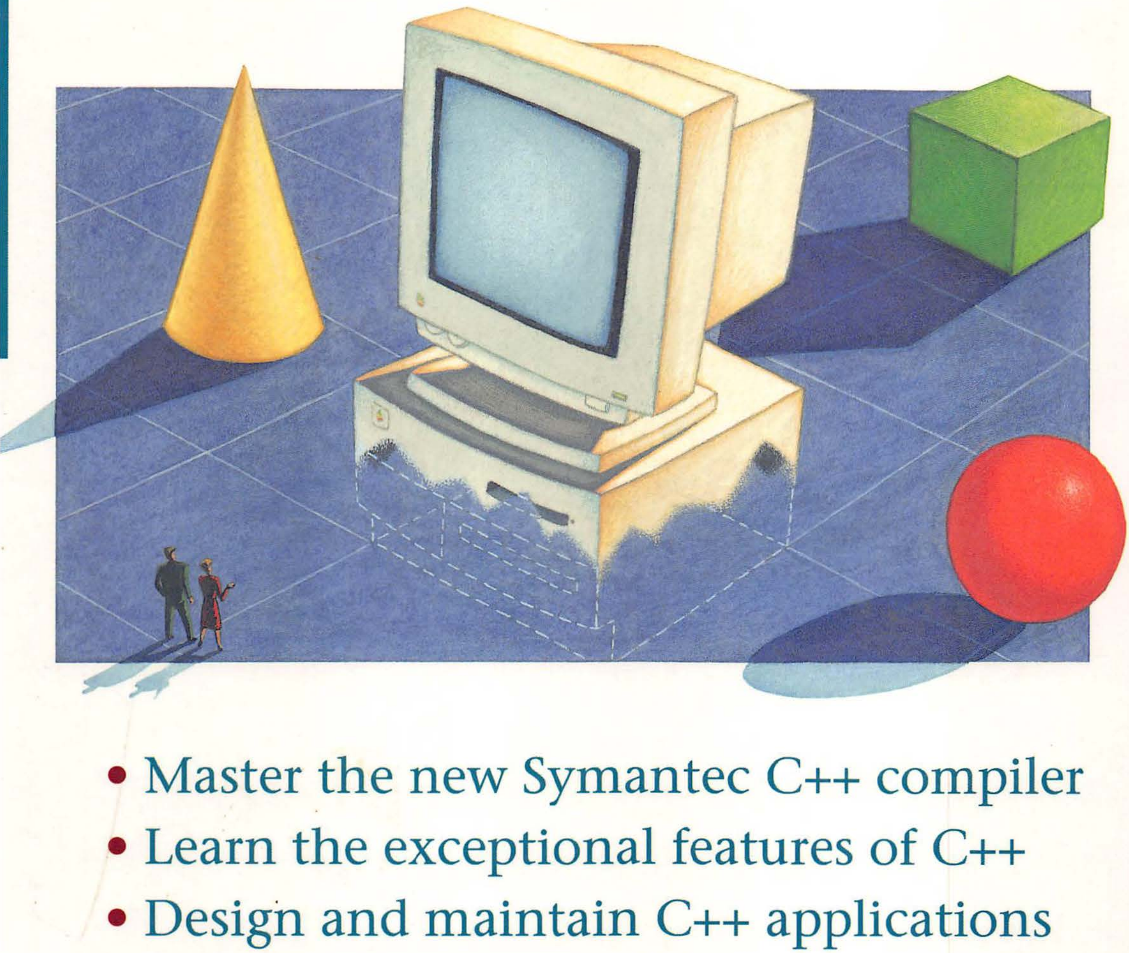
New Technology Building Blocks

DISK INCLUDED

Symantec C++ for the Macintosh: The Basics



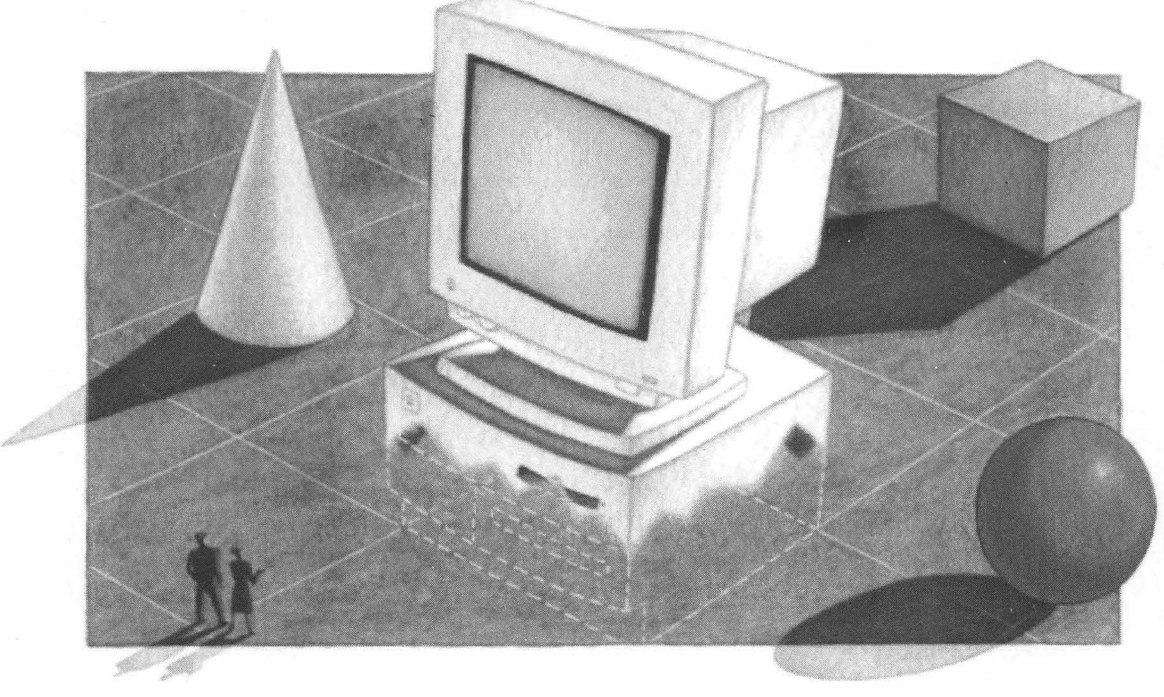
- Master the new Symantec C++ compiler
- Learn the exceptional features of C++
- Design and maintain C++ applications

M&T BOOKS

John May & Judy Whittle

Symantec C++ for the Macintosh: The Basics

Symantec C++ for the Macintosh: The Basics



John May & Judy Whittle

M&T BOOKS

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

Printed in the United States of America

Limits of Liability and Disclaimer of Warranty
The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

ISBN 1-55828-276-9: $39.95

Publisher: Steve Berkowitz
Associate Publisher: Brenda McLaughlin
Development Editor: Margot Owens Pagan
Production Editor: Mark Masuelli
Assistant Production Editor: Joseph McPartland
Copy Editor: Andrea Salvatore
Technical Editor: Ray Valdés

95 94 93 4 3 2 1

Table of Contents

Acknowledgments...xvii

Preface xix

Why This Book is For You..1

Introduction 3

Prerequisites—Software and Hardware Required.................................5
How the Book Is Organized...5
Object-Oriented Methodology–A Powerful Approach 6
C++ 6

