
```

Handling Radio Button Items in a Dialog Box

The code that supports my dialog box must be able to handle radio buttons and edit text boxes. I'll examine the handling of radio buttons first. To keep my code nice and portable, I'll write a simple utility routine that handles the turning off of one radio button and the turning on of another. That's how groups of radio buttons always work; when a radio button is clicked on, it is turned on and the button that was on goes off. I'll call the clicked-on button the new button, and the button that should be turned off the old button. Here's the function that handles that chore:

```
void Set_Radio_Buttons( DialogPtr dlog, short *old_radio,
                        short new_radio )
{
   Handle  hand;
   short   type;
   Rect    box;

   GetDItem( dlog, *old_radio, &type, &hand, &box );
   SetCtlValue( (ControlHandle)hand, 0 );

   GetDItem( dlog, new_radio, &type, &hand, &box );
   SetCtlValue( (ControlHandle)hand, 1 );

   *old_radio = new_radio;
}
```

The Set_Radio_Buttons() function has three arguments. The first is a pointer to the dialog box that holds that button. The second argument is a pointer to the old button—the one that is to be turned off. The third argument is the item number of the new radio button—the button that is to be turned on.

A call to GetDItem() provides information about the old radio button. Note that since `old_radio` was passed in as a pointer, I have to use the * operator here to dereference the pointer. My only purpose in calling this function is to get a handle to the item. GetDItem() returns one in the `hand` variable. Once I have a handle to the button, I can cast it to a ControlHandle and use it as a parameter in a call to SetCtlValue(). This function will turn a control—such as a radio button—on or off. The second parameter to SetCtlValue() tells the routine whether the control should be turned on or off. A value of 1 turns it on, and a value of 0 turns it off. I'll pass in a 0 to turn the old radio button off.

Set_Radio_Buttons() makes second calls to GetDItem() and SetCtlValue() to turn the new radio button on. Then, an assignment is made. Now that the new radio button is turned on, I'll want to consider it the old button. That way, when the user again clicks on a radio button, the proper pair of button item numbers will be sent to Set_Radio_Buttons(). The changing of `old_radio` from its current value to the value of `new_radio` explains why `old_radio` was passed in as a pointer. Since the function is changing the value of a variable—`old_radio`—that variable had to be passed in as a pointer.

My dialog-handling routine, which you'll see shortly, will respond to a mouse click on a radio button by setting the value of `new_radio` to the item

number of the clicked-on item. Then a call to Set_Radio_Buttons() will be made—like this:

```
DialogPtr  dlog;
short      item;
short      new_radio;
short      old_radio;

// open dialog box
// set item equal to item number of clicked-on item

new_radio = item;
Set_Radio_Buttons( dlog, &old_radio, new_radio );
```

Handling Edit Text Items in a Dialog Box

The handling of edit text items is done a little differently than the handling of radio buttons. When a user clicks on a radio button, your code must respond immediately to change the button settings. When a user types in an edit box, however, the program need do nothing. The user can type, edit, and retype in an edit box as much as he or she wants—without any intervention on your program's part. It's not until the user is done with the dialog box—when a click on the OK button confirms that the values in the edit boxes are correct—that your program need take action.

My dialog box has four edit boxes—one for each coordinate of the new rectangle object that's about to be created. Because I'll be repeating the same code four times, I've written a short function that can be called to get the information from each of the edit boxes. Because I pass the function a pointer to the dialog box that holds the edit text boxes, the function is generic enough that it can be used for any dialog box—in this program or any other program I write. Here's a look at the Get_Text_From_Edit() routine:

```
void  Get_Text_From_Edit( DialogPtr dlog, short edit_item,
                          Str255 the_str )
{
   Handle  hand;
   short   type;
   Rect    box;

   GetDItem( dlog, edit_item, &type, &hand, &box );
   GetIText( hand, the_str );
}
```

In addition to a pointer to the dialog box, Get_Text_From_Edit() receives the item number of an edit box and a Str255 as parameters. A call to GetDItem() returns a handle to the edit box, and a call to GetIText() makes use of that handle to get the text from that edit box. GetIText() places that text in the Str255 variable. When the function completes, the calling routine will have the edit text item text in the Str255 variable the_str. The code snippet that follows shows a call to Get_Text_From_Edit(). The call passes to Get_Text_From_Edit() the item number of the edit box that holds the user-entered value that will serve as the left coordinate of the rectangle object.

```
DialogPtr  dlog;
Str255     the_str;
long       L, T, R, B;

// open dialog box
// when OK button is clicked on, get edit box info

Get_Text_From_Edit( dlog, LEFT_ITEM, the_str );
StringToNum( the_str, &L );
```

After the call to Get_Text_From_Edit(), the above code makes a call to StringToNum(). That Toolbox function will convert the string value to the number it represents and store that number in the short variable L.

Creating a New Object Using a Dialog Box

Now that I've covered the handling of radio buttons and edit text boxes, covering the function that opens the dialog box will be straightforward. My previous version of New_Rectangle() posted an alert that let the user select the object type. This latest version of New_Rectangle() will open a dialog box. Handling a dialog box requires a greater effort on the part of the programmer, so New_Rectangle() has grown —but not unmanageably so. I'll show the function in its entirety, and then explain it in detail.

```
Rectangle  *New_Rectangle( void )
{
   DialogPtr  dlog;              // pointer to dialog
   Boolean    dlog_done = false; // done with dialog?
   short      item;              // clicked-on item
   short      new_radio;         // radio button to turn on
   short      old_radio;         // radio button to turn off
```

```
Str255      the_str;          // string from an edit box
long        L, T, R, B;       // rectangle coordinates
short       object_type;      // FatRect or FancyRect
Rectangle   *temp;            // pointer to rectangle
                              // object


dlog = GetNewDialog( DLOG_ID, nil, (WindowPtr)-1L );

old_radio = FANCY_OBJECT_TYPE;
new_radio = FAT_OBJECT_TYPE;
Set_Radio_Buttons( dlog, &old_radio, new_radio );
object_type = FAT_OBJECT_TYPE;

while ( dlog_done == false )
{
   ModalDialog( nil, &item );

   switch ( item )
   {
      case FAT_OBJECT_TYPE:
         new_radio = item;
         Set_Radio_Buttons( dlog, &old_radio,
                            new_radio );
         object_type = FAT_OBJECT_TYPE;
         break;

      case FANCY_OBJECT_TYPE:
         new_radio = item;
         Set_Radio_Buttons( dlog, &old_radio,
                            new_radio );
         object_type = FANCY_OBJECT_TYPE;
         break;

      case OK_BUTTON:
         Get_Text_From_Edit( dlog, LEFT_ITEM, the_str );
         StringToNum( the_str, &L );
         Get_Text_From_Edit( dlog, RIGHT_ITEM,
                             the_str );
```

```
                StringToNum( the_str, &R );
                Get_Text_From_Edit( dlog, TOP_ITEM, the_str );
                StringToNum( the_str, &T );
                Get_Text_From_Edit( dlog, BOTTOM_ITEM,
                                    the_str );
                StringToNum( the_str, &B );
                switch ( object_type )
                {
                   case FAT_OBJECT_TYPE:
                      temp = new  FatRect;
                      break;
                   case FANCY_OBJECT_TYPE:
                      temp = new  FancyRect;
                      break;
                }
                temp->Set_Rectangle( L, T, R, B );
                dlog_done = true;
                break;
             }
          }


          DisposDialog( dlog );

          return ( temp );
       }
```

The first thing New_Rectangle() does is open the dialog box. A call to GetNewDialog(), with the ID of the DLOG resource to use serving as the first parameter, accomplishes this task:

```
dlog = GetNewDialog( DLOG_ID, nil, (WindowPtr)-1L );
```

When a dialog box that contains radio buttons opens, none of the radio buttons will be turned on—that's a task for the programmer. The program uses a #define directive for each of the item numbers of the dialog box items. I'll use these values in setting the radio buttons:

```
#define     FAT_OBJECT_TYPE       2
#define     FANCY_OBJECT_TYPE     3
```

I set the old radio button to a value of FANCY_OBJECT_TYPE and the new radio button to a value of FAT_OBJECT_TYPE, and then I call Set_Radio_Buttons(). Because this routine turns a radio button off before turning one on, I have to pass in the item number of a button to turn off. The buttons are all off at the start, so it really doesn't matter which item number I set the `old_radio` variable to. After calling Set_Radio_Buttons(), I set the `object_type` variable to the item number of the radio button that is turned on. By always keeping the value of this variable the same as the value of the item number of the turned-on button, I'll always know what type of object the user wants to create.

```
old_radio = FANCY_OBJECT_TYPE;
new_radio = FAT_OBJECT_TYPE;
Set_Radio_Buttons( dlog, &old_radio, new_radio );
object_type = FAT_OBJECT_TYPE;
```

After setting the radio buttons, New_Rectangle() enters a while loop, which will run until the user clicks on the OK button. A call to ModalDialog() is at the start of the loop body. If the user clicks on an item, ModalDialog() will return its DITL item number. A switch statement within the while loop examines the returned item number and compares it to the item numbers of the radio buttons and push button. If a match is made, the code under the appropriate case label is executed. In the code snippet below, I've replaced that code with comments that summarize what happens when a mouse click occurs on any of the buttons.

```
while ( dlog_done == false )
{
   ModalDialog( nil, &item );


   switch ( item )
   {
      case FAT_OBJECT_TYPE:
         // handle a click on a radio button


      case FANCY_OBJECT_TYPE:
         // handle a click on a radio button
```

```
      case OK_BUTTON:
         // done with the dialog, get edit box information:
         // get the strings from the four edit text boxes
         // create the new object
         // set the coordinates of the object's rectangle
         // exit the while loop
   }
}
```

If the user clicks on a radio button, the `new_radio` variable is set to the DITL item number of the clicked-on button. Then a call to Set_Radio_Buttons() turns the old button off and this new button on. The `object_type` variable is then updated so that it indicates the object type the user wants created. Below is the code that executes in response to a click on the radio button labeled Fancy Rectangle. Since the code that handles a click on the Fat Rectangle radio button is so similar, I won't cover it.

```
case FANCY_OBJECT_TYPE:
   new_radio = item;
   Set_Radio_Buttons( dlog, &old_radio, new_radio );
   object_type = FANCY_OBJECT_TYPE;
   break;
```

A click on the OK button signals that the user is finished with the dialog box. That means it's time to get the final values that are in the four edit boxes. A call to Get_Text_From_Edit() and StringToNum() is made for each. Then, based on the value of `object_type`, a new object is created. The object's aRect data member is given the values from the edit boxes through a call to Set_Rectangle(). Finally, `dlog_done` is set to true so that the while loop will end.

```
case OK_BUTTON:
   Get_Text_From_Edit( dlog, LEFT_ITEM, the_str );
   StringToNum( the_str, &L );
   Get_Text_From_Edit( dlog, RIGHT_ITEM,
                       the_str );
   StringToNum( the_str, &R );
   Get_Text_From_Edit( dlog, TOP_ITEM, the_str );
   StringToNum( the_str, &T );
   Get_Text_From_Edit( dlog, BOTTOM_ITEM,
                       the_str );
```

```
StringToNum( the_str, &B );
switch ( object_type )
{
   case FAT_OBJECT_TYPE:
      temp = new  FatRect;
      break;
   case FANCY_OBJECT_TYPE:
      temp = new  FancyRect;
      break;
}
temp->Set_Rectangle( L, T, R, B );
dlog_done = true;
 break;
```

NOTE

Before New_Rectangle() could be considered complete, it would have to have some error-checking added to it. For instance, what if the user enters a word—rather than a number—in one of the edit boxes? The function should check for mistakes such as this and post an alert that tells the user what mistake was made. Forgive me if I end this note by saying that the addition of error-checking is left as an exercise for the reader.

When the while loop ends, the dialog box is dismissed by disposing it with a call to DisposDialog(). Then the newly created object is returned to the routine that called New_Rectangle():

```
DisposDialog( dlog );

return ( temp );
```

A Dialog Box Example Program

As always, I'll end the section with an example program. DialogInput uses the New_Rectangle() routine developed in this chapter to display a dialog box like the one shown in Figure 9–17.

If the user were to enter the values shown in the dialog box in Figure 9–17, the rectangle would look like the one shown in Figure 9–18.

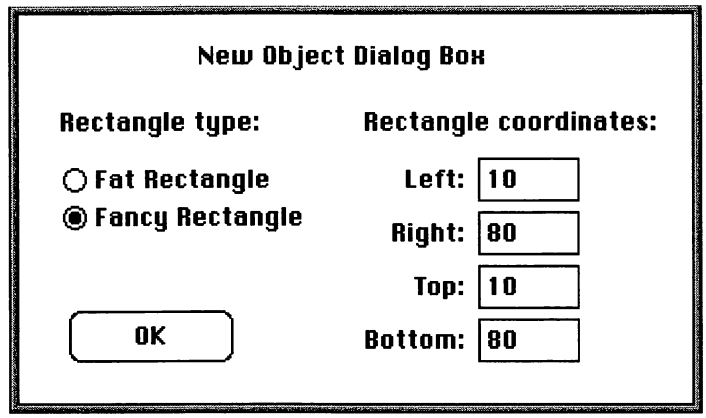


FIGURE 9–17 A dialog box that lets the user provide the specifications for an object

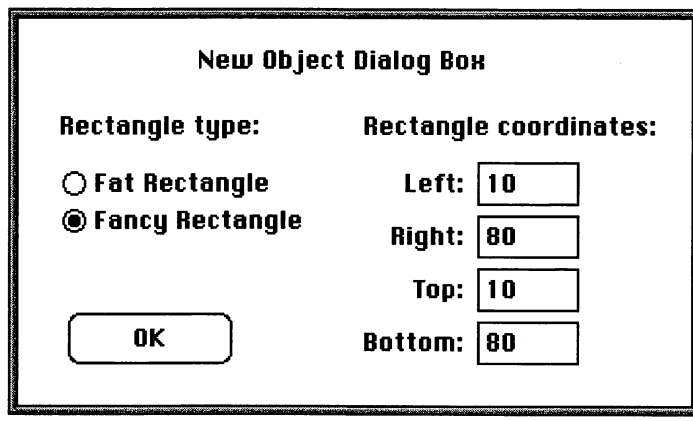


FIGURE 9–18 The rectangle that is drawn after the user creates an object

```
// ******************  DialogInput.cp  ******************

//_____________________________________________________________
//                                          forward references

class    Rectangle;


//_____________________________________________________________
//                                        function prototypes

Rectangle  *New_Rectangle( void );
void        Handle_One_Event( void );
void        Set_Radio_Buttons( DialogPtr, short *, short );
void        Get_Text_From_Edit( DialogPtr, short, Str255 );


//_____________________________________________________________
//                                          #define directives

#define     WIND_ID              128
#define     DLOG_ID              128
#define     OK_BUTTON              1
#define     FAT_OBJECT_TYPE        2
#define     FANCY_OBJECT_TYPE      3
#define     LEFT_ITEM              4
#define     RIGHT_ITEM             5
#define     TOP_ITEM               6
#define     BOTTOM_ITEM            7


//_____________________________________________________________
//                                            global variables

Rectangle  *the_rect;
```

```
//_____________________________________________________________
//                                              class definitions

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short, short, short,
                                   short );
      virtual void  Draw_Rectangle( void );
};


class  FatRect : public  Rectangle
{
   public:
      void  Draw_Rectangle( void );
};


class  FancyRect : public  Rectangle
{
   public:
      void  Draw_Rectangle( void );
};


//_____________________________________________________________
//                                  member function definitions

void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}
```

```
void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


void  FatRect :: Draw_Rectangle( void )
{
   PenSize( 5, 5 );
   FrameRect( &this->aRect );
   PenNormal();
}


void  FancyRect :: Draw_Rectangle( void )
{
   short  i;              // loop counter
   short  height;         // height of the rectangle
   short  num_rects;      // number of inset rects to draw
   Rect   temp;           // work with a temporary rectangle

   height = this->aRect.bottom - this->aRect.top;
   num_rects = ( height/4 );


   temp = this->aRect;

   for ( i = 0; i <= num_rects; i++ )
   {
      FrameRect( &temp );
      InsetRect( &temp, 2, 2 );
   }
}


//_____________________________________________________________________
//                                                              main()

void  main( void )
```

```
{
   WindowPtr   the_window;

   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitCursor();

   the_window = GetNewWindow( WIND_ID, nil,
                              (WindowPtr)-1L );
   SetPort( the_window );
   the_rect = New_Rectangle();

   the_rect->Draw_Rectangle();

   for ( ; ; )
      Handle_One_Event();
}


//_____________________________________________________________
//                          open dialog and create new object

Rectangle  *New_Rectangle( void )
{
   DialogPtr  dlog;
   Boolean    dlog_done = false;
   short      item;
   short      new_radio;
   short      old_radio;
   Str255     the_str;
   long       L, T, R, B;
   short      object_type;
   Rectangle  *temp;

   dlog = GetNewDialog( DLOG_ID, nil, (WindowPtr)-1L );

   old_radio = FANCY_OBJECT_TYPE;
   new_radio = FAT_OBJECT_TYPE;
```

```
Set_Radio_Buttons( dlog, &old_radio, new_radio );
object_type = FAT_OBJECT_TYPE;

while ( dlog_done == false )
{
   ModalDialog( nil, &item );

   switch ( item )
   {
      case FAT_OBJECT_TYPE:
         new_radio = item;
         Set_Radio_Buttons( dlog, &old_radio,
                            new_radio );
         object_type = FAT_OBJECT_TYPE;
         break;

      case FANCY_OBJECT_TYPE:
         new_radio = item;
         Set_Radio_Buttons( dlog, &old_radio,
                            new_radio );
         object_type = FANCY_OBJECT_TYPE;
         break;

      case OK_BUTTON:
         Get_Text_From_Edit( dlog, LEFT_ITEM, the_str );
         StringToNum( the_str, &L );
         Get_Text_From_Edit( dlog, RIGHT_ITEM,
                             the_str );
         StringToNum( the_str, &R );
         Get_Text_From_Edit( dlog, TOP_ITEM, the_str );
         StringToNum( the_str, &T );
         Get_Text_From_Edit( dlog, BOTTOM_ITEM,
                             the_str );
         StringToNum( the_str, &B );
         switch ( object_type )
         {
            case FAT_OBJECT_TYPE:
               temp = new  FatRect;
               break;
```

```
                case FANCY_OBJECT_TYPE:
                    temp = new  FancyRect;
                    break;
                }
                temp->Set_Rectangle( L, T, R, B );
                dlog_done = true;
                break;
            }
        }

        DisposDialog( dlog );

        return ( temp );
    }


    //_____________________________________________________________
    //                                              handle single event

    void  Handle_One_Event( void )
    {
        EventRecord  the_event;
        WindowPtr    window;
        GrafPtr      old_port;

        WaitNextEvent( everyEvent, &the_event, 15L, 0L );

        switch ( the_event.what )
        {
            case mouseDown:
                if ( the_rect != nil )
                    delete the_rect;
                ExitToShell();
                break;

            case updateEvt:
                window = (WindowPtr)the_event.message;
                GetPort( &old_port );
                SetPort( window );
```

```
            BeginUpdate( window );
               EraseRgn( window->visRgn );
               if ( the_rect != nil )
                  the_rect->Draw_Rectangle();
            EndUpdate( window );
            SetPort( old_port );
            break;
      }
}


//_____________________________________________________________
//                                          set radio buttons

void Set_Radio_Buttons( DialogPtr dlog, short *old_radio,
                        short new_radio )
{
   Handle  hand;
   short   type;
   Rect    box;

   GetDItem( dlog, *old_radio, &type, &hand, &box );
   SetCtlValue( (ControlHandle)hand, 0 );

   GetDItem( dlog, new_radio, &type, &hand, &box );
   SetCtlValue( (ControlHandle)hand, 1 );

   *old_radio = new_radio;
}


//_____________________________________________________________
//                                      get edit box contents

void  Get_Text_From_Edit( DialogPtr dlog, short edit_item,
                          Str255 the_str )
{
   Handle  hand;
   short   type;
```

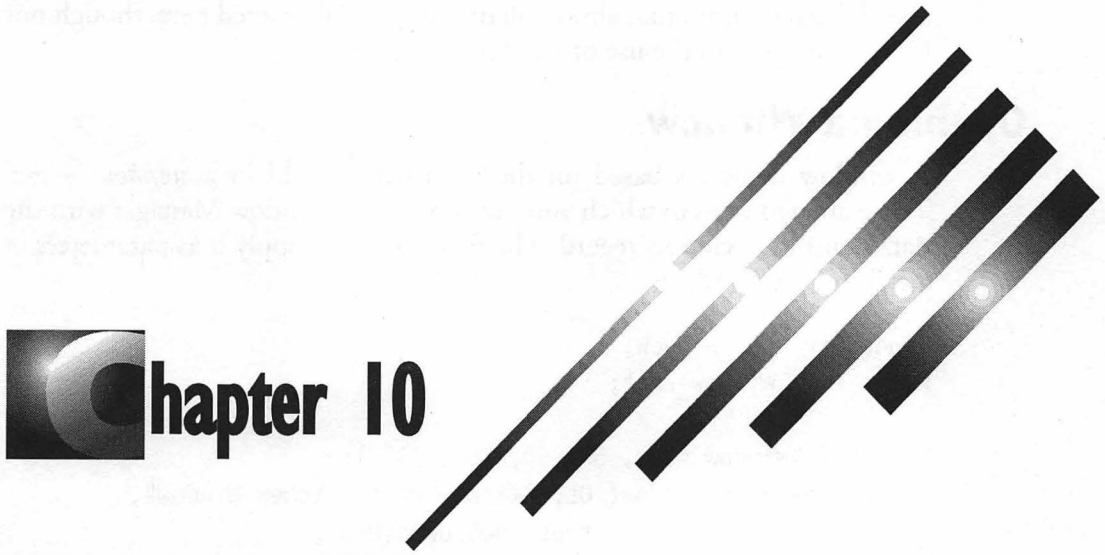
```
    Rect    box;

    GetDItem( dlog, edit_item, &type, &hand, &box );
    GetIText( hand, the_str );
}
```

Chapter Summary

Windows are the part of the graphical user interface that gives feedback to a program's user. Menus, alerts, and dialog boxes are the parts that allow the user to provide input to the program. In this chapter, you saw how alerts and dialog boxes can be used to give a program information about an object that is about to be created.

As your programs become more sophisticated through the inclusion of alerts and dialog boxes, it's important that the windows that display information be updated properly. Having your program respond to update events is the way to keep the contents of a program's window properly drawn.



Chapter 10

Windows as Objects

Windows are intuitively thought of as objects—"things" that can opened, moved, and closed. So it makes sense that an object-oriented program would define a class to represent a window and then create as many instances, or window objects, as needed. That's exactly what this chapter is all about.

Before delving into OOP techniques, you'll have an opportunity to brush up on the basics of windows and how a program handles events that involve them. Though the techniques covered at the start of the chapter deal with procedural programming, they'll apply directly to the main topic covered in the remainder of the chapter—the representation of windows as objects.

Window Basics

This chapter starts with a discussion of the basic techniques for handling windows—but not in an object-oriented program. Before you go jumping

ahead, keep in mind that almost all of the material covered here, though not OOP, will apply to the use of windows as objects.

Opening a Window

A window is always based on the information held in a *window record.* There are two ways in which you can supply the Window Manager with the data that fills a window record. The first way is to supply it as parameters in a call to NewWindow():

```
WindowPtr  the_window;
Rect       window_rect;

SetRect( &window_rect, 50, 50, 350, 150 );
the_window = NewWindow( 0L, &window_rect, "\pNew Window",
                        true, noGrowDocProc,
                        (WindowPtr)-1L, true, 0 );
```

The second way to give the Window Manager the information it needs is to create a WIND resource to hold the information and then let the Window Manager know about it using a call to GetNewWindow(). Assuming that I created a WIND resource with a resource ID of 128, a call to GetNewWindow() would look like this:

```
#define    WIND_ID      128

WindowPtr  the_window;

the_window = GetNewWindow( WIND_ID, nil, (WindowPtr)-1L );
```

Window Data Types

While the WindowPtr is by far the most commonly used window data type, there two others—the WindowRecord and the WindowPeek. You'll want to know about these data types for the upcoming discussions on using windows as objects.

Every window has its own set of properties that define what it looks like and how it behaves. So it makes sense that every window has its own data structure to hold this information. The WindowRecord is a struct data type with 17 members. It's not important that you know the purpose of each of these fields but, for the sake of completeness, I've shown the entire data structure below:

```
struct WindowRecord
{
   GrafPort        port;
   short           windowKind;
   Boolean         visible;
   Boolean         hilited;
   Boolean         goAwayFlag;
   Boolean         spareFlag;
   RgnHandle       strucRgn;
   RgnHandle       contRgn;
   RgnHandle       updateRgn;
   Handle          windowDefProc;
   Handle          dataHandle;
   StringHandle    titleHandle;
   short           titleWidth;
   ControlHandle   controlList;
   struct          WindowRecord *nextWindow;
   PicHandle       windowPic;
   long            refCon;
};
```

NOTE

If you do want to know about every field in the WindowRecord or about other nontrivial data types, consider buying one or more volumes of Apple's *Inside Macintosh* series of books.

Their names should make the purposes of some of the fields of the WindowRecord obvious. The hilited member, for example, tells whether the window is in the foreground and highlighted or in the background with highlighting off.

The WindowRecord member of most importance to the programmer is the very first one—the port field. The port, which is of type GrafPort, holds information about the graphics port of the window. The graphics port is the window's drawing environment. That is, it holds information such as the font that words will be drawn in and the size of the lines that will be used to draw shapes.

The port member—the GrafPort field—is what a WindowPtr points to. You use a WindowPtr to locate a WindowRecord in memory and to then work with the graphics port of that window. In most cases, the window's graphics port is the only WindowRecord member you'll have to access. Occasionally, however, you'll have to work with one or more of the other WindowRecord members. In cases like this, you'll use a WindowPeek instead of a WindowPtr.

Unlike a WindowPtr, which lets you access only the port member of a WindowRecord, the WindowPeek lets you access any WindowRecord member. Figure 10–1 shows the difference between these two data types.

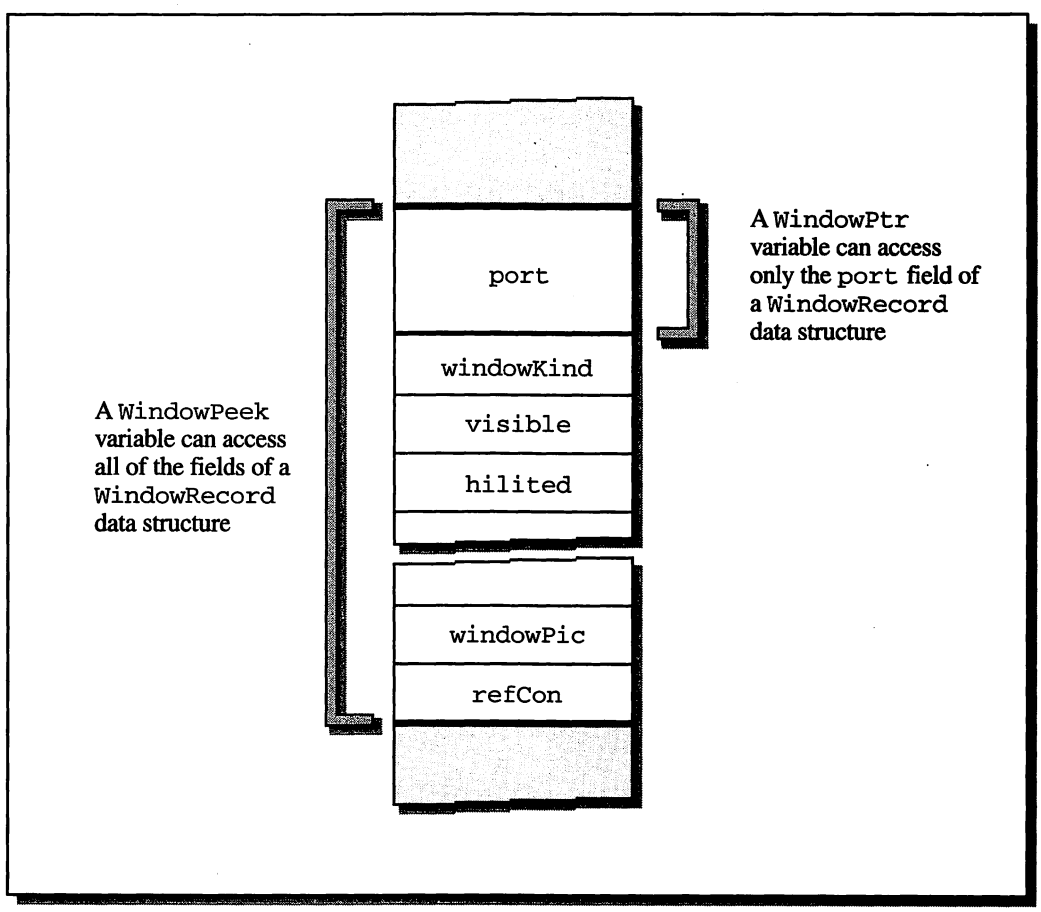


FIGURE 10–1 A WindowPtr allows access to only the port, while a WindowPeek allows access to the entire WindowRecord.

If it seems to you that the WindowPeek is a more powerful data type than the WindowPtr, you're right. The two data types weren't, however, designed to compete with one another; they were created to complement each other. The WindowPtr can access only the WindowRecord's port member because that's the most-used data member. The rest of the data members remain hidden from the WindowPtr—and from accidental tampering by the programmer. If you want to access a field of a WindowRecord other than the port field, you usually won't do so directly. Instead, you'll use a Toolbox routine that was written expressly for accessing a WindowRecord. For instance, to change a window's title you won't manipulate the WindowRecord's titleHandle field yourself. Instead, you'll call SetWTitle() to do the work for you.

NOTE

The WindowRecord fields other than the port field are hidden from the outside world. Here you see that while data hiding is generally thought of as a technique used in object-oriented programming, it applies to other areas of programming as well. To take the OOP analogy a little further, you might think of Toolbox functions that access WindowRecord members (like SetWTitle()) as member functions that access class data members.

The WindowPeek was designed for those few times when a programmer must directly access WindowRecord fields other than the port field. The most common use of a WindowPeek is to change or examine the last member in the WindowRecord—the refCon field. This long integer can be used to hold a value that helps the programmer keep track of a particular window in a program that uses multiple windows. Later in this chapter, you'll see that the refCon field will become very important to the success of a program that represents windows as objects.

Windows and Events

Once your program has a window open, you'll want to be able to work with the window—to have the window respond to events as they happen. Of course, you'll rely on the event-handling routine that is invoked repeatedly by the main() function:

```
void  main( void )
{
   WindowPtr  the_window;
```

```
    Initialize_Toolbox();

    the_window = GetNewWindow( WIND_ID, nil,
                               (WindowPtr)-1L );

    for ( ; ; )
       Handle_One_Event();
}
```

Mac programs are primarily concerned with two event types—a click of the mouse button, or mouseDown event, and an update event, or updateEvt. For clarity, once the event type is determined, it is usually handled by a separate routine. I use this approach in the Handle_One_Event() function that's listed below:

```
void  Handle_One_Event( void )
{
   EventRecord  the_event;
   WindowPtr    window;
   GrafPtr      old_port;

   WaitNextEvent( everyEvent, &the_event, 15L, 0L );

   switch ( the_event.what )
   {
      case mouseDown:
         Handle_Mouse_Down( &the_event );
         break;

      case updateEvt:
         window = (WindowPtr)the_event.message;
         GetPort( &old_port );
         SetPort( window );
         BeginUpdate( window );
            EraseRgn( window->visRgn );
            Update( window );
         EndUpdate( window );
         SetPort( old_port );
         break;
   }
}
```

A click of the mouse button sends the program to a mouse-handling routine—I've called mine Handle_Mouse_Down(). A call to the Toolbox function FindWindow() tells Handle_Mouse_Down() in which part of the screen or window the mouse click occurred and in which window (if any). A switch statement is then used to execute the code appropriate to the location of the mouse-button click.

```
void  Handle_Mouse_Down( EventRecord *the_event )
{
   WindowPtr    window;
   short        the_part;

   the_part = FindWindow ( the_event->where, &window);

   switch ( the_part )
   {
      case inSysWindow:
         SystemClick ( the_event, window );
         break;

      case inDrag:
         DragWindow( window, the_event->where,
                     &screenBits.bounds );
         break;

      case inContent:
         SelectWindow( window );
         break;

      case inGoAway:
         if ( TrackGoAway( window, the_event->where ) )
            DisposeWindow( window );
         break;
   }
}
```

A click in a system window—that is, a nonapplication window such as a desk accessory—brings about a call to SystemClick(). This Toolbox function takes control and handles the event by activating the clicked-on

desk accessory. SystemClick() then takes further action, such as dragging or closing the desk accessory window.

```
case inSysWindow:
   SystemClick ( the_event, window );
   break;
```

A click in the drag bar of an application window results in a call to the Toolbox function DragWindow(). As long as the user holds the mouse button down in the window's title bar, DragWindow() will stay in effect, moving the window as the user drags the mouse.

```
case inDrag:
   DragWindow( window, the_event->where, &screenBits.bounds );
   break;
```

If the user clicks the mouse button in the content of an application window, the Toolbox routine SelectWindow() is called to activate that window. If the window was in the background before the mouse click, it will be brought to the foreground.

```
case inContent:
   SelectWindow( window );
   break;
```

Clicking in a window's go away box—its close box—brings about a call to the Toolbox function TrackGoAway(). TrackGoAway() will monitor the user's actions as the mouse is moved in and out of the go away box. If the user releases the mouse button while over an application window's go away box, the program will call DisposeWindow() to release the memory occupied by the window's WindowRecord.

```
case inGoAway:
   if ( TrackGoAway( window, the_event->where ) )
      DisposeWindow( window );
   break;
```

That covers a click of the mouse button—a mouseDown event. What about when a part of a window that was obscured is brought back into view? That's an updateEvt, and the Handle_One_Event() routine responds to that event type by first determining which window needs updating. That information is held in the message field of EventRecord variable `the_event`.

Next, the current port is saved, and the port belonging to the window that needs updating is made active. Calls to BeginUpdate() and EndUpdate() surround the code that does the actual updating. In this example, that code is handled by a call to a routine I've named Update():

```
case updateEvt:
   window = (WindowPtr)the_event.message;
   GetPort( &old_port );
   SetPort( window );
   BeginUpdate( window );
      EraseRgn( window->visRgn );
      Update( window );
   EndUpdate( window );
   SetPort( old_port );
   break;
```

The Update() routine will be application-specific. That is, there is no one set way of writing it; its contents depend entirely on what your application is doing. In all cases, the update routine should draw the entire contents of the window—whatever they are. In this example, I've simply chosen to have the window hold a rectangle and a line of text.

```
void  Update( WindowPtr  window )
{
   Rect  the_rect;

   SetRect( &the_rect, 10, 30, 80, 80 );
   MoveTo( 10, 20 );
   DrawString( "\pUpdated!" );
}
```

A Multiple-Windows Example Program

Before moving on to the handling of windows as objects, it may help to look at the source code for a complete application that handles multiple windows—without using object-oriented programming. The NoObjectWindows program demonstrates the window-handling techniques discussed to this point—techniques that don't deal with OOP.

NoObjectWindows opens two identical windows from within main(). Figure 10–2 shows what the two windows look like.

```
void  main( void )
{
   WindowPtr  the_window;

   Initialize_Toolbox();

 the_window = GetNewWindow( WIND_ID, nil, (WindowPtr)-1L );
 the_window = GetNewWindow( WIND_ID, nil, (WindowPtr)-1L );

   for ( ; ; )
      Handle_One_Event();
}
```

Notice that the WindowPtr variable `the_window` is declared inside main(), making it unavailable to the rest of the program. There is no single global WindowPtr variable that keeps track of either or both of the windows. When an event that involves a window occurs, information concerning *which* window the event took place in will be held in the message field of the EventRecord variable that WaitNextEvent() fills. Any routine that works with a window can use the information in the message field—cast to a WindowPtr—as a parameter. Figure 10–3 illustrates this.

In Figure 10–3, WaitNextEvent() fills the EventRecord variable `the_event` with information about a mouse click in one of the two windows.

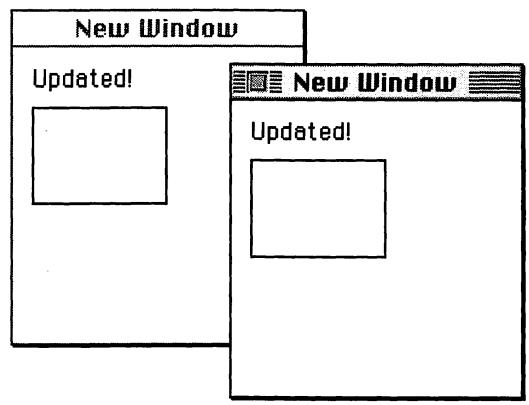


FIGURE 10–2 The output of the NoObjectWindows program

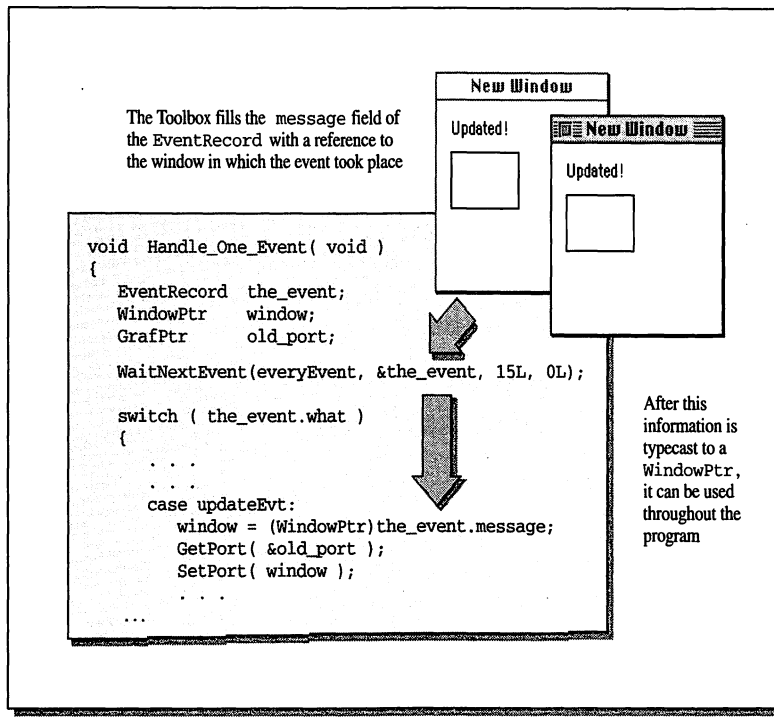


FIGURE 10–3 The message field of an EventRecord provides a reference to a clicked-on window.

Later, the message field of the EventRecord is used to give a local WindowPtr variable named `window` a value—the value being a pointer to the window in which the mouse click took place. A couple of lines later, this WindowPtr is used to make the port of the clicked-on window the active port—ready for updating later in the code.

When the NoObjectWindows program receives an update event, the Handle_One_Event() function uses the method shown in Figure 10–3 to determine which window needs updating. Then an updating routine named Update() is invoked to do the actual drawing to the window:

```
void  Update( WindowPtr  window )
{
   Rect  the_rect;
     SetRect( &the_rect, 10, 30, 80, 80 );
   FrameRect( &the_rect );
   MoveTo( 10, 20 );
   DrawString( "\pUpdated!" );
}
```

As you'll see when you run the program, this approach works well—but only when all windows have identical contents. The routine will draw to the proper window—thanks to the information supplied in the message field of the EventRecord—but it will always draw the same thing.

Unlike the NoObjectWindows program, few Mac programs have multiple windows with identical contents. So for most programs, simply distinguishing between windows is not enough; the program must also know how windows differ so that the proper text or graphics can be drawn. One way of doing this is to define certain types of windows—usually by giving each type a number:

```
#define    MY_GRAPHICS_WINDOW_TYPE    1
#define    MY_TEXT_WINDOW_TYPE        2
```

Then, each time a window is opened, the programmer will embed the number that represents the window's type in the refCon field of the window's WindowRecord. Figure 10–4 shows how this scheme works to differentiate between windows.

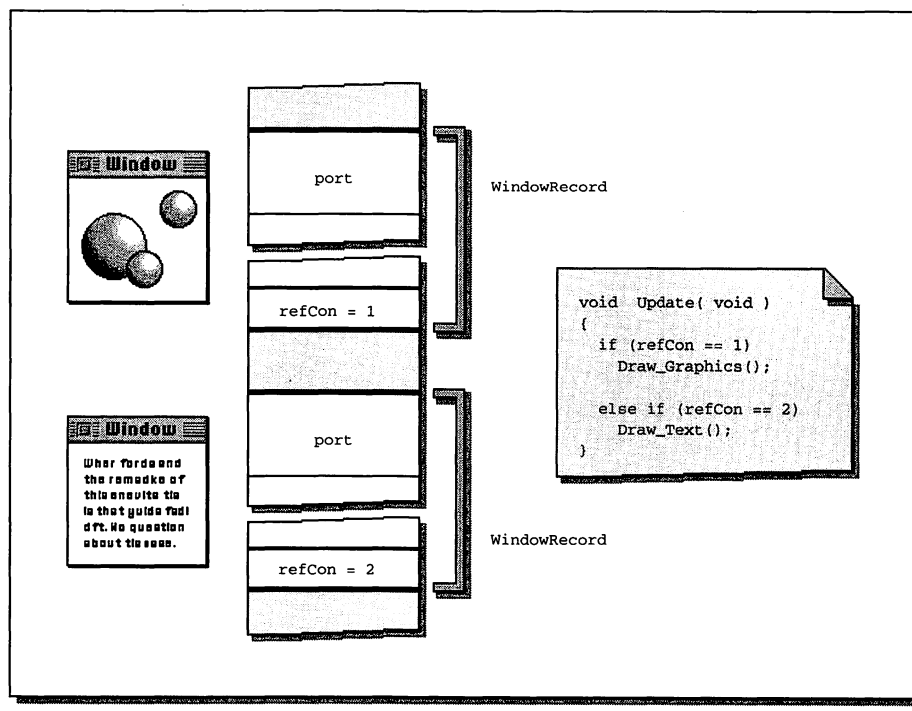


FIGURE 10–4 The refCon field of a WindowRecord can be used to keep track of window types.

In Figure 10–4, two open windows are shown on the left side of the figure. One holds graphics; the other holds text. At the center of the figure is a representation of a section of memory. Within the memory are two WindowRecords—one structure per window. The refCon fields of the WindowRecords have been set to distinguish the type of each of the two windows. On the right side of the figure is a simplified updating routine. This routine checks the value of the refCon for the window that needs updating. Based on this value, the Update() function calls the appropriate routine to draw or write to the window in need of updating.

The NoObjectWindows program doesn't use the above method for updating its windows. Nevertheless, the time you have just spent reading about using a WindowRecord's refCon field hasn't been wasted; you'll use this knowledge in the next section of this chapter.

NOTE

When you run NoObjectWindows, two windows will open—one on top of the other. Drag one window aside to see the second window.

You'll find the source code file for NoObjectWindows—along with the resource file that holds a single WIND resource—on the accompanying disk.

```
// ******************  NoObjectWindows.cp  ****************

//_____________________________________________________________
//                                        function prototypes

void   Initialize_Toolbox( void );
void   Handle_One_Event( void );
void   Handle_Mouse_Down( EventRecord * );
void   Update( WindowPtr );


//_____________________________________________________________
//                                        #define directives

#define     WIND_ID             128
```

```
//_____________________________________________________________
//                                                        main()

void  main( void )
{
   WindowPtr  the_window;

   Initialize_Toolbox();

   the_window = GetNewWindow( WIND_ID, nil,
                              (WindowPtr)-1L );
   the_window = GetNewWindow( WIND_ID, nil,
                              (WindowPtr)-1L );

   for ( ; ; )
      Handle_One_Event();
}


//_____________________________________________________________
//                                        initialize the Mac

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( 0L );
   FlushEvents( everyEvent, 0L );
   InitCursor();
}


//_____________________________________________________________
//                                        handle single event
```

```
void  Handle_One_Event( void )
{
   EventRecord  the_event;
   WindowPtr    window;
   GrafPtr      old_port;

   WaitNextEvent( everyEvent, &the_event, 15L, 0L );

   switch ( the_event.what )
 {
      case mouseDown:
         Handle_Mouse_Down( &the_event );
         break;

      case updateEvt:
         window = (WindowPtr)the_event.message;
         GetPort( &old_port );
         SetPort( window );
         BeginUpdate( window );
            EraseRgn( window->visRgn );
            Update( window );
         EndUpdate( window );
         SetPort( old_port );
         break;
   }
}


//_____________________________________________________________
//                          handle a click of the mouse

void  Handle_Mouse_Down( EventRecord *the_event )
{
   WindowPtr    window;
   short        the_part;

   the_part = FindWindow ( the_event->where, &window);

   switch ( the_part )
```

```
    {
        case inSysWindow:
            SystemClick ( the_event, window );
            break;

        case inDrag:
            DragWindow( window, the_event->where,
                        &screenBits.bounds );
            break;

        case inContent:
            SelectWindow( window );
            break;

        case inGoAway:
            if ( TrackGoAway( window, the_event->where ) )
            {
                DisposeWindow( window );
                ExitToShell();
            }
            break;
    }
}


//_____________________________________________________________
//                                              update a window

void  Update( WindowPtr  window )
{
    Rect  the_rect;

    SetRect( &the_rect, 10, 30, 80, 80 );
    FrameRect( &the_rect );
    MoveTo( 10, 20 );
    DrawString( "\pUpdated!" );
}
```

Representing Windows as Objects

Now that you know the basics of working with windows, you can apply many of the procedural programming techniques to OOP window handling. But, of course, you'll also have to learn a few additional tricks in order to get windows to properly behave as objects.

The Window Class

When designing an object, determine the data the object should hold and the actions that will be taken on the object. Figure 10–5 shows one model of how a window could be represented as an object.

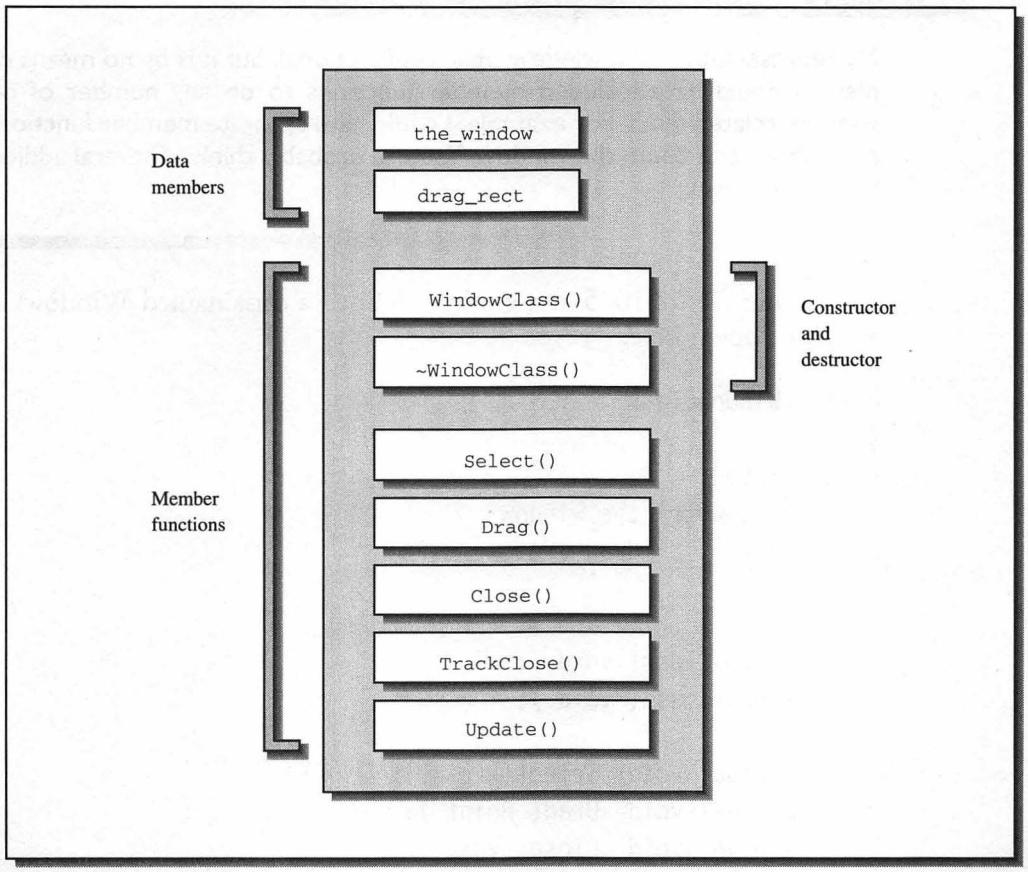


FIGURE 10–5 A figurative representation of a window as an object

My representation of a window object has two data members. A window object will need a window, of course. The data member `the_window` will be a WindowPtr that points to the object's window. There is a rectangle that defines the boundaries within which a window can be dragged on the screen. This rectangle is usually the entire desktop area of the monitor or monitors of the Mac on which the program is running. The second data member, `drag_rect`, will be a Rect variable that holds these drag boundaries.

The window object has seven member functions, two of which are the class constructor and destructor functions. Each of the other five handles a specific window-related task (the function names should tell you what those tasks are).

NOTE

My representation of a window object is functional, but it is by no means complete. I could have included member functions to do any number of other window-related tasks. For example, I could have separate member functions to hide, show, and zoom the window. You can probably think of several additional functions as well.

Using Figure 10–5 as a guide, I defined a class named WindowClass, which is shown here:

```
class  WindowClass
{
   protected:
      WindowPtr  the_window;
      Rect       drag_rect;

   public:
      WindowClass( void );
     ~WindowClass( void );

      virtual void  Select( void );
      virtual void  Drag( Point );
      virtual void  Close( void );
      virtual void  TrackClose( Point );
      virtual void  Update( void );
};
```

Windows and the Constructor Function

While constructor and destructor functions aren't always necessary, the nature of some class types makes their inclusion very helpful. In Chapter 5, I used a class that defined a window object as an example. In that chapter I spoke in general terms. Now I can be more specific. The data member `the_window` will serve as a pointer to the graphics port of a window that is associated with a WindowClass object. When the new operator is used to create a WindowClass object, memory will be allocated for the object. Here's the code that would accomplish that:

```
WindowClass  wind;

wind = new  WindowClass;
```

On the left side of Figure 10–6, you'll see the above code snippet, along with a look at how a section of memory has been affected by the code—it holds an object. Notice that memory is allocated for the data

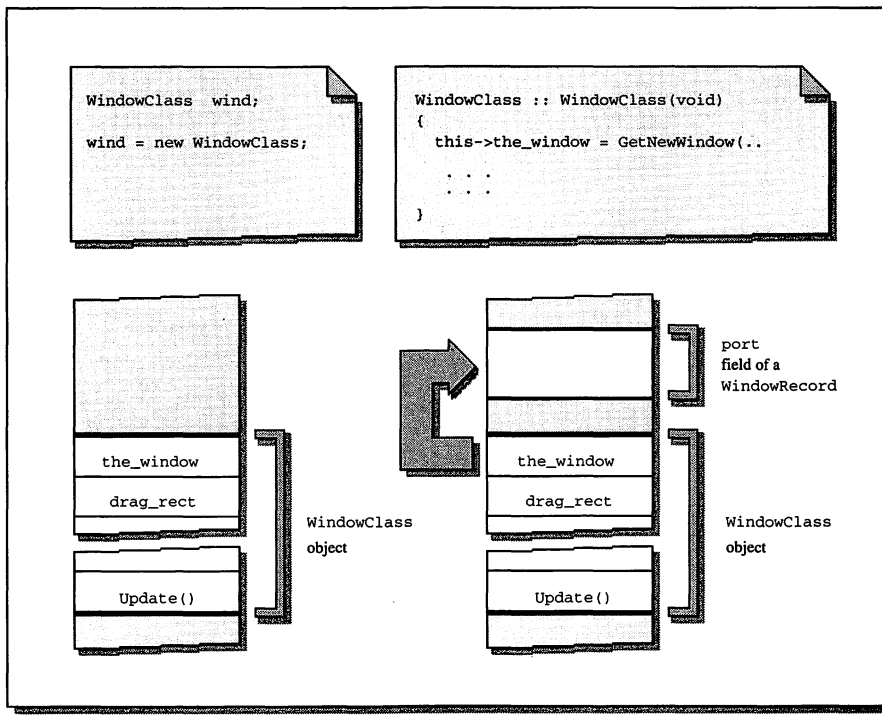


FIGURE 10–6 Memory allocation for a WindowRecord isn't provided for by the new operator.

member the_window, but no memory is allocated for a WindowRecord. The data member the_window is only a pointer to a window—not a window. Only after a call to one of the Toolbox functions NewWindow() or GetNewWindow() is memory for the WindowRecord allocated. On the right side of Figure 10–6, you can see that this is exactly what is happening. The code that allocates the WindowRecord memory—GetNewWindow()—has been placed in the constructor function of the WindowClass class. That way, when the new operator is used to create a new WindowClass object, the constructor will be automatically invoked, and a new window will be loaded into memory and placed on the screen.

Below is the complete constructor function for the WindowClass. The first line in the function body allocates memory for the WindowRecord. The second line sets the WindowRecord's refCon field to a unique value that will enable the program to distinguish this one window from all others that may be on the screen. The address of the newly created object serves as this unique value.