Chapter 1: Basic Programming Concepts 7

Numbering Systems..7
Decimal Numbers...8
Binary Numbers 8
Hexadecimal Numbers 9

Octal Numbers....................11
ASCII Characters....................12
Bits, Bytes, and Nibbles....................13
Kilo, Mega, Giga....................14
Signed and Unsigned Numbers....................15
Ones Complement....................16
Twos Complement Rule....................17
Logic....................17
AND Operation....................18
OR Operation....................19
NOT Operation....................21
XOR Operation....................21
Summary....................22
Exercises....................22

Chapter 2: Object-Oriented Development 27

Procedural versus Object-Oriented Programming....................28
Procedural Programming....................28
Object-Oriented Problem Solving....................28
Putting to Rest an Old Myth....................29
OOP and C++....................31
Advances in Development Environments....................32
Objects and Classes....................33
Objects....................33
Classes....................34
Messages and Methods....................35
Encapsulation....................35
Class Diagrams....................35
Methodology....................38
Specifying Fields and Methods....................38
Subclassing and Inheritance....................38
Abstract Class....................39
Overriding....................40
Multiple Inheritance....................40
Polymorphism and Dynamic Binding....................41

Iterator Methods 42
Where to Begin an Object-Oriented Program 44
Summary....................45
Exercises—Programming with Objects....................45

Chapter 3: The Symantec C++ Environment....................47
Getting Started with Symantec C++ 48
Creating a Project....................48
Segments 51
Creating a Source File 52
Adding Libraries 53
Compiling the Program....................54
Running the Program 55
Building an Application 55
Summary....................57
Exercises 57

Chapter 4: Fundamentals of C++ 59
Comments 60
Statements....................61
White Space 62
Variables 62
Reserved Variables....................63
Predefined Variable Types 63
Declarations 66
Definitions 66
Initialization....................66
Signed and Unsigned....................67
Specified Constants 68
Logical Values....................69
Strings 69
String Constants 70
Scope 70
Operators 70
Unary Prefixed Operators....................74

Deferencing — * 74
Address of — & 75
Negative — - 75
Ones Complement — ~ 75
Logical NOT — ! 75
Increment — + + ...76
Decrement — - - 77
Size of — size of() 78
Cast to — (cast) 78
New Operator — new 79
Delete Operator — delete 79
Unary Postfixed Operators 80
Function Call — ( ) 80
Array Subscript — [ ] 81
Dynamic Arrays, New 82
Dynamic Arrays, Delete 83
Direct Selection — 83
Indirect Selection — - > 84
Increment — + + ...84
Decrement — - - 85
Binary Operators—Arithmetic and Logical 85
Addition — + ...86
Subtraction — - 86
Multiplication — * 86
Division — / 87
Modulus — % 87
Bitwise AND — & 87
Bitwise Inclusive OR — | 88
Bitwise Exclusive OR(XOR) — ^ 89
Left Shift — << 90
Right Shift — >> 90
Equal (or replacement) — = ...91
Binary Operators—Assignment 91
Addition Update — += 93
Binary Operators—Comparison 94

Logical AND and Logical OR....................96
Ternary Operator — ?:....................97
Comma Operator....................98
Summary....................99
Exercises....................99

Chapter 5: Controlling the Program Flow.............103

Statements....................104
Expression and Null Statements....................104
Blocks....................104
If....................105
If–Else....................108
Else–If....................109
Switch....................110
Switch–Break....................112
Which Do I Use—Switch or If-Else?....................114
While....................115
Do–While Loop....................116
For Loop....................116
While versus For....................118
Break....................118
Continue....................119
Labels and Goto....................119
Style....................121
Error from Fortran....................121
Rule 1: Placement of Curly Braces....................122
Rule 2: Use Curly Braces on All If Tests....................123
Rule 3: Space Between Statement Keywords and Parentheses....................125
Rule 4: No Space Between Function Name and Parentheses....................125
Rule 5: Use Spaces Between Binary Operators....................125
Rule 6: Use Spaces After Commas and Semicolons....................126
Rule 7: Capitalize Every Main Word in a Function Name....................126
Rule 8: Capitalize Every Main Word in a Variable Name
Except the First....................127
Rule 9: Use Blank Lines Only When They Convey Meaning....................127

Rule 10: Go Easy on the Use of Underscore (_) 128
Rule 11: Use a Break on the Last Case of a Switch Statement.............128
Rule 12: Operators in Definition and Declaration Statements 129
Summary...129
Exercises 130

Chapter 6: Functions and Variables 133

Functions 135
A Function Definition 135
A Function Definition Example...136
Function Prototypes..136
Variable Number of Arguments...137
Passing Function Arguments..138
Passing by Value 139
Passing by Pointer...139
Passing by Reference...140
Default Arguments 141
Passing Multiple Values 142
Explicit Void..142
The Stack...142
The Heap...146
C and Pascal on the Macintosh...146
Procedures and Functions...147
Order of Parameters Pushed 148
Type Conversion 149
Variable Storage Types..149
"auto" Variables...150
Static Variables 150
External Variables..151
Register Variables 152
"const" Variables..154
Reference Declarations 155
Right–Left Rule 156
Function Overloading 158
Scope Resolution Operator..159

Inline Functions 159
C++ Preprocessor 160
Conditional Directives 161
C++ Preprocessor Examples: 161
#define versus const and inline 162
The Preprocessor and Comments 163
Summary 163
Exercises 163

Chapter 7: Input/Output Streams 165

Streams 166
I/O Channels 166
Predefined I/O Stream Manipulator 167
Generalized Escape Sequence 167
Formatted Output 168
Advanced Formatting 169
Input Stream Example 170
I/O on the Macintosh 171
Summary 171
Exercises 172

Chapter 8 Advanced Data Structures 173

Pointers 174
Initializing a Pointer 175
Void Pointers 176
Arrays 177
Initializing Array Values 178
Initializing String Arrays 178
Indexing Arrays 179
Array Assignment 179
Multidimensional Arrays 180
Free Store Operators 181
Testing for Space 181
Destroying Heap Variables 181
When to Use Dynamic Memory Allocation 183

Enumerated Values 183
Enumerated Variables 184
Enumerated Types 185
"typedef" 185
Structures 186
Declaring a Structure 187
Anonymous Structures 188
Referencing Data Elements of Individual Structures 188
Padding 189
Creating an Array of Structures 189
Structures and Bit Fields 191
Unions 192
Operator Overloading 193
Member Functions and Structures 194
Summary 197
Exercises 197

Chapter 9: Classes in C++ 199

Defining a Class 200
Classes and Structures 201
Data Hiding 201
Objects 202
Member Functions 202
Constructors 203
Destructors 204
Initializing Functions 205
Accessor Methods 206
Collaborators 208
Friends 208
Friend Class 208
Friend Functions 209
Data-Hiding Convention 209
Summary 210
Exercises 210

Chapter 10: Linked List Example 213

Statement of the Problem 214
Creating a Circular List 215
Adding and Deleting Nodes 215
Building the Code 216
    Creating a Friend Class 217
    Insert Function 218
    Extract Function 219
    InputData Function 220
    DisplayData Function 220
    Destructor 221
Writing the Main Function 222
Summary 223
Exercises 224