```
WindowClass :: WindowClass( void )
{
   this->the_window = GetNewWindow( WIND_ID, nil,
                                     (WindowPtr)-1L );

   SetWRefCon( this->the_window, (long)this );
}
```

Earlier in this chapter, you read that Mac programs written in a procedural language like Pascal or C often use the refCon field of a window's WindowRecord to store identifying information about the window. That's what's happening in the WindowClass constructor. The Toolbox function SetWRefCon() accepts a pointer to a window and a long integer as its parameters. It then sets the refCon field of the WindowRecord pointed to by the first parameter to the value passed in as the second parameter.

Let's look at the two parameters I'm passing to SetWRefCon(). The first is the WindowPtr returned by the constructor's call to GetNewWindow(). The second is a little trickier. In Chapter 5, you saw that the *this* keyword is used as a reminder that a member function is working with an object—whichever object invoked it. The *this* keyword serves as a pointer to the object being worked with. So when *this* is used on its own, it is referring to the address of the object being worked with, as shown in Figure 10–7.

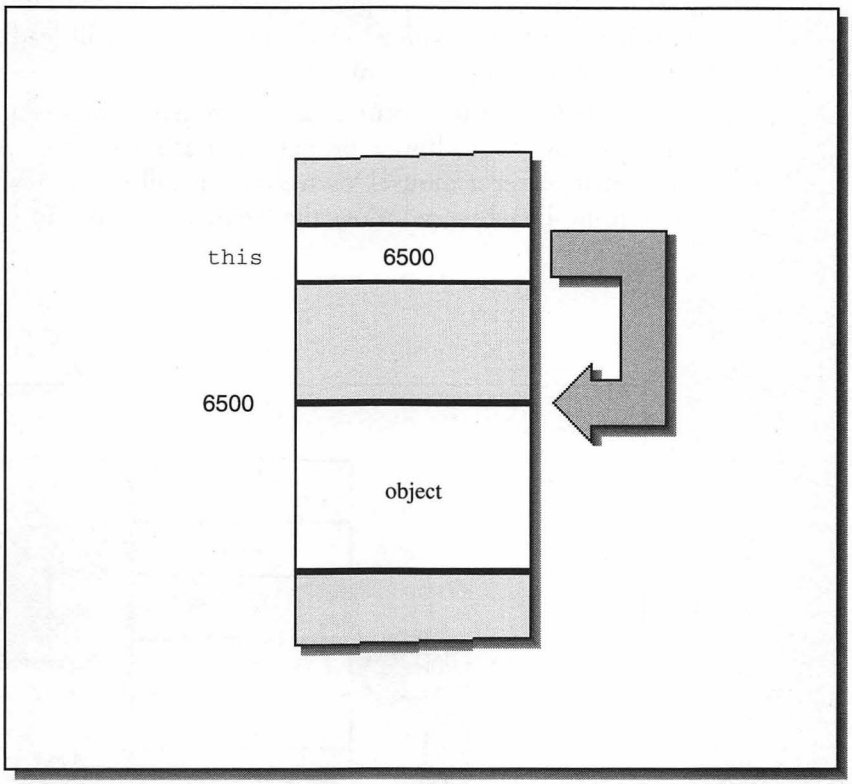


FIGURE 10–7 The *this* keyword is a pointer to an object; it holds the address of an object.

Figure 10–7 shows that *this* is a pointer—an address. In the constructor, the pointer is typecast to a long integer. That makes it a number that is acceptable for use as a parameter in the call to SetWRefCon():

```
SetWRefCon( this->the_window, (long)this );
```

What is the net effect of the above call to SetWRefCon()? The refCon field is set to the same value as the address of the object. Because each object will have its own place in memory and its own starting address, this approach ensures that the refCon field will always hold a unique number—a number that will never appear in the refCon field of any other WindowRecord. Figure 10–8 illustrates this.

Setting the refCon gives an object its own unique reference number, but how does a program make use of this information? Any time a window-

related event occurs, a program should look at the value of the refCon of the window that was involved in the event. That will lead the program to the object involved in the event.

Let's look at the occurrence of a mouseDown event in the content area of a window to see how a program determines which object is involved in the event. After a mouseDown event, a call to FindWindow() is made to determine in which window the event occurred and in which part of the

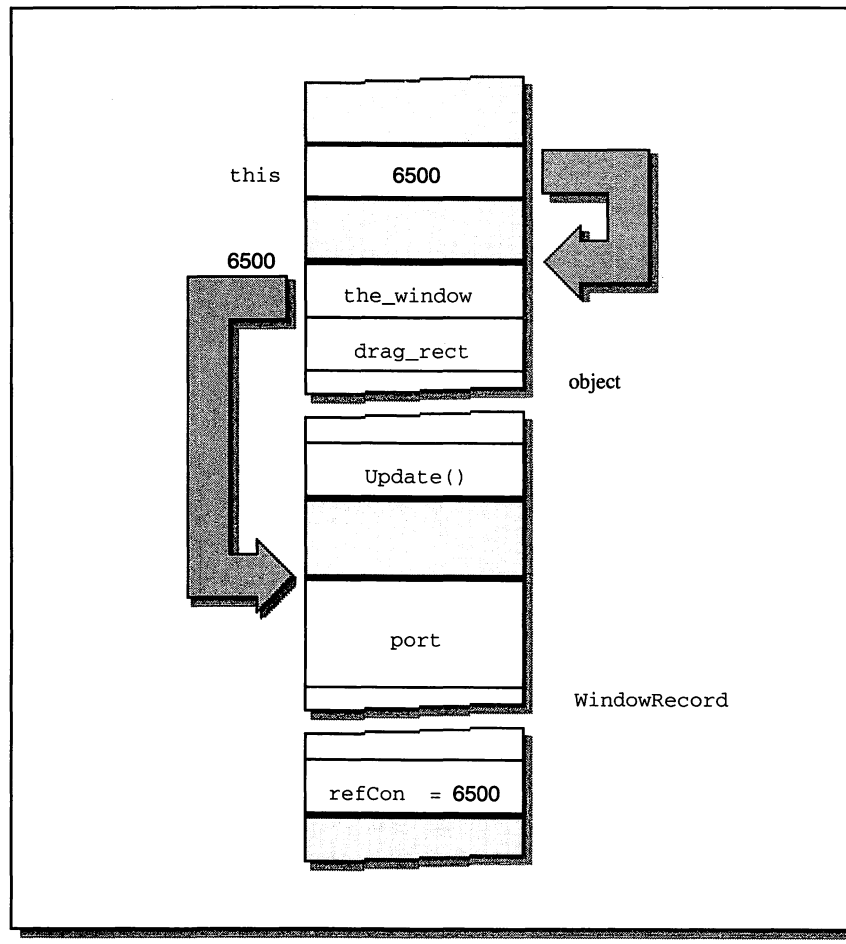


FIGURE 10–8 The refCon field of a WindowRecord can be used to hold the address of an object.

window the mouse button was pressed. This line of code is from Handle_Mouse_Down():

```
the_part = FindWindow ( the_event->where, &window);
```

After the call to FindWindow(), the variable window—a WindowPtr—will point to the window in which the mouse click occurred. What I'll have to do is examine the refCon of the WindowRecord of this window to see which object it belongs to. The following snippet does that:

```
long  wind_ID;

wind_ID = ((WindowPeek)window)->refCon;
```

What the above code does is typecast the WindowPtr variable window to a WindowPeek. Recall from earlier in this chapter that a WindowPeek differs from a WindowPtr in that it allows you to access all of the fields of a WindowRecord—not just the port field. Once the WindowPtr is turned into a WindowPeek, I can access the refCon field. The result is a long integer, which I've assigned to the variable `wind_ID`.

The value that was in the refCon is the address of the object. Refer back to Figure 10–8 to verify that. If I typecast this refCon value to an object pointer, I'll have a pointer to the object that's to be involved in the updating:

```
long         wind_ID;
WindowClass  *temp;

wind_ID = ((WindowPeek)wind)->refCon;

temp = (WindowClass *)wind_ID;
```

Every time a window-related event occurs, I'll want to execute the above code so that I can determine which object is involved. To simplify things, I'll roll it into a function called Which_Object(). This function returns a pointer to the object involved in the window-related event.

```
WindowClass  *Which_Object( WindowPtr wind )
{
   long         wind_ID;
   WindowClass  *temp;

   wind_ID = ((WindowPeek)wind)->refCon;
```

```
    temp = (WindowClass *)wind_ID;

    return ( temp );
}
```

When a window-related event occurs, I'll call Which_Object(), passing in a pointer to the window that received the event. Which_Object() will typecast the WindowPtr to a WindowPeek, access the refCon field of the window's WindowRecord, and then typecast the resulting long integer value to an object pointer. The result? A pointer to the object to which the window belongs. This next code snippet shows how a mouse click in a window's content area would result in the Select() member function being invoked for the proper object. Figure 10–9 illustrates this.

```
void  Handle_Mouse_Down( EventRecord *the_event )
{
    WindowPtr    window;
    short        the_part;
    WindowClass  *wind_obj;

    the_part = FindWindow ( the_event->where, &window );

    switch ( the_part )
    {
        case inContent:
          wind_obj = Which_Object( window );
          wind_obj->Select();
          break;
    }
}

WindowClass  *Which_Object( WindowPtr wind )
    long         wind_ID;
    WindowClass  *temp;

    wind_ID = ((WindowPeek)wind)->refCon;

    temp = (WindowClass *)wind_ID;
      return ( temp );
}
```

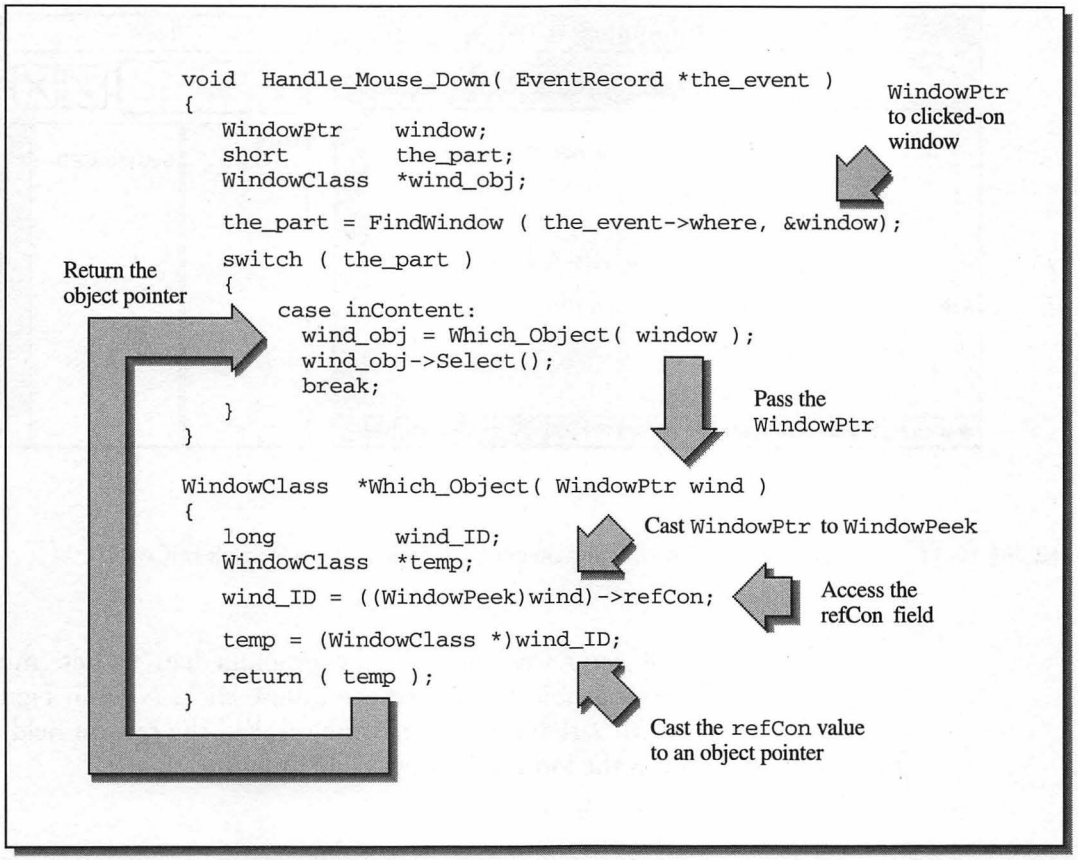


FIGURE 10–9 After a mouse click occurs in a window, determine which object owns the window.

Verifying That Objects Are Distinguishable

To verify that the technique of placing an object's address in the refCon field works, I'll step through a few lines of a program that uses it. I've set a breakpoint in the constructor function of the WindowClass and then used the new operator to create a WindowClass object. The creation of the object invoked the constructor, and the program stopped at the breakpoint. In Figure 10–10, the value of *this*—the pointer to the object—is shown in the Data window of the debugger.

Next, I set a breakpoint in the Which_Object() function and then clicked on the Go button. When I clicked the mouse on the window of my

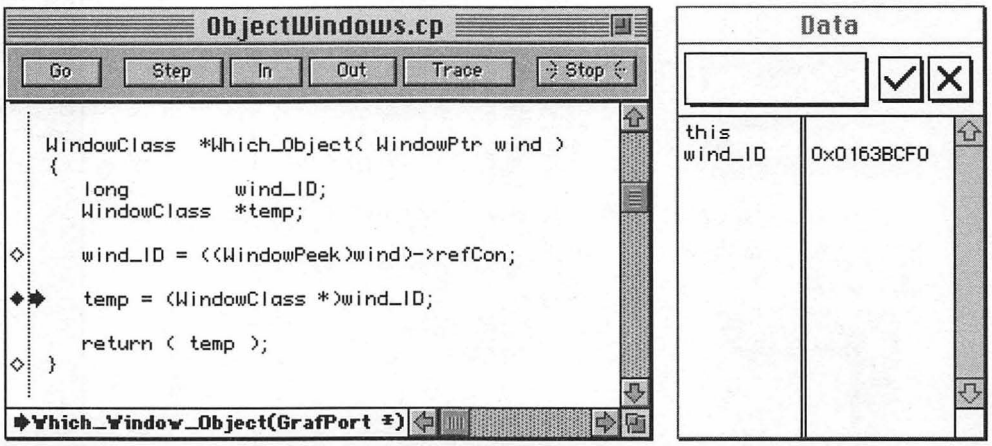


FIGURE 10–11 Identifying the window-owning object by accessing the window's refCon

program, Which_Object() was invoked; the program had to determine which of possibly several objects received the mouse click. Note in Figure 10–11 that the `wind_ID` variable, which has the value of the refCon field of the object's window, is the same as the *this* value in Figure 10–10.

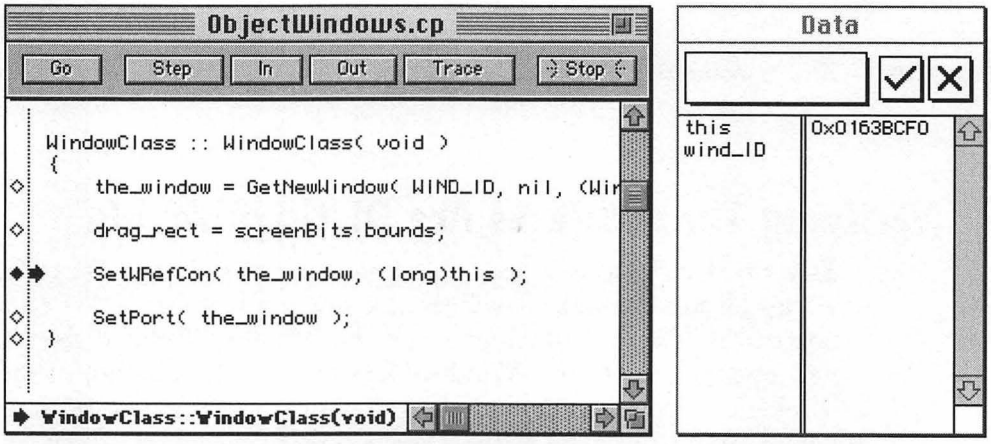


FIGURE 10–10 The refCon of a window is set to the address of the object that owns the window.

Windows and Events

In response to an event, a program that uses window objects will want to determine which object owns the window involved with the event. You can see in the following snippet that Handle_Mouse_Down() calls Which_Object() whenever there's a mouse click in the drag region, content area, or go away box of a window.

```
case inDrag:
   wind_obj = Which_Object( window );
   wind_obj->Drag( the_event->where );
   break;

case inContent:
   wind_obj = Which_Object( window );
   wind_obj->Select();
   break;

case inGoAway:
   wind_obj = Which_Object( window );
   wind_obj->TrackClose( the_event->where );
   break;
```

After the object that owns the window is determined, it's just a matter of sending that object the appropriate message. For example, a click on the drag bar of a window will result in a call to the owning object's Drag() member function. Here's a look at how that function could be implemented:

```
void  WindowClass :: Drag( Point where )
{
   DragWindow( this->the_window, where,
               &this->drag_rect );
}
```

The Drag() function simply calls the Toolbox routine DragWindow(), passing a WindowPtr to the object's window and the object's drag_rect data member. Other member functions are just as simple as Drag():

```
void  WindowClass :: Select( void )
{
   SelectWindow( this->the_window );
}
```

```
void  WindowClass :: Close( void )
{
   delete this;
}


void  WindowClass :: TrackClose( Point where )
{
   if ( TrackGoAway( this->the_window, where ) )
      this->Close();
}
```

Note that the Close() function need only delete the memory allocated for the object; it doesn't have to delete the extra memory allocated for the WindowRecord of the object's window. When the delete operator is used on the object, the object's destructor will be called to take care of cleaning up the WindowRecord memory:

```
WindowClass :: ~WindowClass( void )
{
   DisposeWindow( this->the_window );
   ExitToShell();
}
```

After disposing of the window, the destructor ends the program by calling ExitToShell(). In the next chapter, you'll see a more Mac-like way of ending a program—through the use of a Quit menu item.

An Example of Windows as Objects

The ObjectWindows program presented here does the same thing that the NoObjectWindows program does. ObjectWindows opens two windows—one on top of the other—and allows you to select either one and drag it about the screen. Clicking in the close box of either window closes it and ends the program. Figure 10–12 shows what you'll see when you run ObjectWindows.

The differences between ObjectWindows and NoObjectWindows is not in what the programs do but how they do it. ObjectWindows uses the

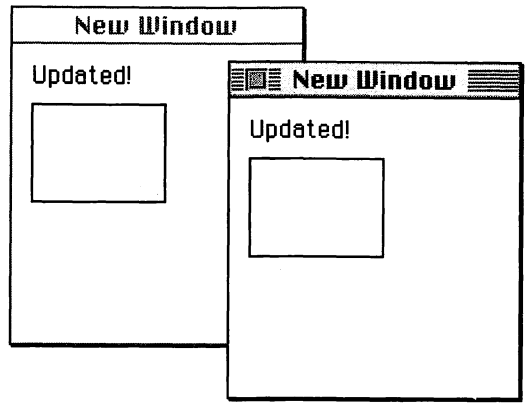


FIGURE 10–12 The output of the ObjectWindows program

object-oriented techniques discussed in this chapter to create the windows, keep track of them, and respond to actions performed on them.

ObjectWindows uses the technique of using the refCon as a means of keeping track of each open window. That allows the program to properly respond to a mouse click that occurs anywhere in a window. What ObjectWindows can't do is update windows that have different window contents, which is the same shortcoming that the NoObjectWindows program had. The reason for this is simple: although the program knows which window needs updating and draws to the correct window, it always draws the same thing—whatever has been defined in the Update() member function. This serious flaw will be remedied in Chapter 11, which is devoted to the development of an OOP program that treats each window individually and includes the contents of the window.

As always, you'll find the source file and resource file for the program on the accompanying disk.

```
// ***************** ObjectWindows.cp ******************

//_____________________________________________________
//                                    forward references

class    WindowClass;
```

```
//_____________________________________________________
//                                    function prototypes

void   Initialize_Toolbox( void );
void   Handle_One_Event( void );
void   Handle_Mouse_Down( EventRecord * );
WindowClass  *Which_Object( WindowPtr );

//_____________________________________________________
//                                    #define directives

#define     WIND_ID             128


//_____________________________________________________
//                                     class definitions

class WindowClass
{
   protected:
      WindowPtr  the_window;
      Rect       drag_rect;

   public:
      WindowClass( void );
     ~WindowClass( void );

      virtual void  Select( void );
      virtual void  Drag( Point );
      virtual void  Close( void );
      virtual void  TrackClose( Point );
      virtual void  Update( void );
};


//_____________________________________________________
//                              member function definitions

WindowClass :: WindowClass( void )
```

```
{
   this->the_window = GetNewWindow( WIND_ID, nil,
                                     (WindowPtr)-1L );

   this->drag_rect = screenBits.bounds;


   SetWRefCon( this->the_window, (long)this );
   SetPort( this->the_window );
}


WindowClass :: ~WindowClass( void )
{
   DisposeWindow( this->the_window );
   ExitToShell();
}


void  WindowClass :: Select( void )
{
   SelectWindow( this->the_window );
}


void  WindowClass :: Drag( Point where )
{
   DragWindow( this->the_window, where,
               &this->drag_rect );
}


void  WindowClass :: Close( void )
{
   delete this;
}


void  WindowClass :: TrackClose( Point where )
```

```
{
   if ( TrackGoAway( this->the_window, where ) )
      this->Close();
}

  void  WindowClass :: Update( void )
{
   WindowPtr  save_port;
   Rect  the_rect;

   GetPort( &save_port );
   SetPort( this->the_window );

   BeginUpdate( this->the_window );
      SetRect( &the_rect, 10, 30, 80, 80 );
      FrameRect( &the_rect );
      MoveTo( 10, 20 );
      DrawString( "\pUpdated!" );
   EndUpdate( this->the_window );

   SetPort( save_port );
}


//____________________________________________________________
//                                                      main()

void  main( void )
{
   WindowClass  *window;

   Initialize_Toolbox();

   window = new  WindowClass;
   window = new  WindowClass;

   for ( ; ; )
      Handle_One_Event();
}
```

```
//_____________________________________________________
//                                    initialize the Mac

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();
   TEInit();
   InitDialogs( 0L );
   FlushEvents( everyEvent, 0L );
   InitCursor();
}


//_____________________________________________________
//                                  handle single event

void  Handle_One_Event( void )
{
   EventRecord  the_event;
   WindowClass  *wind_obj;

   WaitNextEvent( everyEvent, &the_event, 15L, 0L );

   switch ( the_event.what )
   {
      case mouseDown:
         Handle_Mouse_Down( &the_event );
         break;

      case updateEvt:
         wind_obj = Which_Object( (WindowPtr)
                                  the_event.message );
         wind_obj->Update();
         break;
   }
}
```

```
//_____________________________________________________
//                              handle a click of the mouse

void  Handle_Mouse_Down( EventRecord *the_event )
{
   WindowPtr    window;
   short        the_part;
   WindowClass  *wind_obj;

   the_part = FindWindow ( the_event->where, &window);

   switch ( the_part )
   {
      case inSysWindow:
         SystemClick ( the_event, window );
         break;

      case inDrag:
         wind_obj = Which_Object( window );
         wind_obj->Drag( the_event->where );
         break;

      case inContent:
         wind_obj = Which_Object( window );
         wind_obj->Select();
         break;

      case inGoAway:
         wind_obj = Which_Object( window );
         wind_obj->TrackClose( the_event->where );
         break;
   }
}


//_____________________________________________________
//           determine which window object to work with
```

```
WindowClass  *Which_Object( WindowPtr wind )
{
   long          wind_ID;
   WindowClass   *temp;

   wind_ID = ((WindowPeek)wind)->refCon;

   temp = (WindowClass *)wind_ID;

   return ( temp );
}
```

Chapter Summary

Mac programmers and users alike tend to think of a window as an object—something that can be opened and closed, drawn to or written in. So the window serves as a good element of the Macintosh user interface to model as an object-oriented object.

The information that defines a window is held in a WindowRecord data structure. A WindowPtr is a pointer that points to the first member of a WindowRecord—the window's graphics port. If you want to access other members of a WindowRecord you'll use a WindowPeek rather than a WindowPtr. One reason to access a WindowRecord member is to mark each new window that opens with its own ID. The ID will relate a particular window with a particular object. You can do this by using a WindowPeek to access the refCon member of a window's WindowRecord. Later, when a window-related event occurs, you can again access this member of the involved window to find out which object the window belongs to.



Chapter 11

A Complete Example

At this point you've learned the basics of C++ and object-oriented programming. And a little more. OOP concepts like returning objects from functions and dynamic binding are powerful—and often daunting—skills to master. Learning these techniques individually and seeing them in action in short example programs helped ease the transition from procedural programming to object-oriented programming. Now, it's time to apply all of the things you've learned to the creation of a single Macintosh OOP application.

DerivedWindows: A Complete OOP Example

The example programs in the previous two chapters have used the Rectangle class, neglecting the pet shop owner and his unfinished animal database program. In this chapter, I'll apply all the concepts covered in the previous



FIGURE 11–1 The File menu for the DerivedWindows program

chapters—and a few new ones—to the unfinished animal database. The result will be a program called DerivedWindows.

What the Program Does

The complete source code listing for DerivedWindows appears at the end of this chapter. You won't have to be familiar with the source code to be able to see what the program is capable of doing; I can cover that up front.

DerivedWindows displays three menus, but only one—the File menu—will be of real interest. Figure 11–1 shows the menu items in that menu.

Chapter 9 discussed using dialog boxes to solicit object information from the user. Selecting the New Picture item will display the dialog box shown in Figure 11–2. Here the user can choose which one of two pictures to display in a new window.

After the user selects one of the radio buttons, a click on the OK button dismisses the dialog box and opens a window that displays the

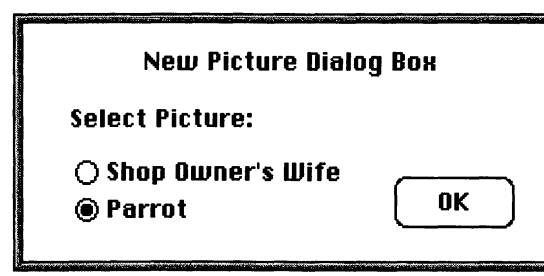


FIGURE 11–2 Selecting a picture to display in the DerivedWindows program

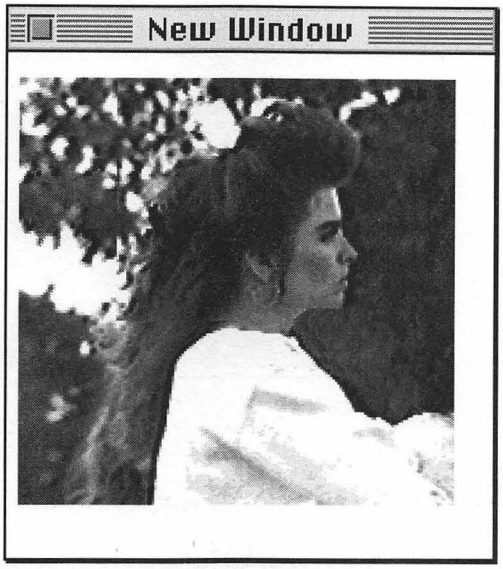


FIGURE 11–3 A picture window in the DerivedWindows program

appropriate picture. The radio button labeled Shop Owner's Wife will display the picture shown in Figure 11–3. While the typical businessman keeps a picture of his wife on his desk, this is the digital age, and the shop owner prefers to keep a picture of his wife readily available on his computer.

If the radio button labeled Parrot was turned on at the time the dialog box was dismissed, a window with the picture shown in Figure 11–4 will be displayed. While this option might not please the shop owner's wife, we are after all, working on a database to hold information about pets.

In its current state the DerivedWindows program is capable of displaying only a picture of the pet shop owner's favorite pet—the parrot. An enhancement to the program would be a series of radio buttons—or perhaps a scrollable list—that would allow the owner to display the picture of any one of a number of exotic animals that he can order but doesn't keep in stock. An inquiring customer could then be quickly shown a picture of any one of those animals.

Selecting New Pet from the File menu opens the dialog box shown in Figure 11–5. In this dialog box, the user can enter the name of an animal and its selling price.

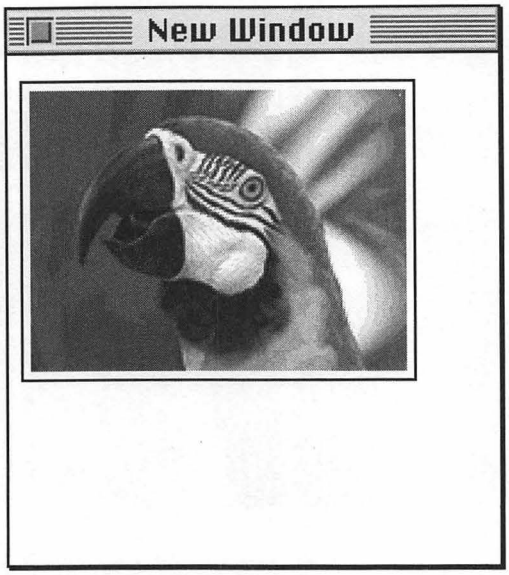


FIGURE 11–4 Another picture window from the DerivedWindows program

After clicking on the OK button in the New Animal dialog box, the dialog box is dismissed, and a new window that displays the entered information opens. An example of such a window is shown in Figure 11–6.

Perhaps the best feature of the program is that it has the ability to open as many windows as the user likes. Each window can hold either of the two pictures or any animal information the user enters. And each window will be properly updated as windows are covered and uncovered. Figure 11–7 shows the program running with five open windows.

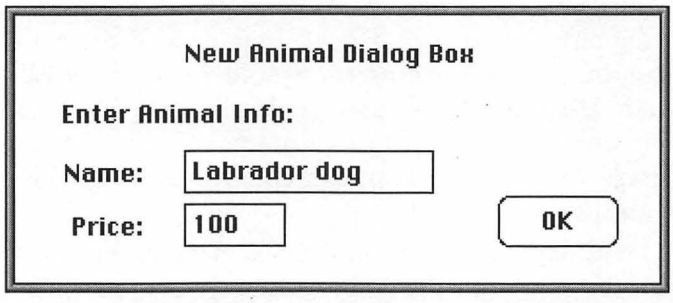


FIGURE 11–5 Entering animal information in the DerivedWindows program

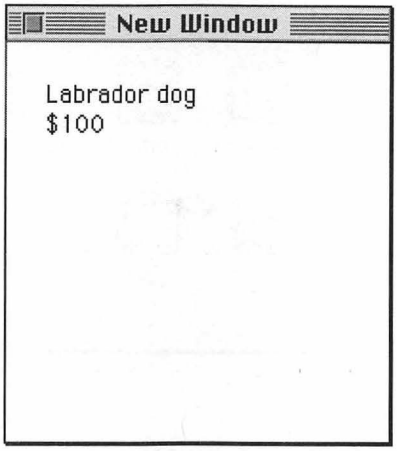


FIGURE 11–6 A pet information window displayed in the DerivedWindows program

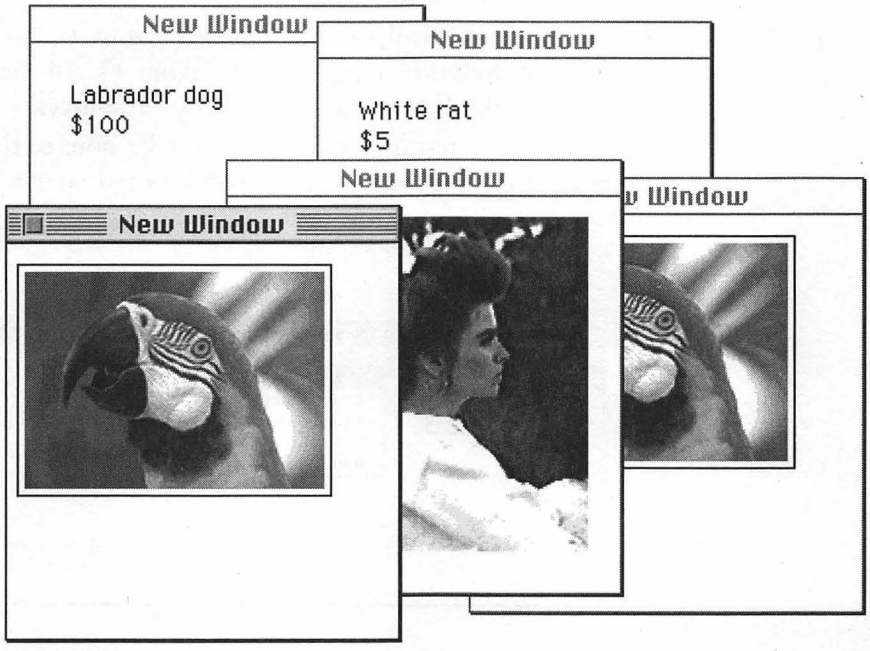


FIGURE 11–7 The DerivedWindows program allows any number of windows to be displayed

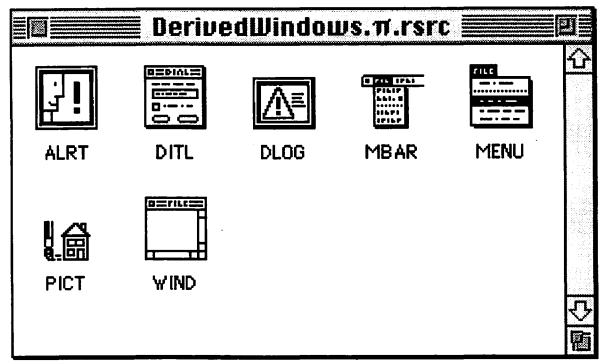


FIGURE 11–8 The resource file for DerivedWindows

The DerivedWindows Resources

DerivedWindows needs several resource types to run properly. Figure 11–8 shows the resource file for the program.

DerivedWindows displays three menus—thus it has three MENU resources. They're shown in Figure 11–9. Figure 11–10 shows the MBAR resource that holds the IDs of the three MENU resources.

While I have the resource file open, I'll take note of the resource IDs and the menu items. This information will be used later when I write the program's source code:

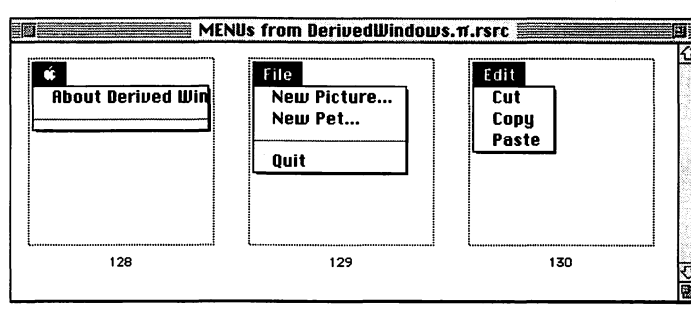


FIGURE 11–9 The three MENU resources that will be displayed in the menu bar

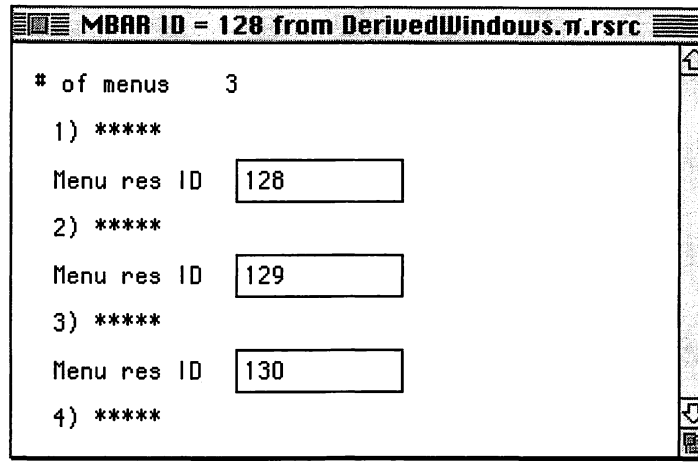


FIGURE 11–10 The MBAR resource that defines the menu bar

```
#define     MENU_BAR_ID          128

#define     APPLE_MENU_ID        128
#define     SHOW_ABOUT_ITEM        1
#define     FILE_MENU_ID         129
#define     NEW_PICT_ITEM          1
#define     NEW_PET_ITEM           2
// item 3 is a dashed line         3
#define     QUIT_ITEM              4
#define     EDIT_MENU_ID         130
```

Earlier you saw that DerivedWindows displays two dialog boxes. The first allows the user to select which picture a window will display. The DITL resource for that dialog is shown in Figure 11–11. The DITL resource for the second dialog box—the one that allows the user to enter information about an animal—is shown in Figure 11–12.

Along with the DITL items, the program will, of course, have two DLOG resources, the IDs of which will match those of the DITL resources they use. Again, I've taken this opportunity to take note of the resource IDs.

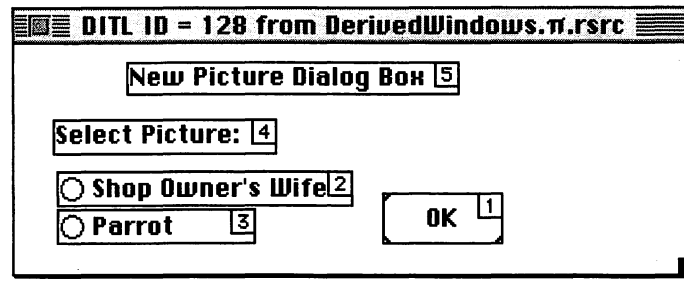


FIGURE 11–11 The DITL that defines the items in the New Picture dialog box

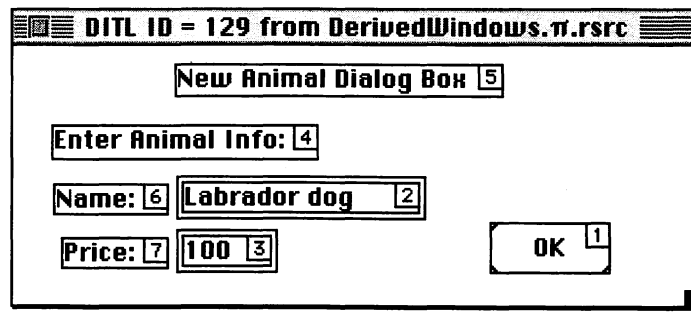


FIGURE 11–12 The DITL that defines the items in the New Pet dialog box

```
#define     PICT_DLOG_ID          128
#define     OK_BUTTON               1
#define     WIFE_BUTTON             2
#define     BIRD_BUTTON             3

#define     PET_DLOG_ID           129
//          OK_BUTTON               1    defined above
#define     NAME_ITEM               2
#define     PRICE_ITEM              3
```

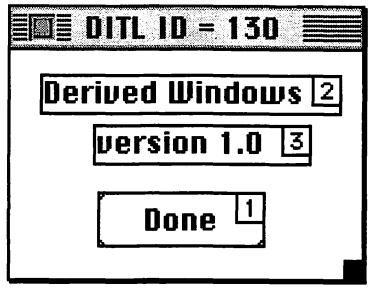


FIGURE 11–13 The DITL that defines the items in the About alert

There's one last DITL that the program requires. The About menu item in the Apple menu displays an alert with information about the program. I've kept that information short and simple, as you can see from the DITL shown in Figure 11–13. The ALRT resource has an ID of 130, as does the DITL it uses:

```
#define     ABOUT_ALRT_ID          130
```

The last resources the program needs are the PICT resources that are used to display pictures in the program's windows. DerivedWindows has just two; you may want to modify the program to use several more. Figure 11–14 shows the two PICT resources, and I've listed their IDs here:

```
#define     WIFE_PICT_ID           128
#define     BIRD_PICT_ID           129
```

The resource file is completed by the addition of a standard WIND resource.

The DerivedWindow Classes

DerivedWindows defines three classes. The first is the WindowClass developed in Chapter 10. There's one addition to this class—a member function named Draw(). The purpose of this function will be described later in this chapter.

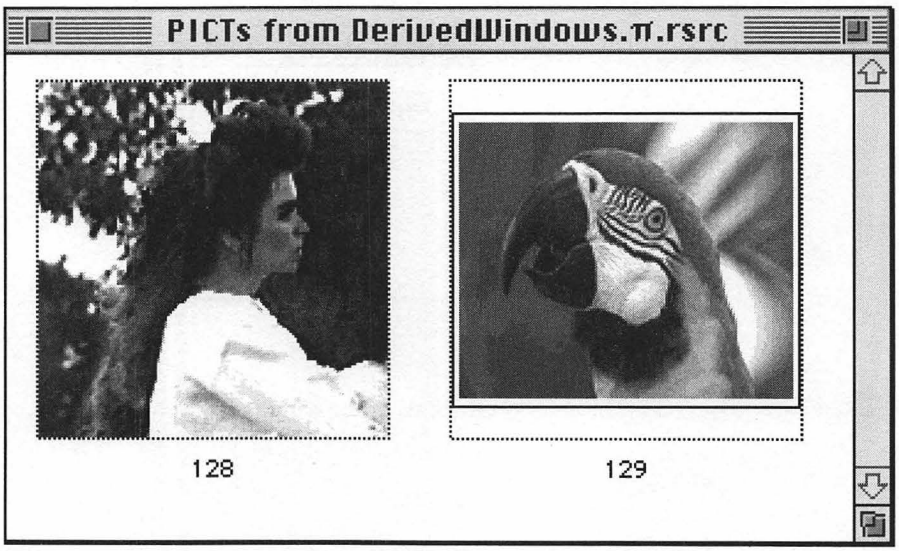


FIGURE 11–14 The PICT resources for the DerivedWindows program

```
class  WindowClass
{
   protected:
      WindowPtr  the_window;
      Rect       drag_rect;

   public:
      WindowClass( void );
     ~WindowClass( void );

      virtual void  Select( void );
      virtual void  Drag( Point );
      virtual void  Close( void );
      virtual void  TrackClose( Point );
      virtual void  Update( void );
      virtual void  Draw( void );
};
```

Here, WindowClass will be used as an abstract class. From WindowClass, two classes are derived—PetWindow and PictWindow. PetWindow is similar to the Animal base class used earlier in the book.

```
class  PetWindow : public  WindowClass
{
   protected:
      Str255  type;
      Str255  cost;

   public:
      void  Set_Type( Str255 );
      void  Get_Type( Str255 );
      void  Set_Cost( Str255 );
      void  Get_Cost( Str255 );
      void  Draw( void );
};
```

Because PetWindow is derived from WindowClass, it will have its own `the_window` and `drag_rect` data members, and it will inherit all of the WindowClass member functions. That means objects of the PetWindow class will have all the functionality provided by WindowClass as well as the ability to store and use information about a pet. The data that PetWindow keeps track of are the type of pet and the cost of the pet. The PetWindow member functions—except for the Draw() function—exist to set and get the data members. Draw() is defined in the base class, so it is inherited and overridden by PetWindow. It will be used in window updating—as you'll see a little later on.

NOTE

Gone are the GoodPet and BadPet classes developed earlier. While they were useful for the simpler examples in the early chapters, those two classes were too similar to one another to really warrant separate class definitions. If I want to keep track of the suitability of a pet, I can just add another data member to the PetWindow class at a later time.

When an object of the PetWindow class is created, its inherited `the_window` data member will point to a window. The program will be able to set the values of the `type` and `cost` data members by sending the object

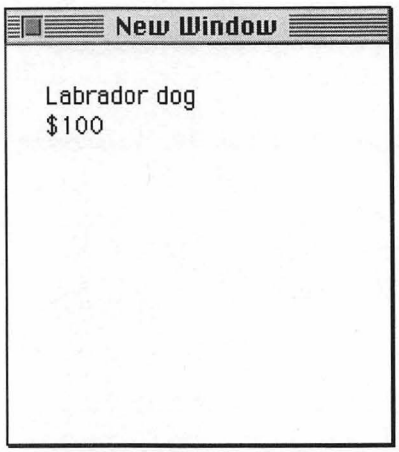


FIGURE 11–15 A typical pet information window created by a PetWindow object

Set_Type() and Set_Cost() messages. When it comes time to write the information to the object's window, the object will be sent Get_Type() and Get_Cost() messages. Then a Draw() message will be sent to the object to write this information to the object's window. The result will be similar to the window pictured in Figure 11–15.

The second class derived from the WindowClass is the PictWindow class. Like the PetWindow class, it inherits all of the data and functions of the WindowClass. That means that a PictWindow object will have its own window. Here's a look at

the PictWindow class:

```
class  PictWindow : public  WindowClass
{
   protected:
      short  pict_ID;

   public:
      void  Set_Pict_ID( short );
      void  Draw( void );
};
```

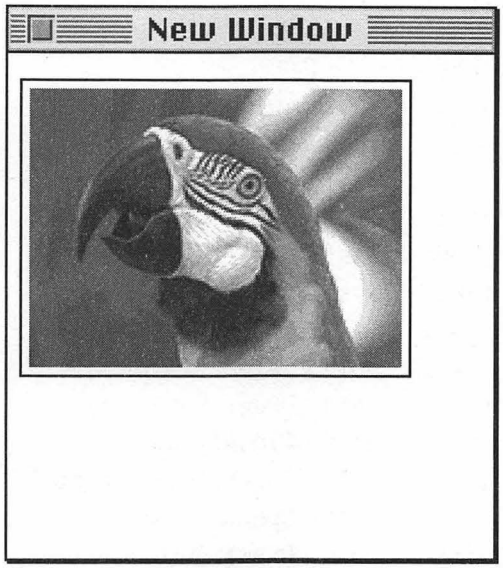


FIGURE 11–16 A typical picture window created by a PictWindow object

The PictWindow has a single data member—a short integer named `pict_ID`, which will hold the ID of a PICT resource. Earlier you saw that the resource file for the DerivedWindow project holds two PICT resources. A PictWindow object will be able to display either one of those pictures. To do so, the program sends a Set_Pict_ID() message to a PictWindow object to set the `pict_ID` to the ID of one of the two PICT resources. Then a Draw message is sent to the object to load this picture into memory and draw it to the object's window. The result will be a window with a picture in it, as shown in Figure 11–16.

The WindowClass Member Functions

You saw the definitions for the WindowClass member functions in Chapter 10, so I'll save a little paper by not repeating them here. If you feel inclined, you can flip ahead several pages to the source code listing for a look at them. What I will do, though, is repeat the definition of the WindowClass so you can see the names of each of those functions:

```
class  WindowClass
{
   protected:
      WindowPtr  the_window;
      Rect       drag_rect;

   public:
      WindowClass( void );
     ~WindowClass( void );

      virtual void  Select( void );
      virtual void  Drag( Point );
      virtual void  Close( void );
      virtual void  TrackClose( Point );
      virtual void  Update( void );
      virtual void  Draw( void );
};
```

One of the WindowClass functions has changed from the last time you saw it—but only slightly. In Chapter 10, you saw that the Update() function was responsible for drawing the contents of a WindowClass object's window. Here the function prepares for drawing (as last chapter's version did) but doesn't perform the actual drawing. Instead it invokes another member function—Draw():

```
void  WindowClass :: Update( void )
{
   WindowPtr  save_port;
   Rect  the_rect;

   GetPort( &save_port );
   SetPort( this->the_window );

   BeginUpdate( this->the_window );
      this->Draw();
   EndUpdate( this->the_window );

   SetPort( save_port );
}
```

The Draw() member function wasn't present in the Chapter 10 version of the WindowClass. Here's the definition for Draw():

```
void  WindowClass :: Draw( void )
{
}
```

No, it's not a misprint. The function body is intentionally empty. The Draw() function, like all of the WindowClass functions, is inherited by both the PetWindow class and the PictWindow class. But, unlike the other WindowClass functions, these derived classes override the function. Each supplies its own version of what Draw() does. And because the WindowClass is an abstract class—meaning no objects are ever created from it—there's no need to have it do anything.

IMPORTANT

Because Draw() is in the list of WindowClass member functions, a definition must appear for it—even if it is a definition that doesn't do anything.

The PictWindow Member Functions

Now let's turn to the member function of the PictWindow class. Here's another look at that class definition:

```
class  PictWindow : public  WindowClass
{
   protected:
      short  pict_ID;

   public:
      void  Set_Pict_ID( short );
      void  Draw( void );
};
```

Set_Pict_ID() does just what its name implies—it sets the PictWindow data member `pict_ID` to whatever value is passed to the function:

```
void  PictWindow :: Set_Pict_ID( short  the_ID )
{
   this->pict_ID = the_ID;
}
```

Once a PictWindow object has its `pict_ID` data member set, the object can draw the picture to the object's window through a call to Draw(). Here's what the PictWindow version of this function looks like:

```
void  PictWindow :: Draw( void )
{
   Rect        pict_rect;
   PicHandle   pict_handle;
   short       pict_wd;
   short       pict_ht;

   pict_handle = GetPicture( this->pict_ID );

   pict_rect = ( **( pict_handle ) ).picFrame;

   pict_wd  = pict_rect.right - pict_rect.left;
   pict_ht  = pict_rect.bottom - pict_rect.top;

   SetRect( &pict_rect, 5, 10, 5 + pict_wd, 10 + pict_ht );

   DrawPicture( pict_handle, &pict_rect );
}
```

Draw() uses the object's `pict_ID` in a call to the Toolbox function GetPicture(). This routine loads the PICT resource with an ID of `pict_ID` into memory and returns a handle to the picture. Draw() then dereferences this handle twice to access the picFrame field of the picture. This field is a Rect that holds the boundaries of the picture. From this rectangle, the routine determines the width and the height of the picture. Then a call to SetRect() establishes a rectangle the size of the picture and sets it at the desired location. This call to SetRect() sets the display rectangle to the upper-left corner of the window to which the picture will be drawn. A call to the Toolbox function DrawPicture() does the actual drawing of the picture.

The PetWindow Member Functions

The PetWindow class is so similar to the Animal class that I'll only cover its member functions in passing. One difference worth noting, however, is the data type of the cost data member. In the past it was a long; here it's a Str255 type. Here's the PetWindow class definition:

```
class  PetWindow : public  WindowClass
{
   protected:
      Str255  type;
      Str255  cost;

   public:
      void  Set_Type( Str255 );
      void  Get_Type( Str255 );
      void  Set_Cost( Str255 );
      void  Get_Cost( Str255 );
      void  Draw( void );
};
```

The Set_Type() and Set_Cost() functions set the values of the two data members, while the Get_Type() and Get_Cost() functions retrieve those values from the data members. Here are the definitions for those functions:

```
void  PetWindow :: Set_Type( Str255 name )
{
   Fill_Str255( this->type, name );
}

  void  PetWindow :: Get_Type( Str255 name )
{
   Fill_Str255( name, this->type );
}


void  PetWindow :: Set_Cost( Str255 amount )
{
   Fill_Str255( this->cost, amount );
}
```

```
void  PetWindow :: Get_Cost( Str255 amount )
{
   Fill_Str255( amount, this->cost );
}
```

The Draw() function is inherited from the WindowClass base class and then overridden. The PetWindow version of this function simply writes out the string values of the two PetWindow data members:

```
void  PetWindow :: Draw( void )
{
   Str255  str;