Chapter 11: Subclassing and Inheritance..............225

Defining Derived Classes 226
Virtual Functions 227
    Pure Virtual Functions 227
Protection Keyword 228
Constructors in Derived Classes 231
Destructors in Derived Classes 232
"I" Functions in Derived Classes 232
Static Members 235
    Static Member Functions 236
    When to Use Static Members 237
Creating Objects (Instances) 237
    Dynamic Object 238
    Initializing Instances 238
    Using Instances 238
Which Functions? 238
Which Data? 240
Accessing Member Data 240
The C++ "this" Pointer 241

Summary....243
Exercises....243

Chapter 12: Phonebook Example....245

Source File Organization....246
Naming Conventions....246
Class Diagrams....247
PhoneBook Project....247
UString.h....247
UString.cp....249
UTelephone.h....251
UTelephone.cp....251
UPhoneBook.h....254
Class TEntity....254
Class TPerson....256
Class TCompany....256
UPhoneBook.cp....257
PhoneBook.cp....257
Pause Routine....258
Summary....259
Exercises....259

Chapter 13: Advanced Features of C++....261

Inline Functions in a Class....262
Operator Overloading in Classes....262
Pointers and Objects....263
Shape Example....270
Templates....281
Template for a Routine....282
Template for a Class....282
Multiple Inheritance....284
Ambiguities of Data Members in Multiple Inheritance....285
Ambiguities of Member Functions in Multiple Inheritance....286
Summary....288
Exercises....288

Chapter 14: ToolBox, Memory and Symantec C++ 289

Using the ToolBox from C++ 290
Pointers and Dynamic Memory 290
Master Pointers 290
Dereferencing a Handle 293
The Keyword "inherited" 294
Summary 295

Chapter 15: Using Symantec C++ 297

THINK Project Manager 298
Source Menu 298
Adding, Removing and Getting Information on Files 299
Debug 300
SourceServer 300
Checking the Syntax, Preprocessing, and Disassembling 300
Precompiling, Compiling, and Making a Project 301
Browser 301
Project Menu 302
Closing the Project 302
Setting the Project Type 303
Bringing the Project Up to Date 304
Using the Debugger and Running the Program 305
Search Menu 305
Go To and Marking 306
Edit Menu 307
Specific Edit Items 307
Tabs and Fonts 307
Shift Left and Shift Right 308
Balances 308
Options Menu 309
THINK Project Manager Options 309
Preferences 309
Editor 311

Debugging 311
Extensions 313
Project Window 313
.o Converter 314
Symantec C++ 315
Language Settings 315
Compiler Settings 316
Code Optimization 317
Debugging 318
Prefix 319
THINK C 320
THINK Rez 320
File Menu 321
Windows Menu 322
THINK Debugger 323
Source Window 323
Data Window 324
File Menu 325
Edit Menu 325
Debug Menu 325
Source Menu 327
Data Menu 327
Windows Menu 328
Summary 329

Appendix A: Glossary 331

Appendix B: Bibliography 337

Index 341

Acknowledgments

We'd like to thank the following people for all their help with this book:

M&T Books for giving us the opportunity to write this book.

The folks at **Symantec Corporation** for their wonderful product and their assistance in this project.

Tony Meadow, **Randy Matamoros**, **David Taylor**, and **John Wilkinson** of Bear River Institute for invaluable cooperation and help with all aspects of Symantec C++ 6.0.

Larry Horwitz and **Ray Valdés** for their detailed technical review and edit of the manuscript.

Tom Condon of Becton Dickinson who kept us apprised of ongoing changes in the product.

The University of California, Berkeley, especially the students and faculty in the Department of Electrical Engineering and Computer Science for their feedback on the manuscript.

Carole McClendon of Waterside Productions for hooking us up with M&T Books and her assistant **Belinda Catalona** for keeping us on track.

Margot Pagan, Project Editor, and **Mark Masuelli**, Production Editor, of M&T Books for all their fine work, pleading, and encouragement throughout the writing process.

Mary and **Shawn May** and **Pat** and **Paul Whittle** for their assurance, inspiration, and support, without which this book might have been impossible.

A big thanks to you all.

Preface

There are many object-oriented programming languages(OOP), including Object Pascal, Object Modula, Eiffel, Objective C, Self, Simula, Smalltalk, Common LISP with CLOS, and C++. Some are quite old, others are new. For example, Smalltalk was defined in 1972, while Self was invented in the last10 years. This should tell you that object-oriented programming is not new, but that it has taken some time for it to become mainstream technology.

Object-oriented programming is quickly becoming mainstream technology. I believe that the primary reason is that it provides a better way to manage complexity. The rapid evolution of desktop applications (contrast Word version 1 and Word version 5), and the operating systems (contrast the first version of the Macintosh operating system and System 7) over which they preside, provides numerous examples of large, complex software packages that have become increasingly difficult to maintain and enhance. The structured programming techniques that are now conventional wisdom are no longer able to help us manage software that is this complex. Object-oriented programming is not a panacea, and will not solve all of our software development woes. It is, however, the next step in the evolution of software development technology.

The old myths about object-oriented programming (that it produces slower code which takes up a lot of disk space, etc.) die hard. Examples abound of high-performance applications that provide sophisticated interfaces and perform many complex calculations that were implemented using object-oriented technology. Adobe's Photoshop and Ray Dream's Designer are examples of such applications that were implemented with C++ and MacApp, Apple's application framework.

Apple Computer has encouraged developers to adopt object-oriented technology for at least five years now, after experimenting with it for more than 10 years. As a result of this, object-oriented programming is mainstream for commercial and in-house development in the world of Macintosh software. If you look at the new Macintosh applications from the last couple of years, that is, applications which have been introduced and not those which were enhanced, a significant percentage of them have been developed using object-oriented programming languages.

During the last five years, C++ has become the primary object-oriented programming language. Market forces determine much of the technology that we use (videotape and audiotape formats come to mind here), perhaps more often than we'd like to admit. Being honest about it, market forces have also selected C++ as the object-oriented language that most programmers will be using during this decade. C++ is a complex language, as complex as COBOL or Ada, albeit complex in different ways than those languages. As such, it is too complex to learn the entire language at one time.

John and Judy's book will help you gain a solid mastery of the basic features of C++. Once you feel comfortable with the topics covered in this book, you should be ready to approach other topics, such as learning an application framework like the THINK Class Library, MacApp, or Bedrock. Only after that, should you learn the more advanced features of C++.

Once you are using C++, you'll be able to participate in using some of the most interesting technology to come along. Application frameworks are collections of classes that provide the standard behaviors of an application. MacApp, Apple's current object-oriented application framework, provides all the code to manage memory, desk accessories, multiple windows, printing, undo and redo, and many other behaviors. MacApp is about to be supplanted by a joint development by Symantec and Apple called Bedrock. This application framework, written in C++, will allow you to more easily develop applications for both Macintosh and Windows. Once you have a Bedrock-based application running on one of these platforms, it will only take a small amount of work to have it running on the other.

Next, about two years ago Apple and IBM established a joint venture named Taligent. The people at Taligent are developing an object-oriented operating system in C++. They are also working on a powerful development environment that will be tightly integrated with the operating system. This will enable you to develop complex applications in less time and with less effort than with any other current software development technology.