   MoveTo( 20, 30 );
   DrawString( this->type );
   MoveTo( 20, 45 );
   DrawString( "\p$" );
   DrawString( this->cost );
}
```

Updating Object Windows

The ObjectWindows example you saw in Chapter 10 had an Update() member function that did the drawing—and updating—for the window of each object. It simply had a couple of calls to graphics routines "hard coded" into it. A call to DrawString() drew a line of text, and a call to FrameRect() drew a rectangle. Because of this, the window for every object the program created had the same things drawn to it—definitely not very Mac-like.

DerivedWindows overcomes the serious flaw of ObjectWindows by providing each type of object—PictWindow and PetWindow—with its own drawing routine. Figure 11–17 illustrates this. In the figure, the window belongs to an object of the PictWindow class. Part of the window that was covered has become exposed, so it needs updating. The Handle_One_Event() function determines which object the window belongs to and then sends that object an Update() message. The Update() routine was inherited from the WindowClass—the class that the PictWindow class was derived from. Update() relies on an object's Draw() routine to do the actual drawing. Because this object is a PictWindow object, it's the PictWindow version of Draw() that is executed in this example. This ensures that a picture—and not the text that the PetWindow class Draw() routine uses—is drawn in the window.

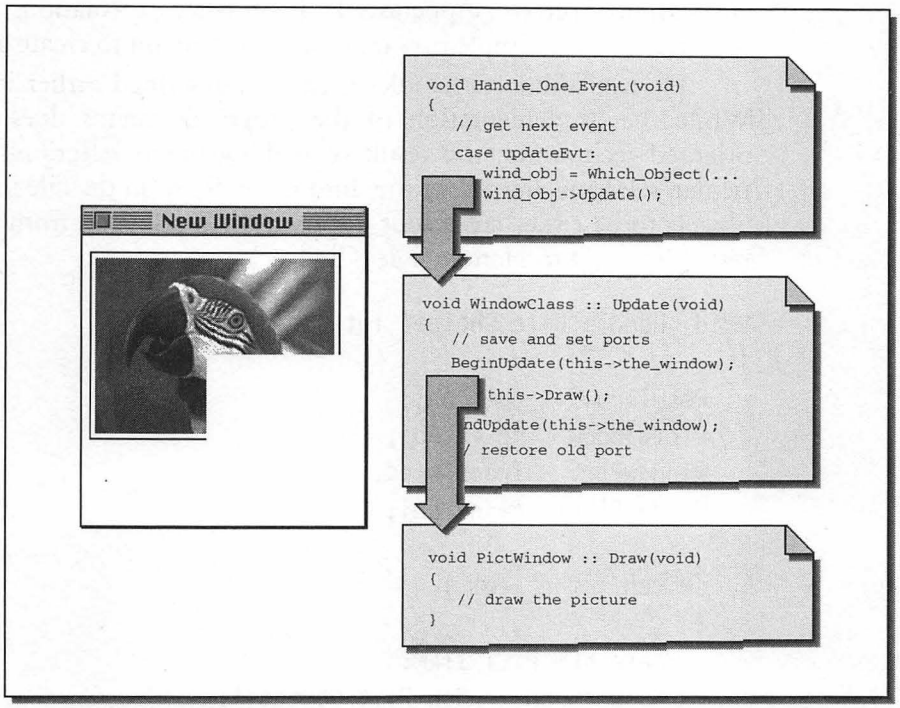


FIGURE 11–17 Each class has its own drawing member function to update an object's window

DerivedWindows goes beyond just providing a separate drawing routine for each class. The Draw() routines don't simply draw predefined graphics. Instead, they base what is to be drawn on the data members of each object. Because every object contains its own copy of each data member, every object can draw something unique to its window. In Figure 11–17, the PictWindow Draw() function will use the `pict_ID` data member value of this particular object to draw the correct picture to the window.

Menus and Objects

None of the example programs in the first 10 chapters includes menus. For the sake of brevity, menu resources and menu code were omitted. That kept examples short and to the point. Of course, now that it's time for a complete example, I'll want to add menus. There isn't anything object-oriented about the way I'm handling menus, so if you're already familiar with menu resources and the display and handling of menus, there won't be any surprises here. If you need some brushing up on how menus are implemented

and handled, refer to Appendix B. It provides explanations, as well as the source code for a simple program that uses a menu to create an object.

DerivedWindows handles menus as described earlier in this chapter. While the implementation of the program's menus doesn't use object-oriented techniques, the result of making menu selections does. In particular, selecting any one of the three menu items in the File menu will cause objects to be either created or disposed of. A selection from the File menu results in a call to Handle_File_Choice(), shown here:

```
void  Handle_File_Choice( int the_item )
{
   PetWindow    *pet_obj;
   PictWindow   *pict_obj;
   WindowPtr    front_wind;
   WindowClass  *wind_obj;

   switch ( the_item )
   {
      case NEW_PICT_ITEM:
         pict_obj = New_Pict_Object();
         break;

      case NEW_PET_ITEM:
         pet_obj = New_Pet_Object();
         break;

      case QUIT_ITEM:
         while ( ( front_wind = FrontWindow() ) != nil )
         {
            wind_obj = Which_Object( front_wind );
            delete  wind_obj;
         }
         ExitToShell();
         break;
   }
}
```

The New Picture menu item results in a call to New_Pict_Object(). This function will display a dialog box that lets the user provide some input

concerning the PictWindow object that is to be created. When the dialog box is dismissed, New_Pict_Object() returns a PictWindow object.

The New Pet menu item works just like the New Picture item. The New_Pet_Object() function opens a dialog box and returns a PetWindow object.

Selecting the Quit menu item ends the program by calling ExitToShell(). Before that, however, a while loop ensures that each object—no matter how few or how many there are—will be deleted. The while loop calls the Toolbox routine FrontWindow() to find out which window is in the foreground. If there is a window open, the body of the loop determines to which object the window belongs. Then that object is deleted. Deleting the object causes its destructor to execute, which disposes of the window and frees the memory it occupied—and that's good memory management. Deleting each object is a necessary step if you want to ensure that your program calls the destructor function of each object. Remember, it's the delete operator that triggers the calling of an object's destructor.

Objects and User Input

DerivedWindows creates an object in much the same way that the example programs in Chapter 9 did—by posting a dialog box that lets the user input information about the object and then returning a new object to the program.

If the user selects the New Picture menu item from the File menu, function New_Pict_Object() is called to open the dialog box pictured in Figure 11–18. The user can use the radio buttons to determine which one of two pictures will be associated with the object.

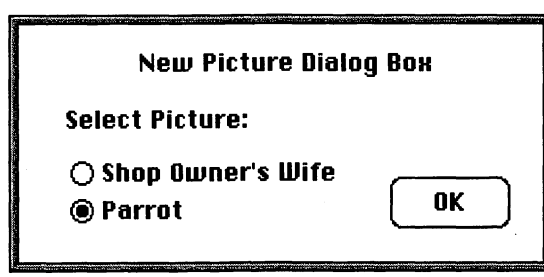


FIGURE 11–18 The New Picture dialog box in the DerivedWindows program

New_Pict_Object() is similar to the New_Rectangle() function developed in Chapter 9. The function begins by opening the dialog box and turning on one of the two radio buttons.

As the user clicks on radio buttons, the short integer variable pict_ID changes value. This variable holds the ID of the PICT resource associated with each radio button. When the OK button is clicked on, a new PictWindow object is created, and its pict_ID data member is immediately set to the PICT resource ID that corresponds to the last radio button selected. The dialog box is then disposed of, and the new object is returned to the program. Here's a look at the New_Pict_Object() function:

```
PictWindow  *New_Pict_Object( void )
{
   DialogPtr   dlog;
   Boolean     dlog_done = false;
   short       item;
   short       new_radio;
   short       old_radio;
   short       pict_ID;
   PictWindow  *temp;

   dlog = GetNewDialog( PICT_DLOG_ID, nil,
                        (WindowPtr)-1L );

   old_radio = BIRD_BUTTON;
   new_radio = BIRD_BUTTON;
   Set_Radio_Buttons( dlog, &old_radio, new_radio );
   pict_ID = BIRD_PICT_ID;

   while ( dlog_done == false )
   {
      ModalDialog( nil, &item );

      switch ( item )
      {
         case BIRD_BUTTON:
            new_radio = item;
            Set_Radio_Buttons( dlog, &old_radio,
                               new_radio );
            pict_ID = BIRD_PICT_ID;
            break;
```

```
            case WIFE_BUTTON:
               new_radio = item;
               Set_Radio_Buttons( dlog, &old_radio,
                                  new_radio );
               pict_ID = WIFE_PICT_ID;
               break;

            case OK_BUTTON:
               temp = new  PictWindow;
               temp->Set_Pict_ID( pict_ID );
               dlog_done = true;
               break;
         }
      }
      DisposDialog( dlog );

      return ( temp );
}
```

The handling of a New Pet menu item selection is similar to the handling of a New Picture choice. A function named New_Pet_Object() is called to open the dialog box pictured in Figure 11–19. The user enters text in the edit boxes to provide information about the animal that is to be added to the program.

The New_Pet_Object() function opens the dialog box shown in Figure 11–19. The function then waits for the user to click on the OK

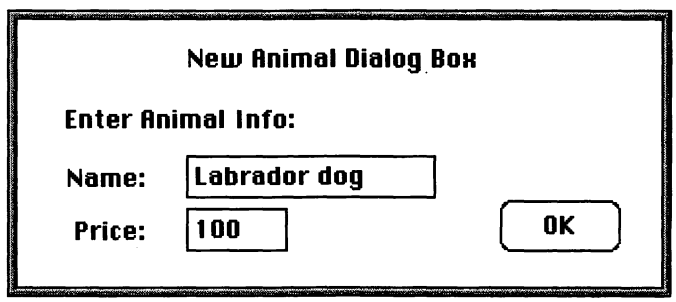


FIGURE 11–19 The New Pet dialog box in the DerivedWindows program

button. When that happens, the strings from the two edit boxes are obtained, a new PetWindow object is created, and the edit box strings are used in the setting of the object's two data members. The dialog box is then disposed of, and the new object is returned.

```
PetWindow  *New_Pet_Object( void )
{
   DialogPtr  dlog;
   Boolean    dlog_done = false;
   short      item;
   Str255     name_str;
   Str255     price_str;
   PetWindow  *temp;

   dlog = GetNewDialog( PET_DLOG_ID, nil, (WindowPtr)-1L );

   while ( dlog_done == false )
   {
      ModalDialog( nil, &item );

      switch ( item )
      {
         case OK_BUTTON:
            Get_Text_From_Edit( dlog, NAME_ITEM,
                                name_str );
            Get_Text_From_Edit( dlog, PRICE_ITEM,
                                price_str );

            temp = new  PetWindow;
            temp->Set_Type( name_str );
            temp->Set_Cost( price_str );
            dlog_done = true;
            break;
      }
   }
   DisposDialog( dlog );

   return ( temp );
}
```

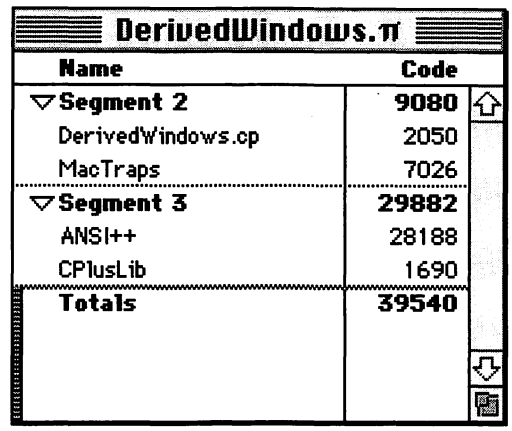


FIGURE 11–20 The DerivedWindows project file

The DerivedWindows Source Code

Now it's time for a look at the complete source code listing for DerivedWindows. You'll find the source code on the accompanying disk in a single file named DerivedWindows.cp. The project file, also included, is pictured in Figure 11–20.

After compiling the code, I performed a simple test to see if I'd properly defined the PetWindow and PictWindow classes to be derived from WindowClass. I selected Browser from the Source menu in the THINK Project Manager. Sure enough, the Browser found that these two classes were derived from the WindowClass. Figure 11–21 shows what the Browser reported to me.

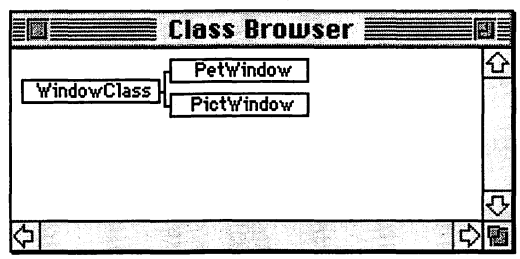


FIGURE 11–21 The DerivedWindows classes, displayed in the THINK Project Manager's Class Browser

```
// ****************  DerivedWindows.cp  ****************

//_____________________________________________________________
//                                          forward references

class    WindowClass;
class    PictWindow;
class    PetWindow;


//_____________________________________________________________
//                                        function prototypes

void         Set_Up_Menu_Bar( void );
void         Initialize_Toolbox( void );
void         Handle_One_Event( void );
void         Handle_Mouse_Down( EventRecord * );
void         Handle_Menu_Choice( long );
void         Handle_Apple_Choice( int );
void         Handle_File_Choice( int );
PictWindow  *New_Pict_Object( void );
PetWindow   *New_Pet_Object( void );
WindowClass *Which_Object( WindowPtr );
void         Set_Radio_Buttons( DialogPtr, short *,
                                short );
void         Get_Text_From_Edit( DialogPtr, short,
                                 Str255 );
void         Fill_Str255( Str255, Str255 );


//_____________________________________________________________
//                                          #define directives

#define     MENU_BAR_ID           128
#define     APPLE_MENU_ID         128
#define        SHOW_ABOUT_ITEM      1
#define     FILE_MENU_ID          129
#define        NEW_PICT_ITEM        1
#define        NEW_PET_ITEM         2
```

```
#define        QUIT_ITEM           4
#define     EDIT_MENU_ID         130

#define     ABOUT_ALRT_ID        130
#define     WIND_ID              128
#define     WIFE_PICT_ID         128
#define     BIRD_PICT_ID         129
#define     PICT_DLOG_ID         128
#define        OK_BUTTON           1
#define        WIFE_BUTTON         2
#define        BIRD_BUTTON         3
#define     PET_DLOG_ID          129
#define        NAME_ITEM           2
#define        PRICE_ITEM          3


//_____________________________________________________________
//                                          class definitions

class  WindowClass
{
   protected:
      WindowPtr  the_window;
      Rect       drag_rect;

   public:
      WindowClass( void );
     ~WindowClass( void );

      virtual void  Select( void );
      virtual void  Drag( Point );
      virtual void  Close( void );
      virtual void  TrackClose( Point );
      virtual void  Update( void );
      virtual void  Draw( void );
};
```

```
class  PetWindow : public  WindowClass
{
   protected:
      Str255  type;
      Str255  cost;

   public:
      void  Set_Type( Str255 );
      void  Get_Type( Str255 );
      void  Set_Cost( Str255 );
      void  Get_Cost( Str255 );
      void  Draw( void );
};


class  PictWindow : public  WindowClass
{
   protected:
      short  pict_ID;

   public:
      void  Set_Pict_ID( short );
      void  Draw( void );
};

//_____________________________________________________________
//                  WindowClass member function definitions

WindowClass :: WindowClass( void )
{
   this->the_window = GetNewWindow( WIND_ID, nil,
                                    (WindowPtr)-1L );

   this->drag_rect = screenBits.bounds;

   SetWRefCon( this->the_window, (long)this );

   SetPort( this->the_window );
}
```

```
WindowClass :: ~WindowClass( void )
{
    DisposeWindow( this->the_window );
}


void  WindowClass :: Select( void )
{
    SelectWindow( this->the_window );
}


void  WindowClass :: Drag( Point where )
{
   DragWindow( this->the_window, where, &this->drag_rect );
}


void  WindowClass :: Close( void )
{
   delete this;
}


void  WindowClass :: TrackClose( Point where )
{
   if ( TrackGoAway( this->the_window, where ) )
      this->Close();
}


void  WindowClass :: Update( void )
{
   WindowPtr  save_port;
   Rect       the_rect;

   GetPort( &save_port );
```

```
   SetPort( this->the_window );

   BeginUpdate( this->the_window );
      this->Draw();
   EndUpdate( this->the_window );

   SetPort( save_port );
}


void  WindowClass :: Draw( void )
{
}


//_____________________________________________________________
//                    PictWindow member function definitions

void  PictWindow :: Set_Pict_ID( short  the_ID )
{
   this->pict_ID = the_ID;
}


void  PictWindow :: Draw( void )
{
   Rect        pict_rect;
   PicHandle   pict_handle;
   short       pict_wd;
   short       pict_ht;

   pict_handle = GetPicture( this->pict_ID );

   pict_rect = ( **( pict_handle ) ).picFrame;

   pict_wd  = pict_rect.right - pict_rect.left;
   pict_ht  = pict_rect.bottom - pict_rect.top;

   SetRect( &pict_rect, 5, 10, 5 + pict_wd, 10 + pict_ht );
```

```
   DrawPicture( pict_handle, &pict_rect );
}


//_____________________________________________________
//                  PetWindow member function definitions

void  PetWindow :: Set_Type( Str255 name )
{
   Fill_Str255( this->type, name );
}


void  PetWindow :: Get_Type( Str255 name )
{
   Fill_Str255( name, this->type );
}


void  PetWindow :: Set_Cost( Str255 amount )
{
   Fill_Str255( this->cost, amount );
}


void  PetWindow :: Get_Cost( Str255 amount )
{
   Fill_Str255( amount, this->cost );
}


void  PetWindow :: Draw( void )
{
   Str255  str;

   MoveTo( 20, 30 );
   DrawString( this->type );
   MoveTo( 20, 45 );
```

```
    DrawString( "\p$" );
    DrawString( this->cost );
}


//_____________________________________________________________
//                                                       main()

void  main( void )
{
    Initialize_Toolbox();

    Set_Up_Menu_Bar();

    for ( ; ; )
        Handle_One_Event();
}


//_____________________________________________________________
//                                          initialize the Mac

void  Initialize_Toolbox( void )
{
    InitGraf( &thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    FlushEvents( everyEvent, 0L );
    InitCursor();
}


//_____________________________________________________________
//                                    set up menu bar and menus

void  Set_Up_Menu_Bar( void )
```

```
{
   Handle      menu_bar_handle;
   MenuHandle  apple_menu;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   apple_menu = GetMHandle( APPLE_MENU_ID );
   AddResMenu( apple_menu, 'DRVR' );

   DrawMenuBar();
}


//_____________________________________________________________
//                                        handle single event

void  Handle_One_Event( void )
{
   EventRecord  the_event;
   WindowClass  *wind_obj;

   WaitNextEvent( everyEvent, &the_event, 15L, 0L );

   switch ( the_event.what )
   {
      case mouseDown:
         Handle_Mouse_Down( &the_event );
         break;

      case updateEvt:
         wind_obj = Which_Object( (WindowPtr)
                                  the_event.message );
         wind_obj->Update();
         break;
   }
}
```

```
//_____________________________________________________________
//                                  handle a click of the mouse

void  Handle_Mouse_Down( EventRecord *the_event )
{
   WindowPtr    window;
   short        the_part;
   long         menu_choice;
   WindowClass  *wind_obj;

   the_part = FindWindow ( the_event->where, &window );

   switch ( the_part )
   {
      case inMenuBar:
         menu_choice = MenuSelect( the_event->where );
         Handle_Menu_Choice( menu_choice );
         break;

      case inSysWindow:
         SystemClick ( the_event, window );
         break;

      case inDrag:
         wind_obj = Which_Object( window );
         wind_obj->Drag( the_event->where );
         break;

     case inContent:
         wind_obj = Which_Object( window );
         wind_obj->Select();
         break;

      case inGoAway:
         wind_obj = Which_Object( window );
         wind_obj->TrackClose( the_event->where );
         break;
```

```
    }
}


//_____________________________________________________________
//                                  handle a click on a menu

void  Handle_Menu_Choice( long  menu_choice )
{
    int  the_menu;
    int  the_menu_item;

    if ( menu_choice != 0 )
    {
        the_menu = HiWord( menu_choice );
        the_menu_item = LoWord( menu_choice );

        switch ( the_menu )
        {
            case APPLE_MENU_ID:
                Handle_Apple_Choice( the_menu_item );
                break;

            case FILE_MENU_ID:
                Handle_File_Choice( the_menu_item );
                break;

            case EDIT_MENU_ID:
                break;
        }
        HiliteMenu(0);
    }
}


//_____________________________________________________________
//                          handle a click in the Apple menu

void  Handle_Apple_Choice( int the_item )
```

```
{
   Str255      desk_acc_name;
   int         desk_acc_number;
   MenuHandle  apple_menu;

   switch ( the_item )
   {
      case SHOW_ABOUT_ITEM :
         Alert( ABOUT_ALRT_ID, nil );
         break;

      default :
         apple_menu = GetMHandle( APPLE_MENU_ID );
         GetItem( apple_menu, the_item, desk_acc_name );
         desk_acc_number = OpenDeskAcc( desk_acc_name );
         break;
   }
}


//_____________________________________________________________
//                        handle a click in the File menu

void  Handle_File_Choice( int the_item )
{
   PetWindow    *pet_obj;
   PictWindow   *pict_obj;
   WindowPtr    front_wind;
   WindowClass  *wind_obj;

   switch ( the_item )
   {
      case NEW_PICT_ITEM:
         pict_obj = New_Pict_Object();
         break;

      case NEW_PET_ITEM:
         pet_obj = New_Pet_Object();
         break;
```

```
      case QUIT_ITEM:
         while ( ( front_wind = FrontWindow() ) != nil )
         {
            wind_obj = Which_Object( front_wind );
            delete  wind_obj;
         }
         ExitToShell();
         break;
   }
}


//_____________________________________________________________
//                          open new Picture object dialog

PictWindow  *New_Pict_Object( void )
{
   DialogPtr   dlog;
   Boolean     dlog_done = false;
   short       item;
   short       new_radio;
   short       old_radio;
   short       pict_ID;
   PictWindow  *temp;

   dlog = GetNewDialog( PICT_DLOG_ID, nil,
                        (WindowPtr)-1L );

   old_radio = BIRD_BUTTON;
   new_radio = BIRD_BUTTON;
   Set_Radio_Buttons( dlog, &old_radio, new_radio );
   pict_ID = BIRD_PICT_ID;

   while ( dlog_done == false )
   {
      ModalDialog( nil, &item );

      switch ( item )
```

```
        {
            case BIRD_BUTTON:
                new_radio = item;
                Set_Radio_Buttons( dlog, &old_radio,
                                   new_radio );
                pict_ID = BIRD_PICT_ID;
                break;

            case WIFE_BUTTON:
                new_radio = item;
                Set_Radio_Buttons( dlog, &old_radio,
                                   new_radio );
                pict_ID = WIFE_PICT_ID;
                break;

            case OK_BUTTON:
                temp = new  PictWindow;
                temp->Set_Pict_ID( pict_ID );
                dlog_done = true;
                break;
        }
    }
    DisposDialog( dlog );

    return ( temp );
}


//_____________________________________________________________
//                          open new Picture object dialog

PetWindow  *New_Pet_Object( void )
{
    DialogPtr  dlog;
    Boolean    dlog_done = false;
    short      item;
    Str255     name_str;
    Str255     price_str;
    PetWindow  *temp;
```

```
    dlog = GetNewDialog( PET_DLOG_ID, nil, (WindowPtr)-1L );

    while ( dlog_done == false )
    {
       ModalDialog( nil, &item );

       switch ( item )
       {
          case OK_BUTTON:
             Get_Text_From_Edit( dlog, NAME_ITEM,
                                 name_str );
             Get_Text_From_Edit( dlog, PRICE_ITEM,
                                 price_str );

             temp = new  PetWindow;
             temp->Set_Type( name_str );
             temp->Set_Cost( price_str );
             dlog_done = true;
             break;
       }
    }
    DisposDialog( dlog );

    return ( temp );
}


//_____________________________________________________
//              determine which window object to work with

WindowClass  *Which_Object( WindowPtr wind )
{
    long         wind_ID;
    WindowClass  *temp;

    wind_ID = ((WindowPeek)wind)->refCon;

    temp = (WindowClass *)wind_ID;
```

```
    return ( temp );
}


//_____________________________________________________________
//                                              set radio buttons

void Set_Radio_Buttons( DialogPtr dlog, short *old_radio,
                        short new_radio )
{
    Handle  hand;
    short   type;
    Rect    box;

    GetDItem( dlog, *old_radio, &type, &hand, &box );
    SetCtlValue( (ControlHandle)hand, 0 );

    GetDItem( dlog, new_radio, &type, &hand, &box );
    SetCtlValue( (ControlHandle)hand, 1 );

    *old_radio = new_radio;
}


//_____________________________________________________________
//                                        get edit box contents

void  Get_Text_From_Edit( DialogPtr dlog, short edit_item,
                          Str255 the_str )
{
    Handle  hand;
    short   type;
    Rect    box;

    GetDItem(dlog, edit_item, &type, &hand, &box );
    GetIText(hand, the_str );
}
```