Anthony Meadows
Series Editor

Why This Book Is For You

Here is a book you need to learn C++ programming on the Macintosh. This hands-on tutorial teaches you C++ programming from the ground up, taking you from the fundamentals of object-oriented programming to the advanced features of C++. Special focus is given to Symantec C++, the latest compiler for Macintosh programming. Through detailed discussions and solid programming examples you'll gain a thorough understanding of Symantec C++ and will be on your way to designing efficient C++ applications.

This book is filled with programming examples you can study and learn from. The source code has been written to compile and run using Symantec C++ and is provided on the enclosed disk.

If you are:

- A software developer for the Macintosh
- A corporate in-house programmer, scientist, or engineer
- Someone who wants to learn more advanced skills for customizing an application in Symantec C++
- A programmer who wants to learn techniques, beyond those presented in the product manual

...this book is for you.

And, even if you're someone who doesn't know anything at all about programming in C++, but want to write programs for the Macintosh, Symantec C++, will put you ahead of the game.

If you're already programming in C, and want to learn C++ because it's the programming language of the future and you want to design your programs as a collection of objects to make them easier to write, modify and maintain; Symantec C++ is a much more effective, and complete, object-oriented language than C.

Or, perhaps you are a programmer who is using Zortech C++ and running it under MPW—you already know something about C++, object-oriented programming and the Class Library—you will want to know about Symantec C++ for the Macintosh.

Symantec C++ for the Mac: The Basics features

- The new Symantec C++ compiler
- the basics of programming
- language extensions of C++
- explanations of encapsulation and data hiding
- examines inheritance
- explains polymorphism, exploration of dynamic binding
- Helps you to understand data structures, functions and variables: an in-depth explanation of their structure of classes. Each chapter features a summary of the information and exercises to help you along.

Introduction

This book is about programming the Macintosh in C++, but it is also about a unique and exciting product: Symantec's new Symantec C++ for the Macintosh. The product is unique because there isn't another C++ compiler for the Mac that doesn't require the Macintosh Programmers Workshop (MPW) or is completely stand-alone. Symantec C++ is exciting because it comes from those wonderful folks who gave you THINK C, the most versatile, complete, economical, and popular C programming package for the Mac. (THINK C is so popular that many programmers who are required to develop an application under MPW first develop on THINK C and then port their applications. The new Symantec C++ will run under both the Finder and MPW, something that was not possible before.)

The intention here is to give you an in-depth presentation on the product itself, as well as the C++ language, and introduce you to object-oriented programming,. With the aid of this book, you will:

- Learn programming basics.
- Master the new Symantec C++ product.
- Comprehend object programming and design concepts.
- Discover the language features of C++.
- Learn how to author an object design using C++.

As a teaching vehicle, this book differs from other volumes on C++ in form and order of presentation. The book begins with an introduction to the concepts of object-oriented programming and the C++ language, goes into a detailed description of Symantec C++, and introduces you to some advanced features of C++. The advantage here is that you can ease into object-oriented programming from the very beginning, rather than wading through long dissertations on the language, the Macintosh Toolbox, and an application framework first.

You do not have to be a programmer, or even know anything about programming, to use this book. However, there is an underlying assumption that you already know the Mac fairly intimately. And that you may be a registered software developer for the Mac, a corporate in-house programmer, a scientist, engineer, or a general user wanting to learn more advanced skills for customizing an application in Symantec C++. It's likely that you'll fit into one of the three following categories:

1. A person who doesn't know anything about Symantec C++, or even C++, but wants to learn how to program the Mac. Chances are you know that Symantec C++ is going to be the major compiler on the market. You'll want to cover every chapter in the book, from programming basics to advanced features of Symantec C++.
2. A programmer who has THINK C and wants to know what's different about Symantec C++. You'll also want to learn more about object-oriented programming and the Class Library. You may want to skip over Chapters 1, 4, 5 and 6, and concentrate instead on Chapter 2 and the more advanced features of C++.
3. A programmer who has been using Zortech C++ and is running it under MPW. You already know something about C++, object-oriented programming, and the Class Library. You're mainly concerned with using the Symantec C++ product. Chapters 3 and 15 will be especially useful to you.

Prerequisites—Software and Hardware Required

The first thing you need is Symantec C++ from Symantec. If you're not a programmer, we'll bring you up to speed on programming conventions.

For practical purposes, we recommend the following Mac hardware:

- 4 MB of RAM, 5 to 8 preferred.
- A fast hard disk with at least 20 MB of free space.
- A fast processor to reduce compile times.

For software, we highly recommend *THINK Reference,* also from Symantec, a comprehensive guide to system information. This package gives you:

- Detailed routine descriptions, declarations, and notes on the Mac operating system.
- Technical notes, example code, and tips from Symantec engineers.
- Sections on fonts, resource types, and other Mac topics.
- Graphics to illustrate key concepts.

THINK Reference also provides *Speed Search,* a method of finding a topic quickly by typing in just a few characters. Topics are organized by trap names, managers, keywords, data interfaces, and structures.

All in all, *THINK Reference* is an essential tool for any serious Mac programmer.

How the Book Is Organized

The objective of the book is to familiarize you with the product, Symantec C++, and to teach you to use that product in programming applications. To accomplish this, the book has the following mix of material: 25% on Symantec C++ (the compiler, editor, preprocessor, assembler, header files, linker, and debugger), 50% on the C++ language (how it differs from C, small and large enhancements, etc.), and 25% on object-oriented programming.

Object-Oriented Methodology— A Powerful Approach

Object-oriented methodology, which is on the cutting edge of programming concepts, accomplishes three main goals:

- It provides natural modeling of real-world processes.
- It encourages and supports reuse.
- It enforces modularity.

Its main concepts are to:

- Create programming objects that correspond to real-world objects.
- Give those objects the ability to store data and to respond to messages.
- Reuse existing objects in whole or in part without changing the existing objects.

If object-oriented programming (OOP) seems abstract, and you find that you have difficulty in knowing how to apply the concepts, don't despair! It will all become evident to you as you proceed through the book—especially when you start to add member functions to data structures (Chapters 8 and 9).

C++

C++ is a powerful successor language to C, and is a better C. (Actually, the term C++—that is, C followed by the increment operator—means "one better than C" or "one more than C.") C++ is actually a hybrid language that supports both procedural and object programming. You can use C++ for data abstraction and for object-oriented programming extensions. Currently, C is the most popular language on both the Mac and the PC. C++ will probably be the closest thing to an industry standard object-oriented language. If you are using C++ and object-related technologies, you are on the leading edge.

1 Basic Programming Concepts

This chapter is a review of the basics of computer programming. If you have already done programming, this material will be familiar to you. If you're comfortable with the basics, you can skip to Chapter 2, which introduces object-oriented programming.

Numbering Systems

We've all been told that computers are dumb and a program is only as smart as the person who wrote it. But we don't often think about computers being stupid until we realize that they understand only one thing: numbers. Furthermore, they recognize only two numbers, 0 and 1.

Each line of code that you write translates into numbers and, consequently, bits or bytes of memory. This section presents the numbering systems the computer uses, along with their associated codes and symbols, and a brief description of their logical use.

Decimal Numbers

The decimal (meaning "pertaining to ten") numbering system, which is the most common system in use, is sometimes referred to as a *base 10* number system. Historians and anthropologists agree that the system developed as it did because humans have 10 fingers. Each of the 10 numbers is represented by an Arabic (or Hindu-Arabic) figure, and early records show that the system was introduced in Europe in the 12th century.

In base 10, the number 9732_{10} is evaluated as shown in Table 1.1.

Table 1.1

1000	100	10	1
10^3	10^2	10^1	10^0
9	7	3	2
(9 x 1000) +	(7 x 100) +	(3 x 10) +	(2 x 1) +
9000 +	700 +	30 +	2

In the far right (ones) column, 10^0 means simply "1 with no zeros after it." The next column to the left is a tens column, the next a hundreds column, and the last (far left) a thousands column. By multiplying 9 by 1000, 7 by 100, 3 by 10, and 2 by 1, and then adding the results together, you get the number 9732_{10}.

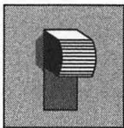
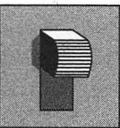
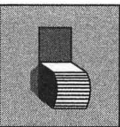
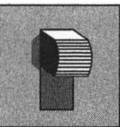
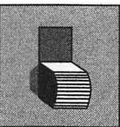
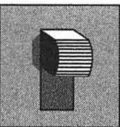
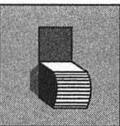
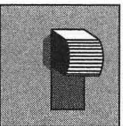
In this book, if a number is not preceded by a 0 or a 0x or followed by a subscript to describe what it is, the number is assumed to be decimal. (In C, any number that is preceded by a 0x is assumed to be hexadecimal, and any number preceded by a 0 is octal. These bases are discussed further on in this chapter.)

Binary Numbers

The binary (meaning "pertaining to two") numbering system is the system used by computers. This is because the simplest state in which an electric circuit exists is: *on* or *off*. The two binary numbers are represented by the Arabic figures 0 and 1. This numbering system is sometimes referred to as a *base 2* number system.

In similar fashion to the evaluation of the decimal number in the previous table, the number 11010101_2 (with the subscript 2 indicating base 2) is evaluated in Table 1.2.

Table 1.2

							
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	1	0	1	0	1	0	1
1 x 128	1 x 64	0 x 32	1 x 16	0 x 8	1 x 4	0 x 2	1 x 1

The icons in the top row of the table represent switches that are either on (the button is up) or off (the button is down). If the switch is off, that column represents a zero, or nil. You can also think of the 1 as a magnetized tape and the 0 as a nonmagnetized tape, or the 1 as an electrical charge and the 0 as no electrical charge. On a CD-ROM (read-only memory) for example, a laser beam is reflected off a mirrored surface. If the reflection goes in one direction, it represents a 1. If it goes in the other direction, it represents a 0.

Look at the first column (far left). The second row states that the value of this column is 2 to the 7th power, and the third row declares that the product of 2^7 is 128_{10} (2 x 2 x 2 x 2 x 2 x 2 x 2 = 128). The fourth row shows that because the switch is on, the number 128_{10} is a valid number. In the third column, the switch is off, so the number 32_{10} is invalid (0 x 32_{10} = 0). By adding 128_{10}, 64_{10}, 16_{10}, 4_{10}, and 1_{10}—the results of the multiplication in the last row of the table—you get the number 213_{10} in the decimal system.

Note:

By the way, the terms most significant bit (msb) and least significant bit (lsb) are used with binary numbers. In the binary table above, the msb is 2^7, and the lsb is 2^0.

Hexadecimal Numbers

The *hexadecimal* (meaning "pertaining to sixteen") numbering system aids people in understanding the number system used by computers. In fact, it is used as a shorthand notation for binary numbers and is known as the *base 16* or *hex* number system. Each of the 16 numbers is represented by an Arabic figure or alphabetical character: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Depending o the system, a hex number may be displayed on a computer as $A87D, 0xA87D, Z'A87D, or A87D. It may also be written as $A87D_{16}$.

To convert binary numbers to hex, collect the binary numbers into four groups of four, starting from the right-hand side (lsb). Keep grouping your way across until you run out of bits. If you have 1 to 3 bits left over, use leading zeros. Then, replace each binary group with a hex equivalent. To convert hex numbers to binary, replace each hex digit with four binary characters. (see Table 1.3).

Table 1.3

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

1010	1000	0111	1101
A	8	7	D

Alpha eight seven dog is easier to say (and comprehend) than one zero one zero one zero zero zero zero one one one one one zero one.

To convert the hexadecimal number to the decimal system, you can create a table similar to the tables evaluating decimal and binary numbers (see Table 1.4).

Table 1.4

16^3	16^2	16^1	16^0
4096	256	16	1
A	8	7	D
A x 4096	8 x 256	7 x 16	D x 1
40960 +	2048 +	112 +	13 +

By adding together the numbers in the bottom row, you get the decimal equivalent of A87D$_{10}$, which is 43133$_{10}$.

Octal Numbers

The octal numbering system (meaning "base 8") is not as common as the hexadecimal, but both C and C++ do make use of it. Like the binary and hexadecimal systems, the octal system is a power of 2. With 3 bits, a total of $2^3 = 8$ possible numbers can be represented. These are the eight digits from 0 through 7.

To convert binary numbers to octal, collect the binary numbers into groups of three, starting from the right-hand side (lsb). Keep grouping your way across until you run out of bits. If you have one or two bits left over, use leading zeros. Then, replace each binary group with an octal equivalent. To convert octal numbers to binary, replace each octal digit with three binary characters. The relationship between octal and binary numbers is shown in Table 1.5.

Table 1.5

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

111	110	011
7	6	3

To convert the octal number to the decimal system, you can create a table similar to the tables evaluating decimal, binary, and hexadecimal numbers. In Table 1.6, we convert 763_8 to its decimal equivalent.

Table 1.6

8^2	8^1	8^0
64	8	1
7	6	3
7 x 64	6 x 8	3 x 1
448+	48+	3+

By adding together the numbers in the bottom row, you get the decimal equivalent of 763_8, which is 499_{10}.

As is the case with hexadecimal numbers, octal numbers are used in place of a binary number to make a quantity or code easier to work with and remember.

ASCII Characters

ASCII (pronounced *ass-key*) stands for the American Standard Code for Information Interchange. An ASCII character is an 8-bit code that is used to represent not only numbers but also letters (both lower-nd uppercase), special symbols, and control functions.

Examples from the extended Macintosh character set are:

A is 0x41

B is 0x42

a is 0x61

b is 0x62

1 is 0x31

2 is 0x32

Carriage return is 0x0D

Space is 0x20

™ is 0xAA

∑ is 0xB7

The standard ASCII numbers use the lower 7 bits of a byte and ignore the first bit. (The first bit is reserved for a signed bit, which we discuss later.) The Macintosh uses all 8 bits (the numbers 00 to FF) to represent an extended set of ASCII digits. The Mac keeps the standard table but adds 80 through FF for special characters, for a total of 256 ASCII characters.