```
//_____________________________________________________
//                          fill the_str with fill_with_str

void Fill_Str255( Str255 the_str, Str255 fill_with_str )
{
   short  str_length;   // length in characters of string
   short  i;            // loop counter

   str_length = *fill_with_str;

   for ( i = str_length; i >= 0; i-- )
      the_str [i] = fill_with_str[i];
}
```

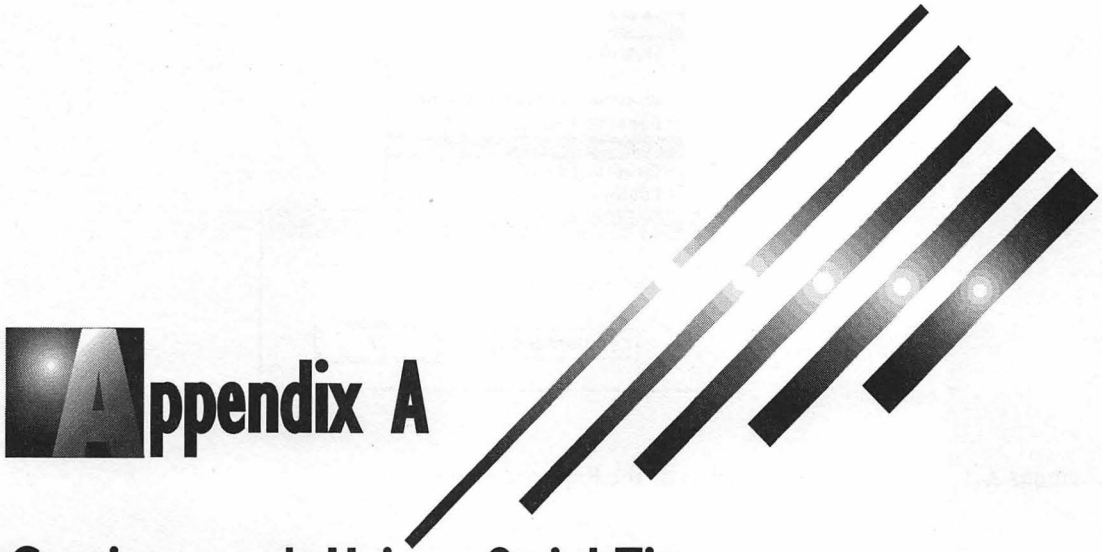
What's Next?

If you're serious about programming, object-oriented programming is a must. In the coming years, you'll see it become the standard in programming. And now, you're ready to be a knowledgeable participant in this wave of the 90s.

So what comes next? You might want to continue practicing your object-oriented skills by modifying the DerivedWindows program. You can easily make the PetWindow class more powerful by just adding more data members and Get and Set member functions to access the new members. Modify the input dialog to let the user enter this new information. Then, with a few additions to the New_Pet_Object() function, you can have PetWindow objects take on this new data.

If DerivedWindows is to become a true database, it must be able to save the information that is entered. For that, you'll want to learn how to write data to files. Another welcome addition would be the ability to print out the information in the data base.

Examining and modifying existing source code is one of the best ways to learn about programming. The disk that is included with this book has more than two dozen programs—so go to it!



Appendix A

Getting and Using QuickTime

QuickTime is an Apple system software extension that adds movie-playing capabilities to your Macintosh. An extension is a piece of software that is loaded into memory when your computer is turned on and remains there until the computer is turned off. By itself, QuickTime doesn't do much. Instead, other programs make use of QuickTime to play movies. One such program is the Simulator C++ software included with this book.

If you have QuickTime, you're all set to use the Simulator C++ program. Just make sure QuickTime is in the Extensions folder in your System Folder. If you don't have QuickTime, get it. And not just for the sake of using the Simulator C++ software. Many new programs require QuickTime to be present on the user's Mac.

Getting QuickTime

Apple wants Mac owners to have QuickTime, so they distribute it freely. The most common place to find it is on one of the online services. If you or someone you know is a member of one of these services, you can download

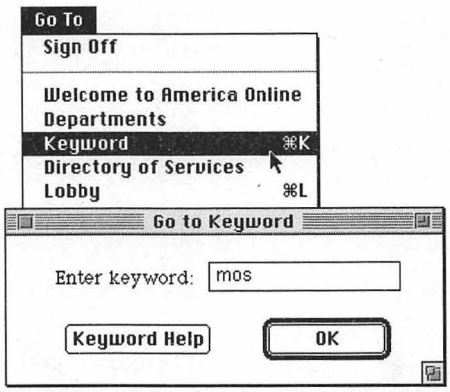


FIGURE A–1 Entering a keyword using the Keyword menu item

the QuickTime file. The remainder of this section describes how to find the QuickTime file on the three most popular online services—America Online, CompuServe, and GEnie.

Downloading from America Online

To download QuickTime from America Online, first log on. Select Keyword from the Go To menu. Type **mos** for Macintosh Operating System in the dialog box that opens, as shown in Figure A–1.

To move into the software libraries section of the Macintosh Operating Systems forum, click on the Software Libraries icon shown in Figure A–2.

Next, search for the QuickTime file. To do this, first click on the Software Search icon shown in Figure A–3.

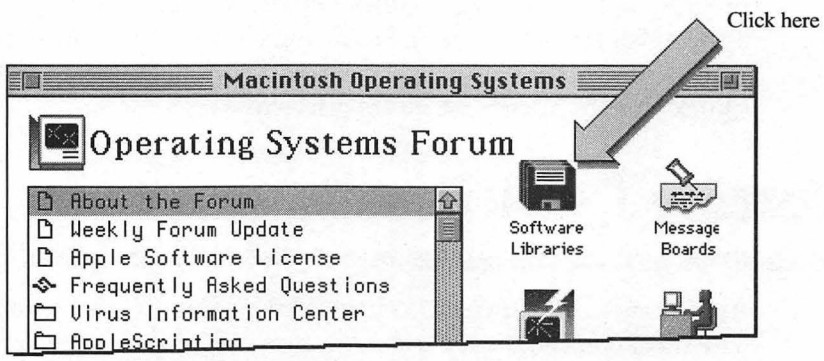


FIGURE A–2 The Software Libraries icon in the MOS forum

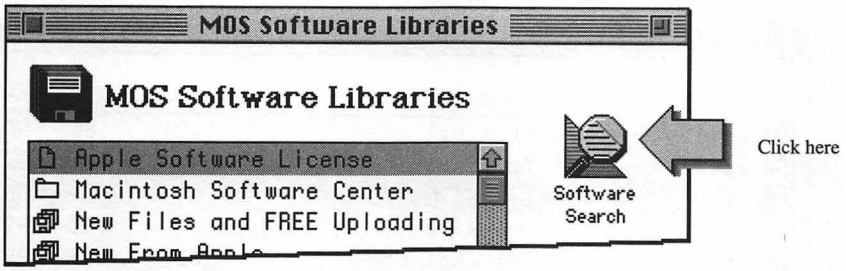


FIGURE A–3 The Software Search icon in the MOS forum

Type **quicktime** in the long edit box at the bottom of the search dialog box. Then click on the List Matching Files button, as shown in Figure A–4.

America Online will display a new dialog box that lists all the files that have something to do with QuickTime. Scroll through the list until you find the QuickTime file itself and then double-click on the filename. Figure A–5 shows the QuickTime file in the scrollable list.

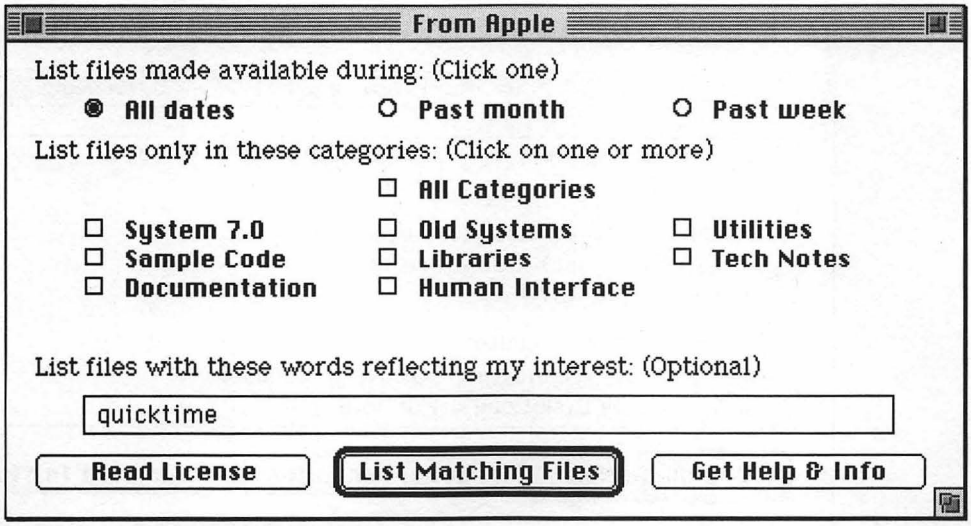


FIGURE A–4 Searching for QuickTime-related files

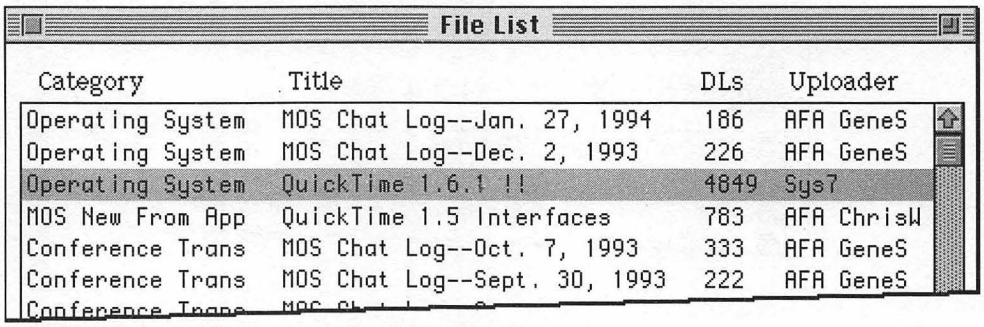


FIGURE A–5 QuickTime, followed by a version number, will appear in the list of files.

After double-clicking on the QuickTime filename, you will see the dialog box shown in Figure A–6, which describes the file and allows you to download it. Click on the Download Now button to begin the download process.

Downloading from CompuServe

To download QuickTime from CompuServe, first log on. Type **go macdev** to go to the Macintosh Developers Forum, which is shown in Figure A–7.

Next, type 3 to enter the libraries section. Figure A–8 shows the Mac Developers Forum menu.

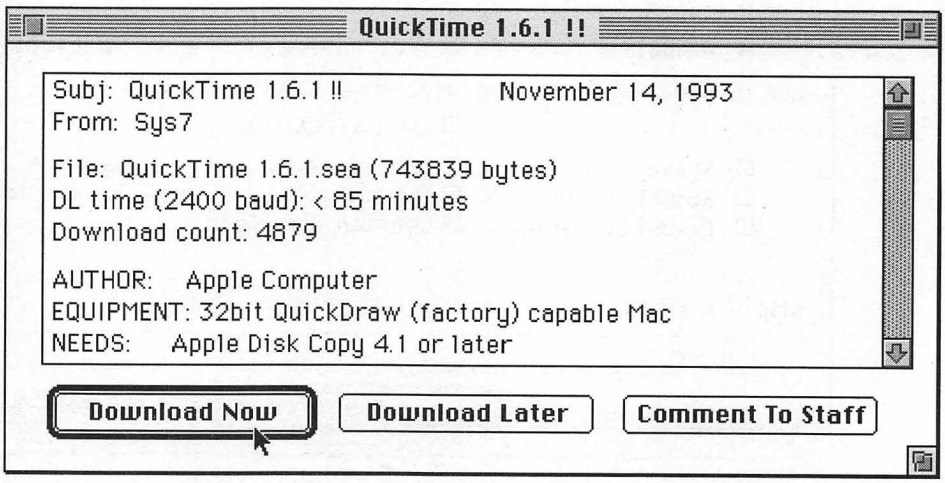


FIGURE A–6 Starting the download of the QuickTime file

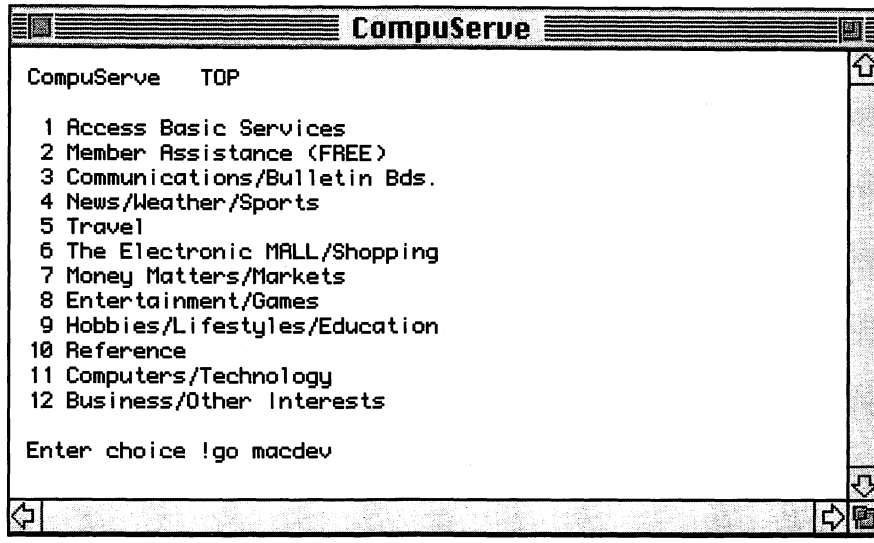


FIGURE A–7 Moving to the Macintosh Developers Forum

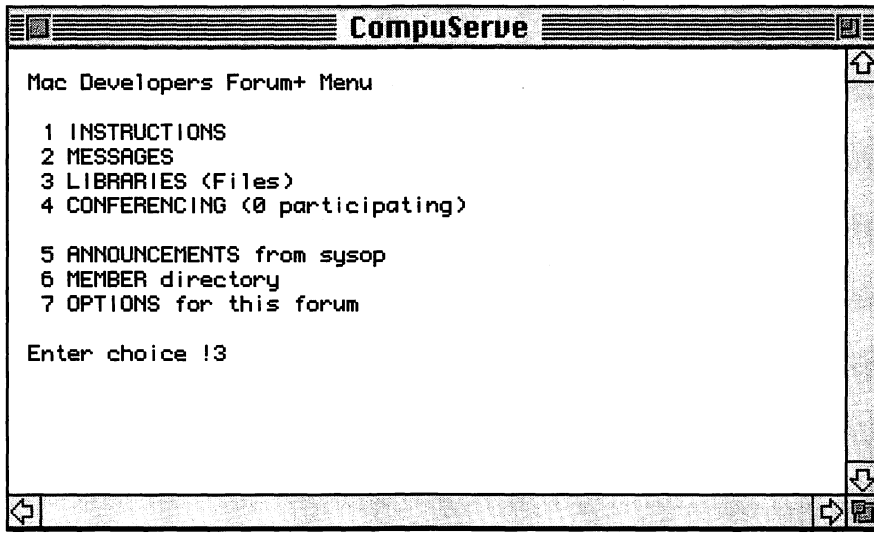


FIGURE A–8 Moving into the libraries section of the Mac Developers Forum

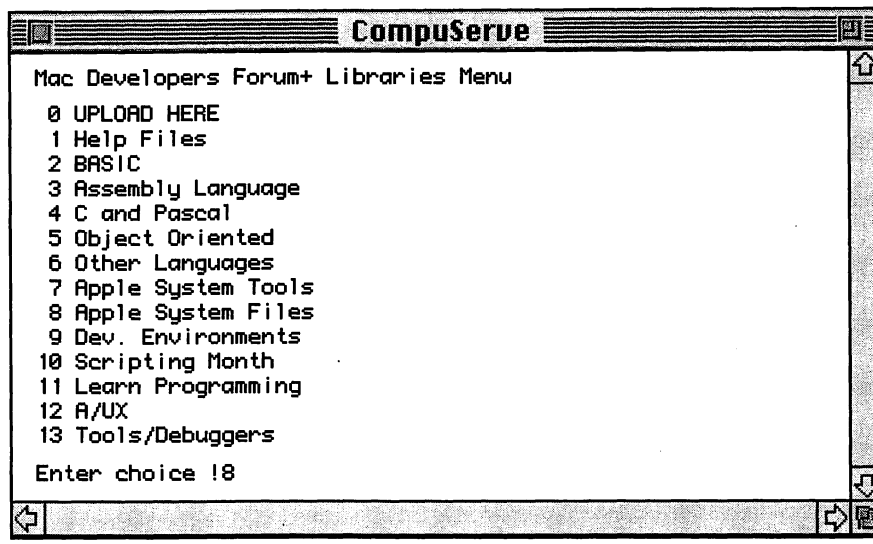


FIGURE A–9 Moving into the Apple System Files library of files

The Mac Developers Forum contains several libraries of files. Type **8** to enter the Apple System Files library, as shown in Figure A–9.

Next, select menu option 1 to browse through the Apple system files. At the first prompt, press Enter. That means the search will take place in the current library. At the next prompt, type **quicktime**. At the final prompt, press Enter so that all files, regardless of age, are included in the search. Figure A–10 shows the responses you should enter.

A description of each file that has something to do with QuickTime will be shown on the screen—one at a time. After each description, press Enter until you reach the QuickTime extension. The title should include the word *QuickTime* along with a version number, as shown in Figure A–11. At that description, type **choices** to see a list of options.

At the list of choices, type **2** to download the file, as shown in Figure A–12.

CompuServe will next provide a list of download options. XMODEM is a very popular transfer protocol, and most communications software has an XMODEM option. Type **1** to select this option. You'll see a message that signals you to begin the download, as shown in Figure A–13.

Initiate the download by selecting the download menu item from the communication software you're using. CompuServe will be sending you the file, and you will be receiving it. Select the appropriate Receive menu item.

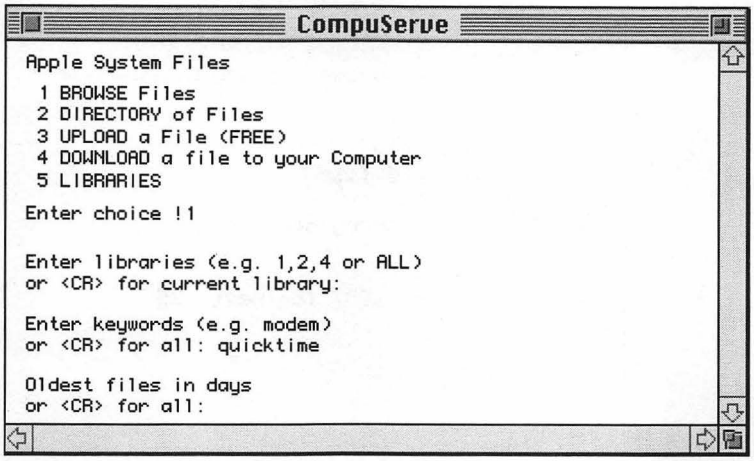


FIGURE A–10 Starting the search for QuickTime-related files

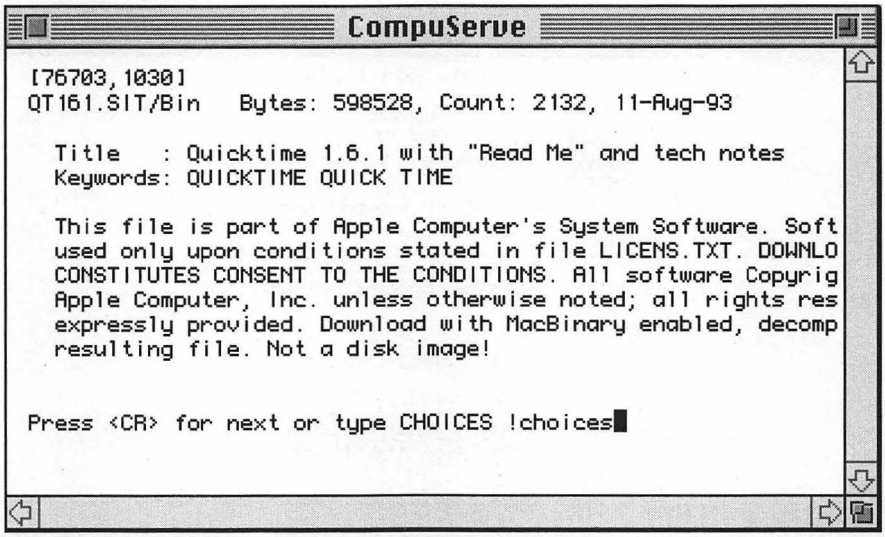


FIGURE A–11 The Quicktime extension file description

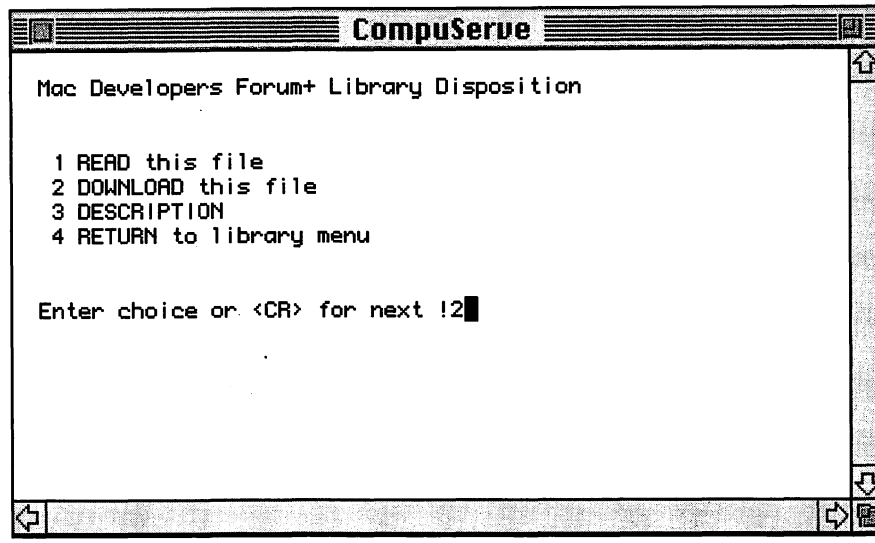


FIGURE A–12 Getting ready to download the QuickTime extension

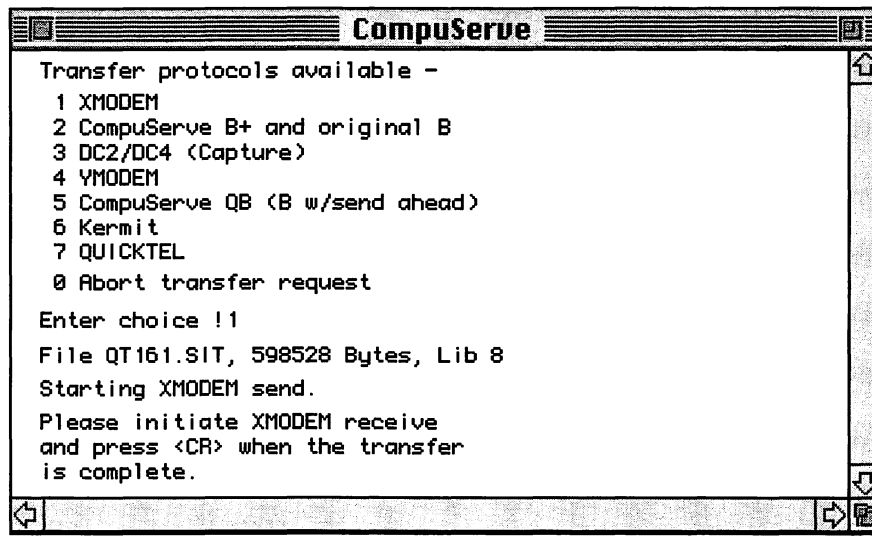


FIGURE A–13 Selecting the XMODEM transfer protocol

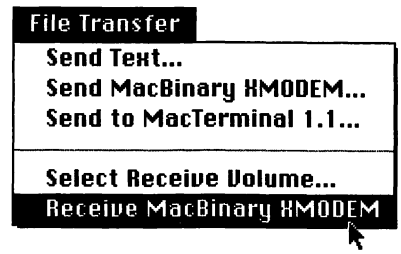


FIGURE A–14 Selecting the XMODEM download option in Microphone

As an example, Figure A–14 shows the proper menu item to choose if you're using Microphone by Software Ventura Corp.

Downloading from GEnie

If you're a member of GEnie, you to can download QuickTime. Begin by logging on. At any prompt type **m 605** to move to the Macintosh Round-Table. Figure A–15 shows how this is done.

Next, enter the Macintosh Software Libraries section by typing **3**, as shown in Figure A–16.

To find the QuickTime extension file, you must perform a file search. Type **3** to begin the search process, as shown in Figure A–17.

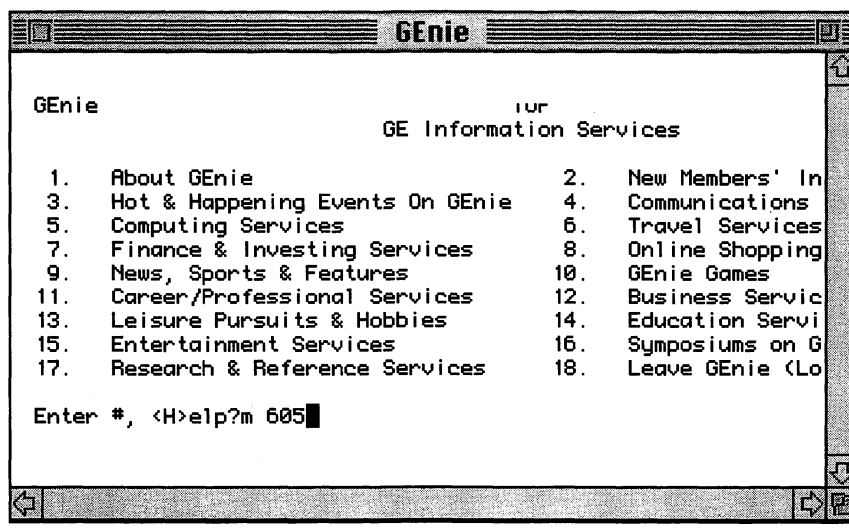


FIGURE A–15 Moving into the Macintosh RoundTable

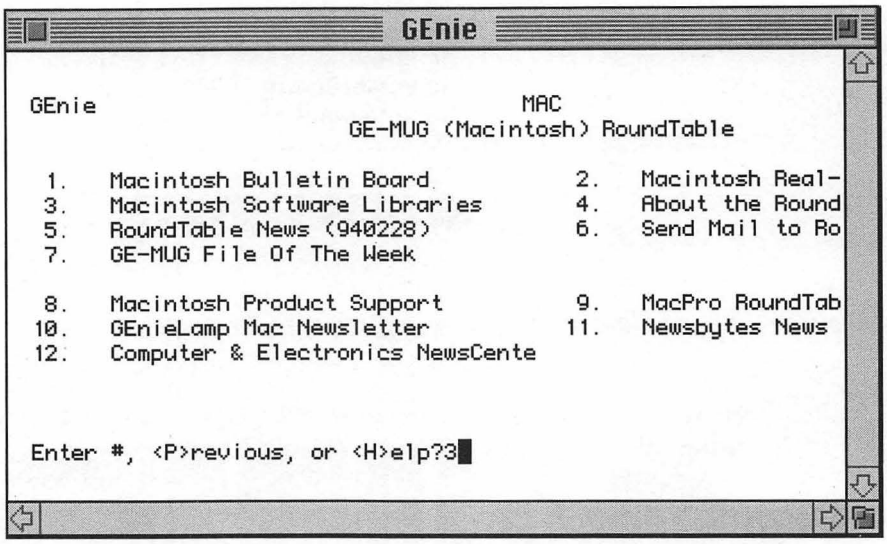


FIGURE A–16 Moving into the Macintosh Software Libraries section

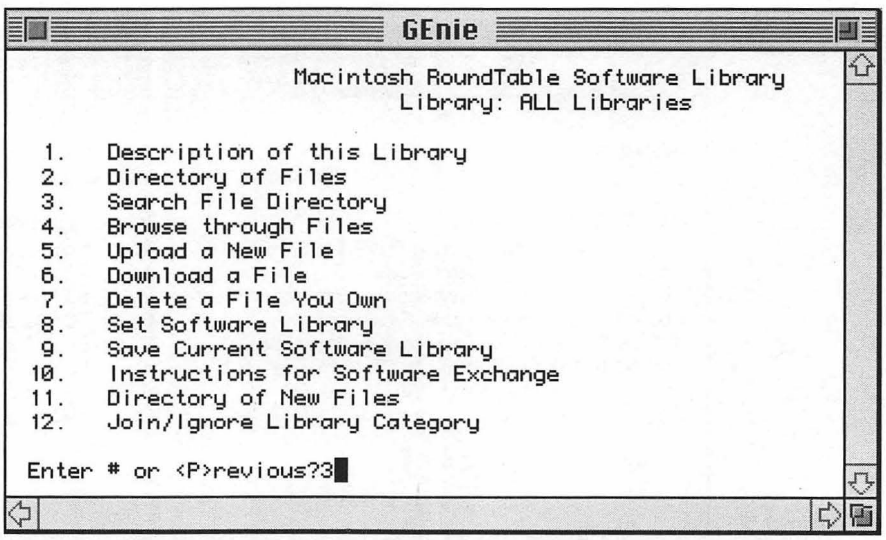


FIGURE A–17 Starting the file search process

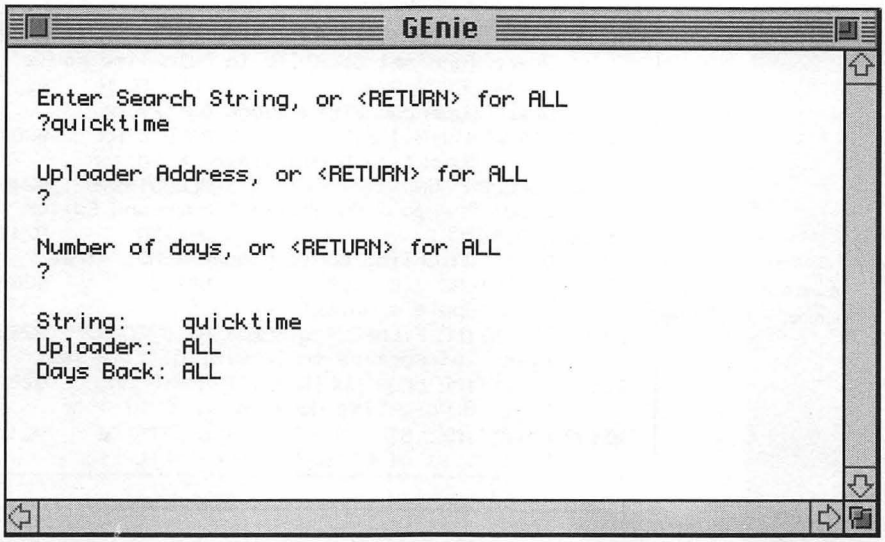


FIGURE A–18 Narrowing the search to all QuickTime-related files

Before beginning the search, GEnie will prompt you for three separate responses. At the first question, type **quicktime**. That tells GEnie to search for all QuickTime-related files. At the second prompt, press Enter. That means that all QuickTime files will be sought, regardless of who originally uploaded them. Finally, press Enter again to tell GEnie that you aren't concerned with how long ago the files were uploaded.Figure A–18 shows what your responses should be.

When the search is complete, a list of files will be displayed. Find the name of the QuickTime file itself. It will be the word *QuickTime* along with a version number. When you find the file, make note of the number that appears to the left of the filename. You'll need this number when you begin the download. Figure A–19 shows the QuickTime extension in a list of QuickTime-related files.

Next, type **n** when asked if you want another search to be performed, as shown in Figure A–20.

Now it's time to download the file. Type **6** to start the download process. You'll be prompted to enter the number of the file to download. Enter the number of the QuickTime file, as shown in Figure A–21.

After you type the file number, a description of the QuickTime extension file will appear. Type **d** to tell GEnie you want to download the file, as shown in Figure A–22.

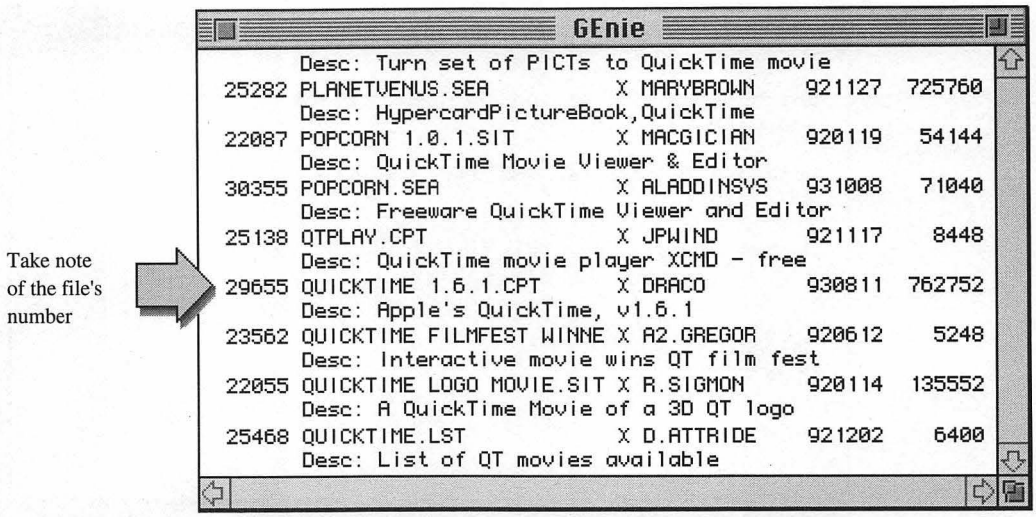


FIGURE A–19 Taking note of the file number of the QuickTime file

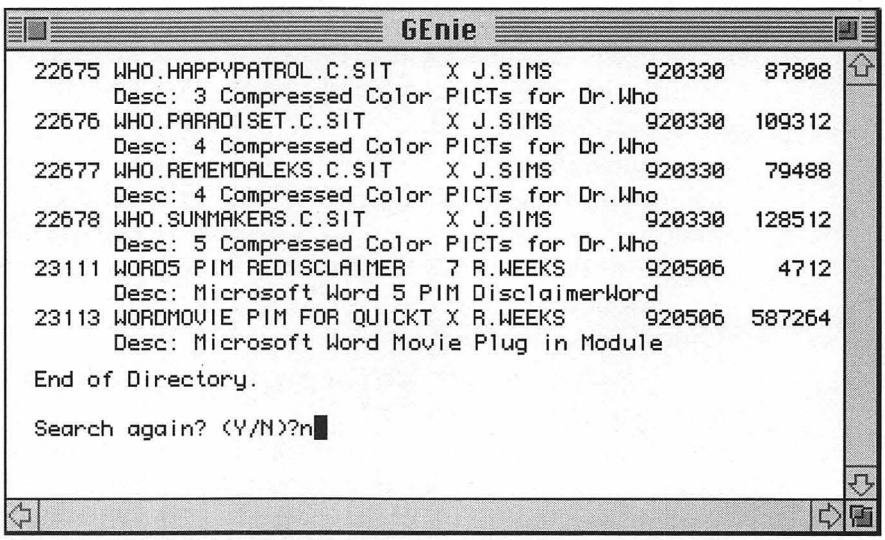


FIGURE A–20 Ending the search process

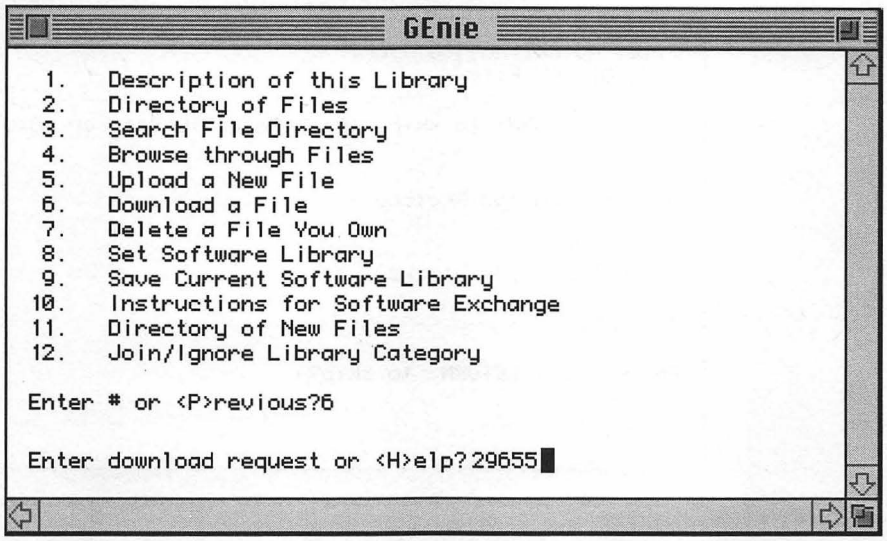


FIGURE A–21 Telling GEnie which file you want to download

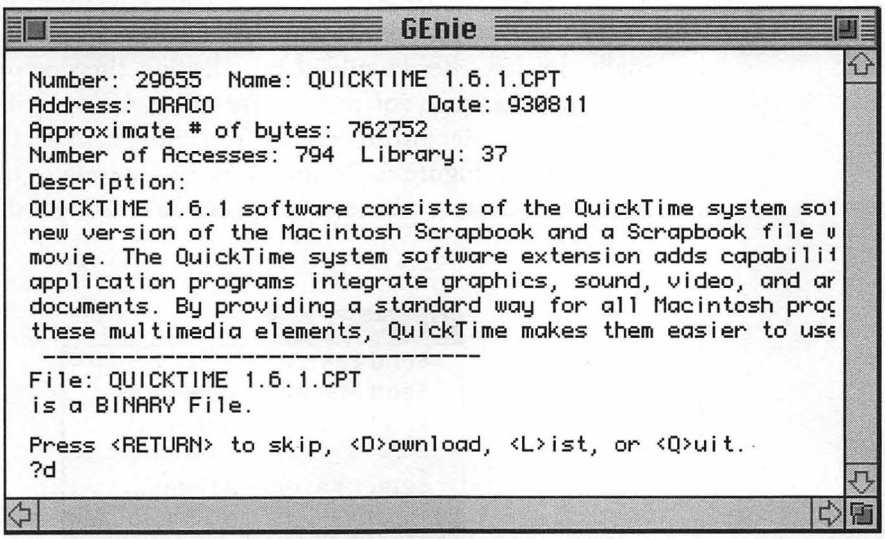


FIGURE A–22 Telling GEnie you want to download the file

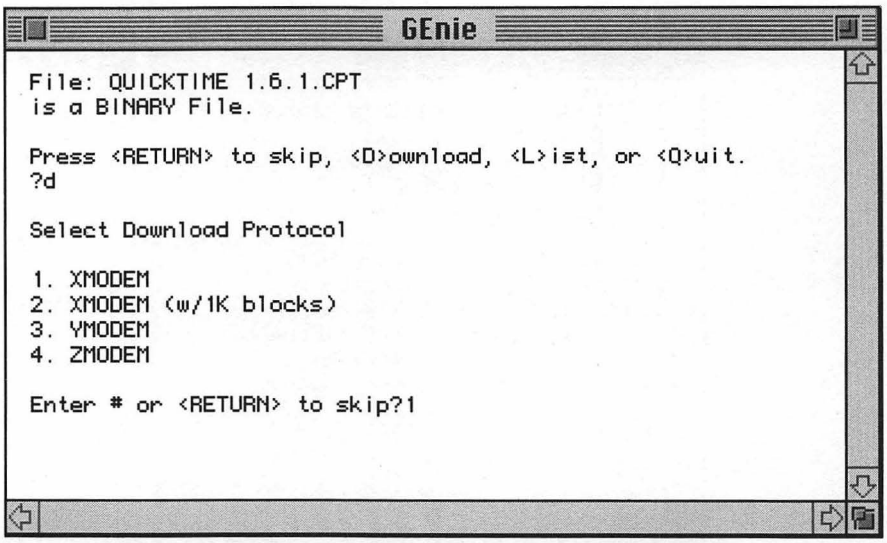


FIGURE A–23 Specifying the transfer protocol to be used in the download

When GEnie asks for the type of download protocol you'll be using, type 1, as shown in Figure A–23. XMODEM is a popular transfer protocol for downloading Macintosh files, and most communication software packages include it.

Finally, initiate the download by selecting the download menu item from the communication software you're using. GEnie will be sending you the file, and your Macintosh will be receiving it. Select the appropriate Receive menu item. Figure A–24 shows the menu item to use if you're using Microphone by Software Ventura Corp. If you're using a different commu-

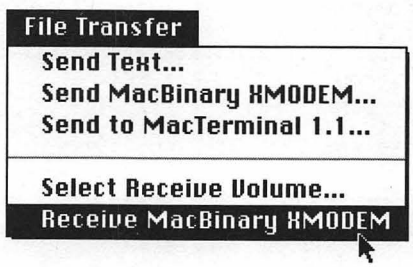


FIGURE A–24 Selecting the XMODEM download option in Microphone

nication program, you'll have a similar Receive XMODEM item in one of the program's menus.

Installing and Using QuickTime

Once you have QuickTime, you have to put it in the System Folder of your Mac. To do this, drag the QuickTime icon to your closed System Folder. It's important that the System Folder not be open at this time. If you're using a version of System 7, an alert will appear to tell you that the file is going to be placed in the Extensions folder. That's exactly what you want, so click on the OK button.

QuickTime is loaded into the memory of your Macintosh each time you start your computer. That's the only time it gets placed in memory. So even though QuickTime is now in the proper folder, it hasn't been loaded into memory. To do this, simply restart your Mac.

Now you're all ready to use the Simulator C++ program and all other software that displays QuickTime movies.

Appendix B

Menu Handling

The Chapter 11 program, DerivedWindows, uses menus to allow the user to create new objects. If you need some brushing up on creating and working with menus, you're in the right spot. This appendix covers the basics of menu resources and the source code you must use to bring these resources to the screen. The source code for MenuDemo—a simple menu-handling program—is also included here.

Menu Resources

Menus start out as resources. Each menu that appears in a program's menu bar has its own MENU resource. Figure B–1 shows how ResEdit displays the three MENU resources for the MenuDemo program that is described at the end of this section.

A program won't know which of the MENU resources to include in its menu bar unless you tie them all together using an MBAR resource. Figure B–2 shows the MBAR resource for the MenuDemo program.

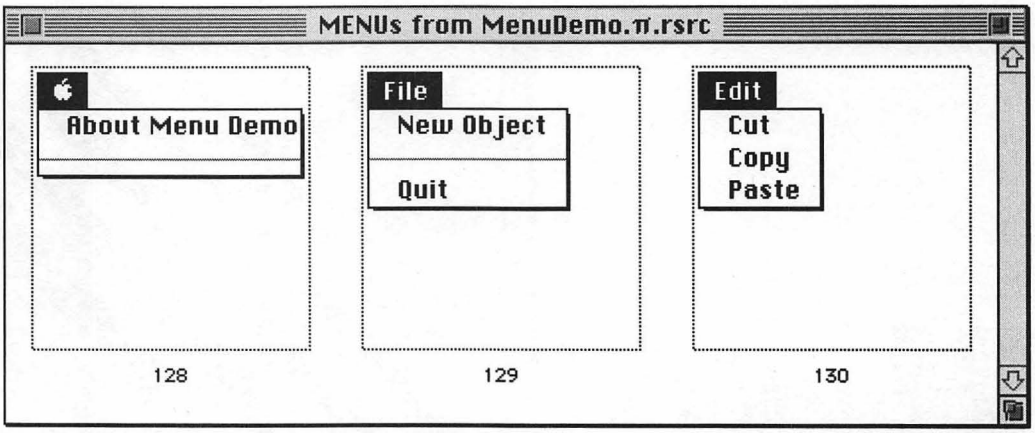


FIGURE B–1 The MENU resources for the MenuDemo program

While you're in your resource editor, it's a good idea to note the IDs of the resources you're creating. The following are the #define directives I'll be

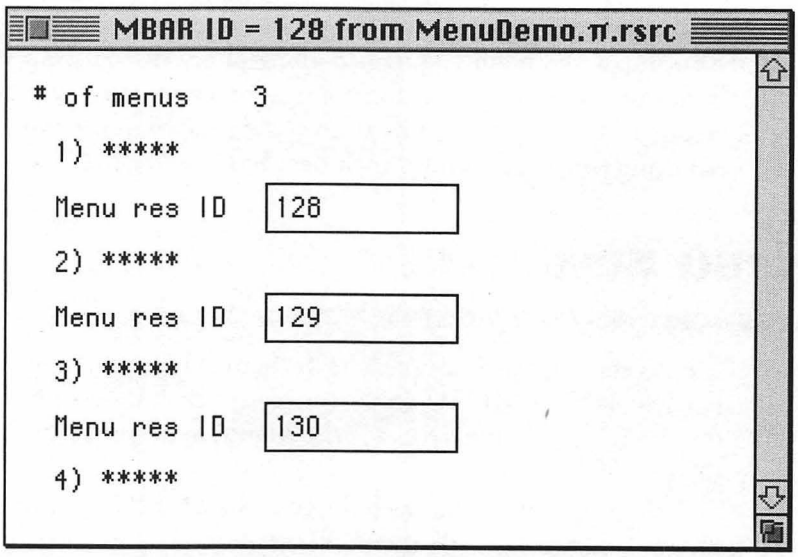


FIGURE B–2 The MBAR resource for the MenuDemo program

using in the MenuDemo program:

```
#define     ABOUT_ALRT_ID       128
#define     MENU_BAR_ID         128
#define     APPLE_MENU_ID       128
#define       SHOW_ABOUT_ITEM     1
#define     FILE_MENU_ID        129
#define       NEW_OBJECT_ITEM     1
// item 2 is a dashed line        2
#define       QUIT_ITEM           3
#define     EDIT_MENU_ID        130
```

Menu Code

Displaying a menu bar in a program is easy; the Menu Manager does all of the work. Here's a routine that adds a menu bar:

```
void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;
   MenuHandle  apple_menu;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );

   SetMenuBar( menu_bar_handle );
   DisposHandle( menu_bar_handle );

   apple_menu = GetMHandle( APPLE_MENU_ID );
   AddResMenu( apple_menu, 'DRVR' );

   DrawMenuBar();
}
```

The Toolbox function GetNewMBar() uses the specified MBAR resource to create a menu list containing a handle to each individual menu that will appear in the menu bar. SetMenuBar() installs the menus in the menu bar. Once the menus are installed, your program no longer needs the handle to the menu bar, so dispose of it with a call to DisposHandle().

You can get a handle to any single menu by making a call to GetMHandle(). You'll want to do that for the Apple menu. After your program receives the MenuHandle, use it in a call to AddResMenu(). This Toolbox function adds the contents of the Apple Menu Items folder—a folder found in the System Folder of all Macs running System 7. If the program will run on a Mac using a version of System 6, any system resources of type DRVR will be added instead. Resources of this type are generally desk accessories.

With all the behind-the-scenes work done, it's time to display the menu bar. A call to DrawMenuBar() takes care of this task.

Setting up the menu bar places it at the top of the screen, but there's more work to be done. If the user clicks on the menu bar, you'll want the Handle_Mouse_Down() routine to take notice. I've added an inMenuBar case label to Hande_Mouse_Down() to handle a click in the menu bar:

```
void  Handle_Mouse_Down( EventRecord *the_event )
{
   WindowPtr  window;
   short      the_part;
   long       menu_choice;

   the_part = FindWindow( the_event->where, &window );

   switch ( the_part )
   {
      case inMenuBar:
         menu_choice = MenuSelect( the_event->where );
         Handle_Menu_Choice( menu_choice );
         break;
   }
}
```

The Toolbox routine MenuSelect() tracks the cursor as the user moves the mouse over the menu bar. MenuSelect() is responsible for the work involved in dropping and hiding menus as the user passes over them. When the mouse is released while on a menu item, MenuSelect() returns a long integer value that contains a reference to both the menu and the menu item. Pass this value on to a separate menu-handling routine such as my Handle_Menu_Choice():

```
void  Handle_Menu_Choice( long  menu_choice )
{
```

```
    int  the_menu;
    int  the_menu_item;

    if ( menu_choice != 0 )
    {
       the_menu = HiWord( menu_choice );
       the_menu_item = LoWord( menu_choice );

       switch ( the_menu )
       {
          case APPLE_MENU_ID:
             Handle_Apple_Choice( the_menu_item );
             break;

          case FILE_MENU_ID:
             Handle_File_Choice( the_menu_item );
             break;

          case EDIT_MENU_ID:
             break;
       }
       HiliteMenu(0);
    }
}
```

Handle_Menu_Choice() uses calls to HiWord() and LoWord() to extract the two pieces of information held in the one `menu_choice` variable. It then uses a switch statement to branch off to a routine written to handle selections from one menu.

Handle_Apple_Choice() handles a selection from the Apple menu. This is a standard routine that can be pasted into just about any of your programs:

```
void  Handle_Apple_Choice( int the_item )
{
   Str255      desk_acc_name;
   int         desk_acc_number;
   MenuHandle  apple_menu;

   switch ( the_item )
```

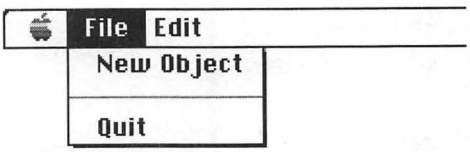


FIGURE B–3 The menu bar displayed by MenuDemo

```
    {
        case SHOW_ABOUT_ITEM :
            Alert( ABOUT_ALRT_ID, nil );
            break;

        default :
            apple_menu = GetMHandle( APPLE_MENU_ID );
            GetItem( apple_menu, the_item, desk_acc_name );
            desk_acc_number = OpenDeskAcc( desk_acc_name );
            break;
    }
}
```

If the user selects the About menu item, Handle_Apple_Choice() posts an alert that contains information about the program. Any other menu selection results in a call to OpenDeskAcc() to open the selected desk accessory—or, in System 7, any other item found in the Apple menu.

Other menu-handling routines will be specific to your program. The MenuDemo program, which is presented in the next section, demonstrates how to handle a File menu that includes a menu item that creates a new object and a Quit menu item.

Menu Example

MenuDemo is a program that uses the menu-handling techniques set forth in this appendix. It uses the resources pictured in the figures of this appendix, as well as an ALRT resource and a WIND resource. When you run MenuDemo, you'll see a menu bar like the one pictured in Figure B–3.

Selecting the About item from the Apple menu displays an alert like the one shown in Figure B–4.



FIGURE B–4 The alert displayed when the About menu item is selected

Choosing New Object from the File menu creates a new Rectangle object. The user will see a window open with a rectangle drawn in it—as pictured in Figure B–5.

It may seem to you that MenuDemo has regressed from the examples in Chapter 9. First, it has only a single class—the Rectangle class. There are no derived classes. Furthermore, a New Object menu selection creates an object but requires no input from the user. A better way of doing things would be to have this menu item open a dialog box that let the user supply all the information about the object that is to be created. Since you're very familiar with the Rectangle derived classes and the use of dialog boxes to accept user input, I thought it would be best to shorten the example code and place the emphasis on the menu code. In Chapter 11, you'll see a complete example that ties menu items to dialog boxes that accept user input and create new objects.

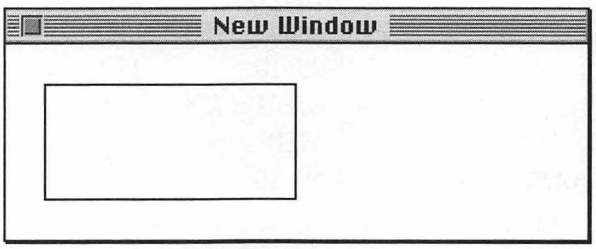


FIGURE B–5 A new object displays a window with a rectangle drawn in it.

```
// ********************  MenuDemo.cp  ********************


//_____________________________________________________________
//                                              forward references

class    Rectangle;


//_____________________________________________________________
//                                             function prototypes

void  Initialize_Toolbox( void );
void  Set_Up_Menu_Bar( void );
void  Handle_One_Event( void );
void  Handle_Mouse_Down( EventRecord * );
void  Handle_Menu_Choice( long );
void  Handle_Apple_Choice( int );
void  Handle_File_Choice( int );


//_____________________________________________________________
//                                              #define directives

#define     MENU_BAR_ID          128
#define     ABOUT_ALRT_ID        128
#define     WIND_ID              128
#define     APPLE_MENU_ID        128
#define       SHOW_ABOUT_ITEM      1
#define     FILE_MENU_ID         129
#define       NEW_OBJECT_ITEM      1
#define       QUIT_ITEM            3
#define     EDIT_MENU_ID         130


//_____________________________________________________________
//                                                global variables

Rectangle  *the_rect;
```

```
//_____________________________________________________________
//                                              class definitions

class  Rectangle
{
   protected:
      Rect  aRect;

   public:
      virtual void  Set_Rectangle( short L, short T,
                                   short R, short B );
      virtual void  Draw_Rectangle( void );
};


//_____________________________________________________________
//                                 member function definitions

void  Rectangle :: Set_Rectangle( short L, short T,
                                  short R, short B )
{
   SetRect( &this->aRect, L, T, R, B );
}


void  Rectangle :: Draw_Rectangle( void )
{
   FrameRect( &this->aRect );
}


//_____________________________________________________________
//                                                      main()

void  main( void )
{
   WindowPtr   the_window;

   Initialize_Toolbox();
```

```
   Set_Up_Menu_Bar();

   the_window = GetNewWindow( WIND_ID, nil,
                              (WindowPtr) -1L );
   SetPort( the_window );

   for ( ; ; )
      Handle_One_Event();
}


//_____________________________________________________________
//                                          initialize the Mac

void  Initialize_Toolbox( void )
{
   InitGraf( &thePort );
   InitFonts();
   InitWindows();
   InitMenus();           // using menus, need to
                          // initialize
   TEInit();
   InitDialogs( 0L );
   FlushEvents( everyEvent, 0L );
   InitCursor();
}


//_____________________________________________________________
//                                    set up menu bar and menus

void  Set_Up_Menu_Bar( void )
{
   Handle      menu_bar_handle;
   MenuHandle  apple_menu;

   menu_bar_handle = GetNewMBar( MENU_BAR_ID );
```

```
    SetMenuBar( menu_bar_handle );
    DisposHandle( menu_bar_handle );

    apple_menu = GetMHandle( APPLE_MENU_ID );
    AddResMenu( apple_menu, 'DRVR' );

    DrawMenuBar();
}


//_____________________________________________________________
//                                          handle single event

void  Handle_One_Event( void )
{
    EventRecord  the_event;
    WindowPtr    window;
    GrafPtr      old_port;

    WaitNextEvent( everyEvent, &the_event, 15L, 0L );

    switch ( the_event.what )
    {
        case mouseDown:
            Handle_Mouse_Down( &the_event );
            break;

        case updateEvt:
            window = (WindowPtr)the_event.message;
            GetPort( &old_port );
            SetPort( window );
            BeginUpdate( window );
                EraseRgn( window->visRgn );
                if ( the_rect != nil )
                    the_rect->Draw_Rectangle();
            EndUpdate( window );
            SetPort( old_port );
            break;
    }
```

```
}
  //_____________________________________________________________
//                                    handle a click of the mouse

void  Handle_Mouse_Down( EventRecord *the_event )
{
   WindowPtr  window;
   short      the_part;
   long       menu_choice;

   the_part = FindWindow( the_event->where, &window );

   switch ( the_part )
   {
      case inMenuBar:
         menu_choice = MenuSelect( the_event->where );
         Handle_Menu_Choice( menu_choice );
         break;
   }
}
  //_____________________________________________________________
//                                      handle a click on a menu

void  Handle_Menu_Choice( long  menu_choice )
{
   int  the_menu;
   int  the_menu_item;

   if ( menu_choice != 0 )
   {
      the_menu = HiWord( menu_choice );
      the_menu_item = LoWord( menu_choice );

      switch ( the_menu )
      {
         case APPLE_MENU_ID:
            Handle_Apple_Choice( the_menu_item );
            break;
```

```
            case FILE_MENU_ID:
               Handle_File_Choice( the_menu_item );
               break;

            case EDIT_MENU_ID:
               break;
         }
         HiliteMenu(0);
      }
}


//_____________________________________________________________
//                        handle a click in the Apple menu

void  Handle_Apple_Choice( int the_item )
{
   Str255      desk_acc_name;
   int         desk_acc_number;
   MenuHandle  apple_menu;

   switch ( the_item )
   {
      case SHOW_ABOUT_ITEM :
         Alert( ABOUT_ALRT_ID, nil );
         break;