Bits, Bytes, and Nibbles

Here, in abbreviated form, is the long and the short of bits, bytes, and nibbles:

- 4 bits equal a nibble (sometimes spelled nybble).
- 8 bits equal a byte (never spelled bite).
- 2 nibbles equal a byte.

```
1010      1000
A           8
nibble    nibble
       byte
```

- 2 bytes is equal to a 2-byte word called a short int (short for integer) in C++. It is called an integer in Pascal.

```
0xA83F
```

- 4 bytes is equal to a 4-byte word called a long int in C++. It is called a long integer in Pascal.

```
0x84A326AD
```

Sometimes a 2-byte word is just called a word, and a 4-byte word is called a long.

Table 1.7 shows the relationships of integer types in these languages: 68000 Assembly Language (ASM), Pascal, Fortran, and C++. The last three have an informal relationship with the 68000 Assembly Language but not with each other. (However, C++ has a relationship with Pascal through the Toolbox.) The reason Pascal and Fortran have a relationship with 68000 ASM is because the microprocessor on the Mac is a Motorola 68000, and they're all running 68000 code.

Table 1.7

Relationships of Integer Types				
	68000 ASM	**Pascal**	**Fortran**	**C++**
1 Byte	Byte	byte	Integer *1	char
2 Bytes	Word	integer	Integer *2	short (int)
4 Bytes	Long	long int	Integer *3	long (int)

The thing to remember here is that *Word*—whenever you are speaking of microprocessors or computers—is the size of an instruction. On the 68000, an instruction is 16 bits long. This becomes important when you talk about the *word size* of a computer. (On a Cray computer, the word size is 64 bits; on a VAX, it is 32 bits; and on the old 8080 microprocessors running on CP/M, a word was 8 bits.)

Kilo, Mega, Giga

The terms *Kilo, Mega,* and *Giga* are a type of shorthand or "techese." Kilo means thousand, Mega means million, and Giga means billion. However, in computerese, the terms stand for the actual values shown in Table 1.8.

Table 1.8

	Scientific and Engineering		**Computer Science**	
Kilo (K)	1,000	10^3	2^{10}	1,024
Mega (Meg)	1,000,000	10^6	2^{20}	1,048,576
Giga (G)	1,000,000,000	10^9	2^{30}	1,073,741,824

- 800K bytes on a floppy are 800 x 1,024 bytes, or 819,200 bytes.
- 20Meg hard drive is 20 x 1,084,576 bytes, or 20,971,520 bytes.
- Half Gig CD ROM is 0.5 x 1,073,741,824 bytes, or 536,870,912 bytes.

Signed and Unsigned Numbers

A number is signed or unsigned depending wholly on how you use it. By default, the computer thinks integer variables are signed, which means that they can represent both positive and negative numbers. Unsigned numbers can only be positive. If you're not going to be using negative values (that is, in the range of -32,768 to +32,767), you can use unsigned numbers to force the compiler to read the contents of a variable to be in the range of 0 to 65,535. Here are some rules of thumb about computing unsigned and signed numbers:

- *Unsigned Byte*—To compute the decimal equivalent, you simply convert the binary number to decimal.
- *Signed Byte*—If the first bit (most significant bit or msb) is zero, compute the decimal equivalent. 0x00 to 0x7F is a number from 0 to 127. Or, if the first bit (msb) is one, invert all the bits, add one, compute the decimal equivalent, and make that number negative. This operation is called *twos complement* and is simply a method for representing the values of negative numbers. Hence, 0x80 to 0xFF is a number from –128 to –1.

Table 1.9

Ones Complement

Largest Positive Number

+127	0111 1111
+126	0111 1110
	.
	.
	.
+1	0000 0001
0	0000 0000
–0	1111 1111
–1	1111 1110
	.
	.
	.
–127	1000 0000

Largest Negative Number

If there is a twos complement, there must be a *ones complement*, right? There is, and the operation consists of simply inverting all the bits (a *complement*) and adding the negative sign. At one time, the ones complement representation was widely used in digital computers. But there was a definite problem with it, as you can see from in Table 1.9.

Ones Complement Rule

- If sign bit is zero, just convert to decimal.
- If sign bit is one, invert all bits and convert to decimal.

As you can see, the ones complement allows for both a positive and a negative 0, which cannot be. Twos complement takes care of the problem by getting rid of the negative 0, as shown in Table 1.10.

Table 1.10

Twos Complement

Largest Positive Number

+127	0111 1111
+126	0111 1110
	.
	.
	.
+1	0000 0001
0	0000 0000
–1	1111 1111
–2	1111 1110
	.
	.
	.
–128	1000 0000

Largest Negative Number

Twos Complement Rule

- If sign bit is zero, just convert to decimal.
- If sign bit is one, invert all bits, *add one,* then convert to decimal.

Ones complement has a minus range of –0 to –127, while twos complement has a minus range of –1 to –128.

Therefore:

- Unsigned bytes have a range from 0 to 255.
- Signed bytes have a range from –128 to +127.
- Unsigned words have a range from 0 to 65,535.
- Signed words have a range from –32,768 to 32,767.
- Unsigned long words have a range from 0 to 4,294,967,295.
- Signed long words have a range from –2,147,483,648 to 2,147,483,647.

Logic

The system of logic we know was developed by the ancient Greeks as a branch of philosophy. It's doubtful that any of the Greek philosophers foresaw the fruits of their efforts used in electronic circuitry in the 20th century, but who knows?

Specifically, logic can be applied to an electric circuit in its simplest state; that is, a switch is either on or off. If the switch is off, it is also open. Figure 1.1 shows a diagram of a simple electric circuit with a battery, an open switch, and a light bulb.

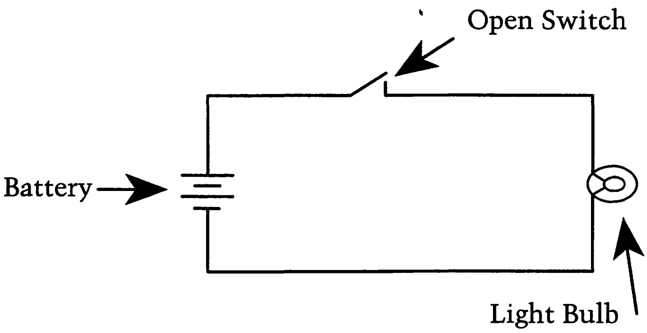


Figure 1.1 *Open-Switch operation.*

As you can see, when the switch is open, the light bulb is *off* because no current can flow through the circuit, and no voltage is applied to the light bulb. (See Figure 1.2.)

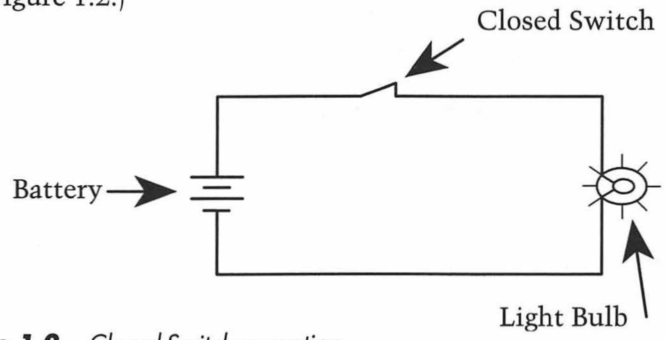


Figure 1.2 *Closed-Switch operation.*

In the closed-switch operation, the switch is closed and the light bulb is on.