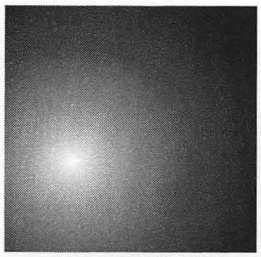
      default :
         apple_menu = GetMHandle( APPLE_MENU_ID );
         GetItem( apple_menu, the_item, desk_acc_name );
         desk_acc_number = OpenDeskAcc( desk_acc_name );
         break;
   }
}

  //_____________________________________________________________
//                        handle a click in the File menu

void  Handle_File_Choice( int the_item )
```

```
{
   switch ( the_item )
   {
      case NEW_OBJECT_ITEM:
         the_rect = new Rectangle;
         the_rect->Set_Rectangle( 20, 20, 150, 80 );
         the_rect->Draw_Rectangle();
         break;

      case QUIT_ITEM:
         if ( the_rect != nil )
            delete the_rect;
         ExitToShell();
         break;
   }
}
```





Index

Symbols

. structure member operator, 68–69
: between class name and access specifier for derived class, 167
:: scope resolution operator, 106, 113, 206–207
-> operator to invoke a member function, 122
~ preceding destructor class name, 149
" " to enclose name of included header file, 55
< > to enclose name of included Apple-supplied header file, 55–56
ƒ character for folder names, 3, 8
π (pi) symbol, creating, 3–4, 6

A

abstract classes, 199–219. *See also* classes
  as common ancestor of family of derived classes, 199, 249
  creating, 201–204
  data members of, 201–202
  difference between ordinary base class and, 200
  member functions of, 202–204
  reasons for creating, 200
AbstractClass sample program
  class hierarchy for, 218
  output of, 211–212
  purpose of, 209–210
  source code of, 212–216
access specifier, 167–168
Add Files dialog box, 9, 11
AddResMenu() Toolbox function, 406
Alert() Toolbox function, 269, 281, 289
AlertInput sample program
  output of, 275
  purpose of, 267, 269–270, 274–276
  resource file for, 267–268
  source code for, 270–273
alerts
  to create new objects, 265–273
  example of, 269–273, 276
  resources for, 267–268
  to select object type, 268–269
  for user to enter information, 266
ALRT resource, 267–268, 280, 408
America Online service, downloading QuickTime from, 388–390

ANSI++ library, 9, 16
application
  converting project's source code into, 26–27
  icon for, 26, 28
  naming, 26–27
arguments, functions with different number of, 79–83

B

BadCAllocation sample program
  memory allocation in, 96–99
  source code for, 96
base class
  abstract. *See* abstract classes
  data members and member functions in, 165
  defined, 164
  example of derived class and, 166
  sample code for, 166–167
base objects, member functions used by, 187–188
BeginUpdate() Toolbox function, 279, 317
branching statements, 60–67
breakpoints during debugging, setting, 23–25, 148–149

C

C compiler, preprocessor in, 54
C language
  allocating memory in, 85–99
  basic differences between C++ language and, 77–79
  as basis of C++, 51–76
  C++ as superset of, 32, 47
  comments using /* and */ in, 33, 79
  OOP and, 45–48
C++
  additions made to C language to produce, 77–108
  allocating memory in, 99–104
  basic differences between C language and, 77–79
  comments using /* or // in, 33, 79
  defining a class in, 43
  as dominant computer language, xv
  example of, 345–385
  introduction to, 31–49
  as superset of C language, 32, 47
C++ code, writing and running, 17–19
C++ compiler
  overview of using, 1–30
  preprocessor in, 54
CarStruct sample program for struct, 70–72
CatalogStruct sample program, 73–75
char (character) data type, 51, 53–54
Class Browser window, 217–219, 369
class data type, 71–75
  struct to represent data members of, 72
class hierarchy, 216–219
class keyword in a forward reference, 230
classes, 42–45. *See also* abstract classes; derived classes
  creating family of, 204–209
  declaring, 45, 109–112
  defined, 42–43, 353–357
  defining, 43–44
  instance of, 119
  multiple, 159–162
  objects and, 109–157
click of mouse button, events for, 314–316
close box, clicking in, 316
Close() function, 336
comments in C and C++ languages, 33, 79
CompuServe online service, downloading QuickTime from, 390–395
constant
  defined, 55
  setting, 55
Constructor sample program
  output of, 148
  source code for, 146–147
constructors, 143–149
  called by new operator, 144, 148
  headers for, 144
  purpose of, 143
  windows and, 327–333
control panel for Simulator C++, xxiv–xxv
  starting topic from, xxvi
CPlusLib library, 9

D

data members
of abstract classes, 201–202
accessing, 134–139, 173–178
as data, 121
defined, 36
defining, 112
pointers as, memory allocation and deallocation for, 151–152
private, 135–137
protected, 175–178
setting and getting value of, 202
types of, 111–112
data types, basic, 51–54
Data window of Symantec debugger, 20–21
updating values in, 25–26
DebuggerDemo program, 20–26
debugging, 23–26
source code of, 22
#define directive, 55
delete operator
calling destructor after use of, 155, 365
memory not released by, 151
to release allocated memory, 103–104, 129
derived classes, 159–197. *See also* classes
creating, 164, 167–168, 204–206
data members and member functions inherited in, 165, 185, 205
example of, 166
member functions of, 206–209
reasons to create, 162–166
returned objects and, 233–238
derived objects, 169–184
creating, 170–171
data member access and, 173–178
examples of, 178–184
member functions used by, 187–188
using, 171–173
DerivedClass1 sample program
output of, 180
source code for, 178–180
DerivedClass2 sample program
derived class in, 181
output of, 184
source code for, 181–183
DerivedClass3 sample program
member functions of, 185–186
output of, 186–187
DerivedRectangle sample program
purpose of, 233–234
results of, 235
source code for, 236–238
DerivedWindows sample program, 345–385
classes for, 353–357, 369
member functions for, 357–362
menus for, 346, 363–365
modifying, 385
objects created in, 365–368
project file for, 369
resources for, 350–354
source code for, 369–385
updating windows in, 362–363
Destructor sample program
output of, 155
source code for, 153–155
destructors, 149–155
calling, 155, 365
headers for, 151
purpose of, 149, 151
Development folder, 2–3
dialog box, handling edit text items in, 293–299
dialog box to create new object, sample, 289–308
radio button items in, 291–293
resources for, 289–291
sample display of, 290
DialogInput sample program
output of, 300
purpose of, 299
source code for, 301–308
disk with this book
project files on, 28
running example programs from, xviii
source code on, xvi, 28
DisposeHandle() function, 405
DisposeWindow() function, 316
DITL resource for a dialog box, 267–268, 280, 290–291, 297–298, 351–353
DLOG resource for a dialog box, 290, 296, 351
do-while loop, 57–59

double variable, 53
drag bar, clicking in, 316
Drag() member function, 335
DragWindow() Toolbox function, 316, 335
DrawMenuBar() function, 406
DrawPicture() Toolbox function, 360
DRVR resource, 406
dynamic binding, 239–264
  example of, 243–248
  implementing, 266
  polymorphism implemented through, 243

E

edit text items in a dialog box, 293–299
else-if branching statement, 60, 63–65
encapsulation
  defined, 38, 137
  public data members as breaking down, 137–139
  as strength of OOP, 251
EndUpdate() Toolbox function, 279, 317
EraseRgn() function, 279
error-checking capability, adding, 299
error messages, compiler, for passing object to a function, 250
error messages window, 17
event loops, 46, 279
events
  update, 276–279
  windows and, 313–317, 335–336
ExitToShell() Toolbox function, 277–278, 336, 365
Extraction dialog box, xxii

F

File menu
  New option of, 12
  Save As option of, 12
files
  adding, to project, 9–15
  decompressing, xxi–xxii
  header (.h), 55
  project (.n), 6, 369
  resource (.rsrc), 267–270, 350
  self-extracting, xxi–xxiii
  source code (.cp), 12–15
FindWindow() Toolbox function, 315, 320
float variable, 53
floating-point number data type, 51
  decimal points not permitted in, 53
folder. *See also* individual folders
  creating, 3
  opening, 3
for loop, 57, 60
forward references
  defined, 230
  function prototypes and, 229–230
FrontWindow() Toolbox function, 365
function overloading, 77, 79–85
  for identically named functions with different argument types, 83–84
  for identically named functions with different numbers of arguments, 79–83
  member function overriding distinct from, 191
  reasons to use, 84–85
function overriding, 187–196
function prototypes
  forms of, 78
  forward references and, 229–230
  in header files, 55
  purpose of, 78
FunctionOverload sample program, 80–83
  dual Draw_Line() functions in, 80–81
  output of, 81
  source code for, 82–83
functions
  defined, 35
  differences between C and C++, 78–79
  different argument types for, 83
  passing objects to, 249–257
  returning objects from, 222–233
  same name for multiple, 79–85

G

GEnie online service, downloading QuickTime from, 395–401
GetMHandle() Toolbox function, 406

GetNewDialog() function, 296
GetNewMBar() Toolbox function, 405
GetNewWindow() Toolbox function, 268, 310, 328
GetPort() function, 278–279
global data
  OOP's elimination of need for, 35
  procedural programming languages' reliance on, 35
GoodCAllocation sample program, source code for, 91–92
GrafPort field, 312
graphics port (GrafPort), 311

H

header (.h) files, 55
HelloWorld sample program
  adding files to project for, 9–14
  converting project source code into application for, 26–17
  creating project folder for, 3–8
  debugging, 19–26
  ending execution of, 17
  error messages for, 17
  running, 17–19, 28
  segmenting project for, 15–17
  source code for, 18
  writing code for, 17–18
HiWord() function, 407
hypertext defined, xxx

I

if branching statement, 60–62
if-else branching statement, 60, 62–64
#include directive, 55–56
inheritance in derived classes, 164–165
instance variables. *See* data members
instances. *See* objects
int variable, variance in size of, 52
integral number data type, 51–52

L

libraries
  adding, to a project, 9–12
  defined, 9
long double variable, 53
long variable, 52
lookup table to determine which function to execute, 243–244
looping statements, 56–60
  purpose of, 56
  types of C, 56–57
LoWord() function, 407

M

MacTraps library, 9
main() function
  assignment statements in, 105
  as nonmember function, 115
malloc() function, 86–87, 89, 118, 127–128
MBAR resource, 351, 403
member access of struct members, 68–71
member functions
  of abstract classes, 202–204
  body of, writing, 115–117
  data members accessed using, 134
  defined, 36
  defining, 112–117
  of derived classes, 206–208
  header of, 112–115, 144
  invoking, 122–127
  listed in class definition, 46
  "normal" functions and, 114–115
  objects and, 121–122
  OOP sample program's, 357–362
  overriding, 185–196
  as part of a class, 112
  as pointers to actual functions, 121
  public, 135, 138
  working with an object's data members, 139–141
member operator (.), 68–69
MemberFunctions sample program
  output of, 125
  source code for, 125–127

memory allocation
in C, 85–99
in C++, 99–104
of pointers used as data members, 151–152
Symantec debugger to examine bad, 94–99
Symantec debugger to verify proper, 92–94
Menu Manager, 405
MENU resource, 403–404
MenuDemo sample program
purpose of, 408
source code for, 409–416
MenuSelect() Toolbox function, 406
menus, 363–364, 403–416
adding menu bars to, 405–408
resources for, 403–405
sample program to create, 408–416
message, function of OOP's, 37–38, 124–125
methods. *See* member functions
movie controller, xxviii–xxix
MultipleClasses sample program
output of, 162
purpose of, 160
source code for, 160–161
MultipleObjects sample program
output of, 134
purpose of, 132
source code for, 132–133

N

new operator
call to constructor by, 144, 148
for memory allocation, 99–104, 121
New Project dialog box, 5–6
NewWindow() Toolbox function, 310, 328
nil pointers
defined, 278, 282
example of, 282–284
NoObjectWindows sample program
output of, 318
purpose of, 317
resource file for, 321
source code for, 321–324
update events received by, 319–320

O

object-oriented language, xv
object-oriented programming (OOP)
advantages of, 38–41, 138
C and, 45–48
code modification in, 40–41
complete example of, 345–385
introduction to, 31–49
as programming methodology, 33
objects, 117–134
candidates for representation as, 45–46
classes and, 109–157
creating, 119, 200, 227–228, 265–273
creating multiple, 43–45, 130–134
data and functions tied together as, 35–36
declaring, 117–120, 127–128
deleting, 129–130
derived class. *See* derived objects
distinguishable, verifying, 333–334
instances of classes as, 119, 122
as key element in OOP, 36
member functions and, 121–122
memory allocation for, 148
OOP sample program's, 365–368
passing, to functions, 249–257
returned from functions, 222–233
shapes as, 222–227
updating, 274–289
users and, 265–308
variables declared as type class as, 120
windows as, 46, 309–343
ObjectWindows sample program
differences between NoObjectWindows program and, 336–337
flaw in, 362
output of, 337
source code for, 337–343
OpenDeskAcc() function, 408
operators
of C language used by C++, 56
defined, 56
table of commonly used, 56

overriding member functions, 186–196
by calling base member function from within derived member function, 208–209
reasons for, 185–187
Overriding sample program
base class in, 191
derived class in, 192–193
output of, 192, 194
source code for, 194–196

P

pages for Simulator C++ software, xxiv–xxxi
Highlight Word, xxviii–xxix
Movie, xxviii
Question, xxx
Status, xxxi
parameter, example of rectangle object passed as, 249–257
PassedRectangle sample program
results of, 253–254
source code for, 254–257
pet shop owner programming example
abstract classes for, 201–204, 209–216, 354–355
class hierarchy for, 216–219
classes defined for, 358, 359, 361
complete example for, 345–385
derived classes for, 162–187, 355
dialog boxes for, 346–348, 365–367
dynamic binding in, 257–263
family of classes for, 204–209
inheritance in, 355
member functions in, 357–362
menus for, 346, 363–365
modifying elements of, 385
objects created in, 365–368
overriding a function in, 187–196
project file for, 369resources for, 350–353, 357
updating windows in, 362–363
windows of pet information in, 348–349, 356
windows of pictures in, 346–349, 357
PICT resource, 353–354, 360, 366
pointers
advantages of, 85
declared to data types, 86
defined, 85
deleting, 130
to derived or base class, 239–241
generic, 86, 88
with new operator, 101–103
nil, 278, 282
review of, 85
struct variables and, 89–92
Symantec debugger used with code that contains, 92–99
polymorphism
defined, 190
implemented through dynamic binding, 243
preprocessor directives, 54–56
private keyword
as access specifier, 168, 178
to limit access, 134–138
procedural programming
approach of, 34–35
code modification in, 38–40
shift away from, 35
procedural programming language, xv, 35
program execution
setting breakpoints during debugging for, 22–24
stepping through single lines of code for, 24–25
programming approaches, procedural and object, 34–41
project
adding files to, 9–15
adding libraries to, 9–12
application created from, 26–27
creating new, 1–8
defined, 1
naming, 3–4, 6
running, 17–19, 28
segmenting, 15–16
updating, 17, 19
Project menu
Build Application option of, 26–27
Run option of, 17–18, 20, 23, 148
Use Debugger option of, 20–21, 23
project window
opening new, 3–4

showing library files, 13
showing size of program's compiled code, 19
showing source code file, 16
protected keyword as access specifier, 168, 175–178
public keyword
effect on data member access of, 175, 178
encapsulation lost by using, 137–139
to set access control, 134–135

Q

QuickTime, xvi, 387–401
to add movie-playing capabilities to system, 387
installing, 401
loaded into memory, 401
places to get, 387–401

R

radio button items in dialog box, 291–293
rectangle object
derived class example of, 233–238
example of, 222–227
as a parameter, 249–257
RectangleClass sample program
purpose of, 224
source code for, 225–226
refCon field
to hold address of an object, 320, 333–334
setting value for, 328–330
to track multiple windows, 313, 320–321, 337
typecasting value in, to object pointer, 331–332
resource file, 267–280
contents of, 350
naming convention for, 270
resources
alert, 267–268
dialog box, 289–291
IDs for, 350
menu, 403–405
OOP sample program's, 350–353
return keyword, purpose of, 203, 227
ReturnDerivedPet sample program
output of, 259
pointer assignments in, 258
source code for, 260–263
ReturnDerivedRect sample program
output of, 245
source code for, 246–248
ReturnedRectangle sample program
output of, 231
source code for, 231–233
routine
called in a procedural language, 37
defined, 35
OOP message telling object to use specified, 37, 124–125

S

scope resolution defined, 77
scope resolution operator, 105–107, 113–114, 206–207
ScopeOperator sample program
output of, 106
source code for, 106–107
segments, dividing project into, 15–17
Select() member function, 332
SelectWindow() Toolbox function, 316
SetMenuBar() function, 405
SetPort() function, 278–279
SetWRefCon() Toolbox function, 328–329
SetWTitle() Toolbox function, 313
shape as an object, 222–226
short variable, 52
Simulator C++ software tutorial, xvi
chapters of, xxvi
features of, xx
folder of, xxiii
installing, xx–xxiii
QuickTime used by, 387
running, xvii, xxiv
selecting topic from, xxvi–xxvii
using, xviii–xx, xxiv–xxxi
sizeof() function, 86–87
source code. *See also* individual sample programs
application created from project's, 26–27
for book's programs on disk, xvi

compiling, 17
files of, added to C++ project, 12–15
Source menu
Add Files option of, 9–10, 15
Browser option of, 216–219, 369
Source window of Symantec debugger, 20–21
setting breakpoints in, 23
Standard Libraries folder, 12
stdlib.h header file required to use malloc() function, 89
string data type, 51, 53
given a value after it is declared, 54
struct C language data type
as basis for C++ class data type, 51, 71–72
defining and declaring, 67–68
member access for, 68–71
Symantec debugger used with code that contains, 92–99
struct variables, 120
need for using malloc() eliminated by using, 128
pointers and, 89–92
structure template, 67–68
declaring, 118–119
structures, 67–75
body of, 68
defined, 67
members of, 68
subclasses. *See* derived classes
Supplemental Note, xxix
switch branching statement, 60, 65–67
Symantec C++ 6.0, creating project using, 3–5
Symantec C++ 7.0, creating project using, 5–8
Symantec debugger, 19–26
for code that contains pointers and structs, 92–99
to examine bad memory allocation, 94–99
starting, 148
to verify a proper memory allocation, 92–94
SystemClick() Toolbox function, 315–316

T

THINK C compiler, 3
THINK Project Manager
checking proper class definitions using, 369
editing a source code file using, 1–2
as environment of the Symantec C++ compiler, 2
running example source code using, xviii–xix
viewing class hierarchy using, 216–219
this operator
to refer to address of object, 328–329
as reminder that member function is working with an object, 139–143, 192
value of, displayed in Data window of the debugger, 333
Toolbox calls, C++ use of, 46
TrackGoAway() Toolbox function, 316
typecasting a generic pointer, 88, 91

U

update events
redrawing window's contents and, 276–279, 319
updateEvt for, 314, 316–317
UpdateObject sample program
purpose of, 280–281
resources for, 280
source code for, 284–289
user, objects and, 265–308

V

variable
monitoring value during debugging of, 23
sensible name for, 143
virtual keyword to inform compiler of multiple versions of a function, 190, 192

W

WaitNextEvent() Toolbox function, 276–278, 318
while loop, 57–58
whole number data type, 51–52
WIND resource, 353, 408
creating, 310
to define window attributes, 268–270
Window Manager, supplying data to, 310
window object, representing, 325–326
window record, 310
WindowPeek data type, 310, 312–313, 331

WindowPtr data type, 310, 312–313, 328, 331
WindowRecord data type, 310–311
  memory allocation for, 327–328
windows
  basic techniques for handling, 309–324
  clicking in, 314–316
  constructor function and, 327–333
  data types for, 310–313
  defining types of, 320
  events and, 313–317, 335–336
  multiple, 317–324
  as objects, 46, 309–343
  opening, 310
  pointers to, 328–332
  redrawing contents of, 274–289
  updating obscured, 316–317, 362–363

Prima Computer Books You Can Order Directly

WINDOWS Magazine Presents: Access from the Ground Up	$19.95
Adventures in Windows	$14.95
CompuServe Information Manager for Windows: The Complete Handbook & Membership Kit (with two 3½" disks)	$29.95
Computers Don't Byte: The Absolute Beginner's Guide to Getting Started with the PC	$7.95
Computing Strategies for Reengineering Your Organization	$24.95
CorelDRAW! 5 Revealed!	$24.95
Create Wealth with Quicken	$19.95
DOS 6: Everything You Need to Know	$24.95
DOS 6.2: Everything You Need to Know	$24.95
WINDOWS Magazine Presents: Encyclopedia for Windows	$29.95
Excel 4 for Windows: The Visual Learning Guide	$19.95
Excel 5 for Windows: The Visual Learning Guide	$19.95
Free Electronic Networks	$24.95
WINDOWS Magazine Presents: Freelance Graphics for Windows: The Art of Presentation	$27.95
Improv 2.1 Revealed! (with 3½" disk)	$27.95
Internet After Hours	$19.95
Lotus Notes 3 Revealed!	$24.95
Making Movies with Your PC	$24.95
Microsoft Office In Concert, Professional Edition	$27.95
NetWare 3.x: A Do-It-Yourself Guide	$24.95
Novell NetWare Lite: Simplified Network Solutions	$24.95
Paradox 4.5 for DOS Revealed! (with 3½" disk)	$29.95
PageMaker 5 for Windows: Everything You Need to Know	$19.95
PC DOS 6.1: Everything You Need to Know	$24.95
PowerPoint: The Visual Learning Guide	$19.95
QuickTime: Making Movies with Your Macintosh	$24.95
Smalltalk Programming for Windows (with 3½" disk)	$39.95
Software Developer's Complete Legal Companion (with 3½ disk)	$32.95
Superbase Revealed!	$29.95
Think THINK C! (with two 3½" disks)	$39.95
Visual Basic for Applications Revealed!	$27.95
Windows 3.1: The Visual Learning Guide	$19.95
Windows for Teens	$14.95
WinFax PRO: The Visual Learning Guide	$19.95
Word for Windows 2: The Visual Learning Guide	$19.95
Word for Windows 2 Desktop Publishing By Example	$24.95
Word for Windows 6: The Visual Learning Guide	$19.95
WordPerfect 5.1 for Windows Desktop Publishing By Example	$24.95
WordPerfect 6 for DOS: The Visual Learning Guide	$19.95

To order by phone with Visa or MasterCard, call (916) 632-4400, Monday–Friday, 9 a.m.– 4 p.m. Pacific Standard Time.

To order by mail fill out the information below and send with your remittance to: Prima Publishing, P.O. Box 1260, Rocklin, CA 95677-1260

Quantity	Title	Unit Price	Total
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____
			_____
		Subtotal	_____
		7.25% Sales Tax (CA only)	_____
		Shipping*	_____
		Total	_____

Name _____

Street Address _____

City _____ State _____ ZIP _____

Visa/MC No. _____ Expires _____

Signature _____

*$4.00 shipping charge for the first book and $0.50 for each additional book.

About This Disk

The 1.4MB disk that accompanies this book contains one self-extracting, compressed file, which, when expanded, will give you 4MB of software. Because it is self-extracting, no additional software is required to decompress this file—everything you need is right here on this one disk. The decompressed software consists of the Simulator C++ software tutorial and the Symantec C++ project and source-code files for plenty of example Macintosh programs.

The Simulator C++ software is a Macintosh program created especially for *Symantec C++: Object-Oriented Programming Fundamentals for the Macintosh.* Through the use of text, graphics, QuickTime movies, and questions, this software tutorial enhances the C++ and OOP concepts covered in the book. To run the Simulator C++ program, you will need a Macintosh with any version of System 7. The Simulator will run on either a black-and-white or a color system with any size monitor. Instructions for the program's use appear in the book's introduction.

Each of the over 30 C++ example programs used in this book also appear on the disk, which saves you a lot of typing. You will find that the Symantec project file and source code file for each example will work with either version 6 or 7 of the Symantec C++ compiler.

Disclaimer and Notice of Limited Warranty

The enclosed disk is warranted by Prima Publishing to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disk combination. During the sixty-day term of limited warranty, Prima will provide a replacement disk upon the return of a defective disk.

The remedy for breach of this limited warranty shall consist entirely of replacement of the defective disk and shall cover no other damages, including loss or corruption of data, changes in the functional characteristics of the hardware or operating system, deleterious interaction with other software, or any other special, incidental, or consequential claims that arise.

Prima and the author specifically disclaim any and all other warranties, either express or implied, including warranties of merchantability, suitability to a particular task or purpose, or freedom from errors. In no event shall Prima or the author be liable for damages in excess of the purchase price paid for the book/disk combination, even if Prima and/or the author have previously been notified that the possibility of such damages exist.

BY BREAKING THIS SEAL, YOU AGREE TO ABIDE BY THE CONDITIONS SET FORTH IN THE DISCLAIMER AND NOTICE OF LIMITED WARRANTY FOUND IN THIS BOOK.



PRIMA

PRACTICAL PROGRAMMING SERIES

Master the Language of the Future *Today!*

Symantec C++: Object-Oriented Programming Fundamentals for the Macintosh is your comprehensive guide to the world of C++ object-oriented programming. You learn the basics of this dynamic language through the use of step-by-step examples designed not only to hone the skills you already possess but also to provide you with the skills that will soon be required of all programmers. Best of all, the example codes contained in the book and accessory disk will run on both the Mac and the Power Mac!

Bonus Interactive Software Enhances the Learning Process!

The companion 1.4-MB high-density disk contains all example source codes used throughout the text and features Simulator C++, a friendly, multimedia software tutorial enlivened with Apple's QuickTime® movies. Use this state-of-the-art software to leaf through animated pages and "bring to life" key programming concepts—learn by doing!

This book, in combination with the multimedia software, enables you to:

- **Electronically walk through—*at your own pace*—scores of interactive examples, each more challenging than the last**
- **Learn C++ using a hands-on, *entertaining* approach**
- **Fully realize the potential of your Macintosh or Power Mac**
- **Build up a storehouse of programming skills to prepare you for the object-oriented future**
- **Acquire absolute fluency in C++ with unmatched speed *and* ease**

COVER VERSIO 6

DAN PARKS SYDOW, author of **Think THINK C!** (also from Prima), is a professional software engineer who has developed educational software for the Macintosh. He currently is a writer of Macintosh programming books.

User Level: Intermediate

PRIMA COMPUTER BOOKS
An Imprint of Prima Publishing

U.S. $39.95
U.K. £37.49 Net (Inc. VAT)
Can. $55.95

ISBN 1-55958-633-8
9 781559 586337 53995>

Macintosh/Programming