If *open* and *off* are represented by 0, and *closed* and *on* by 1, they can be represented by Table 1.11.

Table 1.11

Switch	Bulb
0	0
1	1

AND Operation

Figure 1.3 represents an **AND** operation, where both A and B must be closed for C to be on. The **AND** operation in C programming is represented by an & (ampersand) or by a && (double ampersand), depending on whether the operation is caried out on bit-values (the bitwise AND) or truth-values (the logical AND).

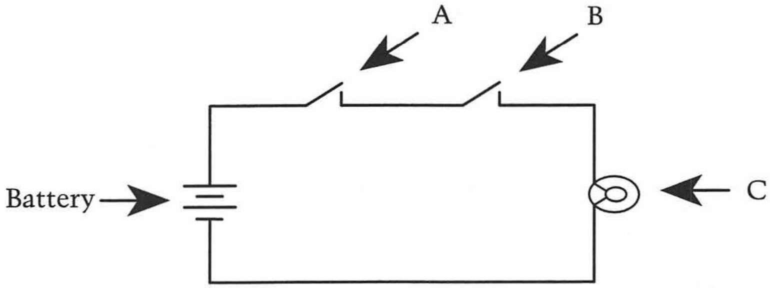


Figure 1.3 *AND operation.*

The truth table for Figure 1.3 follows:

Table 1.12

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

In column A in the table above, we have the state of switch A. Column B represents the state of switch B, and column C represents the light bulb as either off or on. Remember that 0 is off and 1 is on. This table shows that both switch A and switch B must be closed for C to be on.

$$1100_2 \ \& \ 1010_2 = 1000$$

$$\begin{array}{rr} & 1100 \\ \& & 1011 \\ \hline & 1000 \end{array}$$

In the stacked formula above, start with the far right-hand bits. Check the truth table to see what 1 & 0 (C programming uses the & symbol to represent the AND operation) produce in column C. (They produce a 0.) Then move to the left, adding the bits in each column, and checking the truth table for the answer. When you reach the far left bits (1 & 1), you see that the result is 1, and the light bulb is on.

Try this exercise:

$$A5_{16} \ \& \ 5A_{16} = 00_{16}$$

In the example above, convert the hex numbers to binary, and then AND them together.

OR Operation

Figure 1.4 represents an **OR** operation, where either A or B can be closed for C to be on. The **OR** operation in C programming is represented by a | (vertical bar).

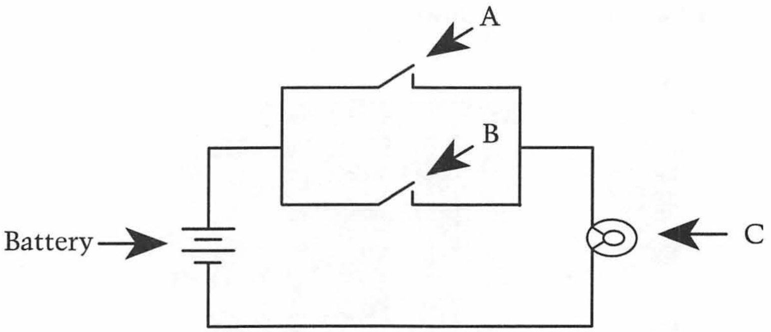


Figure 1.4 *OR operation.*

The truth table for Figure 1.4 follows:

Table 1.13

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

$$1100_2 \ | \ 1011_2 =$$

$$\begin{array}{rr} & 1100 \\ | & 1011 \\ \hline & 1111 \end{array}$$

As you did for the AND formula, start with the right hand bits, look at the truth table for the answer, and put that number on the total line.

Try this exercise:

$$A5_{16} \ | \ 5A_{16} = FF_{16}$$

NOT Operation

The truth table below represents a **NOT** operation, where you invert the bits from 0 to 1 and vice versa. The **NOT** operation in C programming is represented by a ~ (tilde).

Table 1.14

A	B
0	1
1	0

$\sim1011_2 = 0100_2$

Try this exercise:

$\sim5A_{16} = A5_{16}$

Note:

All numbers in the C++ language are assumed to be in the decimal system unless prefixed by the letters "0x"

XOR Operation

The truth table below represents an **XOR** operation. The **XOR** operation is the same as an **OR** operation, but in the case where both A and B equal 1, the result will be 0. The **XOR** operation is represented by a ^ (caret).

Table 1.15

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Summary

In this chapter, you've learned the basic concepts of programming:

- Numbering systems.
- ASCII characters.
- Bits, bytes, and nibbles.
- Signed and unsigned numbers.
- Logic operations and truth tables.

An understanding of these basics is absolutely essential before you can develop proficiency in writing code. Now that you have this under your belt, try the following exercises. Then let's move on to the concepts of object-oriented programming in Chapter 2.

Exercises

1.1 Convert the following binary numbers to decimal:

(a) 1010_2
(b) 111_2
(c) 01011010_2

1.2 Convert the following decimal numbers to binary:

(a) 23_{10}
(b) 100_{10}
(c) 145_{10}

1.3 Convert the following octal numbers to binary:

(a) 123_8
(b) 7642_8
(c) 3527_8

1.4 Convert the following binary numbers to octal:

(a) 11001_2
(b) 110110101_2
(c) 10011101_2

1.5 Convert the following octal numbers to decimal:

(a) 35_8

(b) 342_8

(c) 1234_8

1.6 Convert the following decimal numbers to octal:

(a) 42_{10}

(b) 1260_{10}

(c) 4235_{10}

1.7 Convert the following binary numbers to hexadecimal:

(a) 1111_2

(b) 10101010_2

(c) 1111101110101111_2

1.8 Convert the following hexadecimal numbers to binary:

(a) $A6_{16}$

(b) $A04_{16}$

(c) $7AB4_{16}$

1.9 Convert the following decimal numbers to hexadecimal:

(a) 123_{10}

(b) 2352_{10}

(c) 3619_{10}

1.10 Convert the following hexadecimal numbers to decimal:

(a) $A13B_{16}$

(b) $E9_{16}$

(c) $7CA3_{16}$

1.11 Represent the following numbers as ASCII characters:

(a) 41_{16}

(b) 61_{10}

(c) 250_{10}

1.12 What are the hexadecimal and decimal values of the following ASCII characters:

(a) A
(b) a
(c) ™

1.13 Divide the following bits into nibbles:

(a) 11100100_2
(b) DEADC0DE_{16}

1.14 Divide the following into words (short):

(a) 1111110110110110111_2
(b) F00DFACE_{16}

1.15 Divide the following into longs:

(a) A6_{16}
(b) $\text{FEDCBA9876543210}_{16}$

1.16 How many bytes are in each of the following:

(a) 800K
(b) 520Meg
(c) 3Gig

1.17 Find the eight-bit one's complement form of the following numbers:

(a) FF_{16}
(b) -10_{10}
(c) 123_{10}.

1.18 Find the eight-bit two's complement form of the following numbers:

(a) FF_{16}
(b) -14_{10}
(c) 99_{10}

1.19 AND the following numbers:

(a) ABCD16 & EF12_{16}
(b) 9316 & DE_{16}
(c) 10102 & 0101_2

1.20 OR the following numbers:

(a) 6A16 | DD_{16}

(b) 0AAC16 | 8427_{16}

(c) 10012 | 0110_{2}

1.21 NOT the following numbers:

(a) $FFFF_{16}$

(b) AA_{16}

(c) 11100111_{2}

1.22 XOR the following numbers:

(a) BE16 ^ BC_{16}

(b) 9BC816 ^ $FFFF_{16}$

(c) 10012 ^ 1011_{2}

2 Object-Oriented Development

Object-oriented programming (OOP) is a design methodology that incorporates several sophisticated and efficient mechanisms for managing the complexity of present-day application development. Object program design models the world as a collection of *objects* that interact by passing messages back and forth. Programming with objects is not complicated but merely different from traditional, procedural-based programming (which has been *the* method for programming until recently). Because the concepts of object-oriented programming are different and may at first seem somewhat obtuse, we do not expect you to grasp everything in this chapter right away. Instead, you may want to read through this chapter once and then come back to it again after you have read the chapters on structures (Chapter 8) and classes (Chapter 9).

Object-oriented programming, which arrived on the scene in 1971, is the next step beyond procedural programming, a kind of natural progression born of necessity. Apple has made its message clear: OOP is the wave of the future for Macintosh applications, and Apple's primary internal development language is C++ and is object oriented.

Procedural versus Object-Oriented Programming

Procedural programming treats action and data as two separate entities; that is, you define data structures and then develop a set of routines (or perhaps a library), which in turn operate on data that you pass into them as arguments. In object-oriented programming, when you define the structures, you define their actions at the same time. Instead of routines acting on data, you have sets of objects interacting with each other.

Procedural Programming

With procedural programming, you define variables to represent the data used by your program. You might even group related variables into structures or records. You then write subroutines, procedures, or functions that operate on those variables.

Procedural-based programming is a time-consuming, multiphased process but, nonetheless, is currently the most common type of programming. The following are typical phases of a procedural-based development methodology:

1. State the problem.
2. Analyze to obtain a feel for what needs to be done by talking with users.
3. Discuss all possible solutions and identify the most effective solution.
4. Generate a high-level design of the proposed solution.
5. Using the high-level design as a guide, generate a detailed design for the subsequent solution.
6. Begin programming.
7. Commence integration and testing.
8. Conduct Alpha and Beta testing.
9. If there are any problems in any of these phases, loop back and start all over again.

Object-Oriented Problem Solving

The premise of object-oriented problem solving is that the best way to develop software is not to develop but to reuse. Also, object-oriented programming models the world as a collection of *objects* that interact by passing messages back and forth. OOP defines classes to model real-world data (the classes provide automatic data modularity). You then write the methods (i.e., functions) that operate on the data in your classes.

Some typical benefits that derive from object-oriented methodology are listed below:

1. The OOP approach encourages the use of modern software engineering technology. Because it is a very structured system for describing a problem and the solution to that problem, it requires you to work within constraints and rules. And, it makes it easier for other programmers to aid in the implementation of the specific object-oriented design. As an example, suppose that you wanted to write a computer program that would compose a book. Assume, then, that you say to the program, "Write a book." Where would it begin? If you say, "Write a romance," you put a constraint on the program. If you then say, "Use only English words," and "Confine it to 250 pages," more constraints are added. These constraints make it easier for programmers to use tools, and this focusing by programmers leads to even more innovative CASE tools.
2. Another benefit of OOP is that it promotes and facilitates software reusability. It's conceivable that in many instances OOP projects will require only 10 to 20% new code. The rest is reusable.
3. When well done, OOP solutions more closely resemble real-world solutions; in other words, the solutions are more natural.
4. OOP results in software that is easily modified, extended, and maintained because it enforces modularity. If you represent the OOP units as black boxes that are independent of one another, they are easy to modify and maintain because you only have to work on one at a time. They are also easy to extend because all you have to do is add more black boxes; that is, you don't have to make modifications before you add something.
5. OOP results in a significant reduction of integration problems for the same reasons as item 4 above.

Putting to Rest an Old Myth

In the beginning, object-oriented programming was slower than procedural programming and took up more memory. Over the past several years, great strides have been made in compiler technology to optimize code. Consequently, tremendous gains are achievable by making the overall structure of today's programs object oriented.

There are, however, places within your code where you may have concerns about speed, memory, and disk space (specifically with certain numerical algorithms), and you may want to use procedural based programming in these areas.

Because of the immense increase in the capability of hardware, you do not have to be as concerned about speed and memory as you once might have been. In general, use object-oriented programming when you:

- Have a boss or client that is breathing down your neck to finish the program.
- Require easy error checking.
- Want easy maintenance.
- Want a modular structure.
- Want good user interface.
- Want to run on all Macs and perhaps other platforms.

There is an analogous story that made the rounds of the integrated circuit design firms in Silicon Valley in California. Designers were desperate to get more and more devices on a silicon wafer, to make that jump from 100 devices (small-scale integration, or SSI) to 1 million per wafer (very large scale integration, or VLSI). As the designs for the masks became ever more complex, the engineers involved had to start pasting the circuitry maps first on a table, then the floor, and finally the walls and ceilings. The size of the map got so big that engineers started talking about hiring a "tall, thin designer. He would have to be 30 feet tall in order to see out over the entire map, and he would naturally be thin because the firm wouldn't pay enough for him to eat properly."

The point is well taken, though, that technological advances occurred so rapidly that firms became hard-pressed to keep up not only with the technology but with the costs involved. Carver Mead (a professor of Computer Science, Electrical Engineering, and Applied Physics at the California Institute of Technology), estimated that if you looked at the designs of the masks and equated the density to be that of a normal city—that is, eight city blocks to a mile—the map in 1963 would cover a city the size of Pasadena. In 1978, the map would equal the size of the Greater Los Angeles area, in 1985 the equivalent of California and Nevada combined, and in 1990 the continent of North America. In the meantime, the costs of the capital equipment for making a VLSI chip were approaching $1 billion, leaving almost no margin for error. In order to stop the spiraling costs and to get a higher yield of dies per wafer, chip makers began putting constraints on the designers by coming up with a set of rules to design by. In this respect, they were willing to sacrifice the efficiency of packing more transistors into an integrated circuit in trade for making the design process much easier and more reliable. And this is what object-oriented programming is all about: imposing constraints and setting standards for coding. In object-oriented programming, we are willing to sacrifice memory and performance in order to produce more reliable programs that do more things. This is not a great sacrifice because of extensive changes in technology in the past few years: memory is becoming considerably less expensive than it was five years ago, and CPUs are substantially faster.

OOP and C++

OOP has a set of commonly used terms (even though there is no standard set of concepts and terms). C++ employs the same concepts but uses a slightly different terminology. In this book, when we discuss the field of object-oriented programming that is not language specific, we use the common terminology. When we discuss C++, we introduce the C++ terms and stick with them. In the following table, we present both general OOP terms and C++ terms so that you can make mental translations if necessary.

Commonly Used OOP Terms	C++ Terms
Method	Member function
Instance variable	Data member
Class	Class
Superclass	Base class
Subclass	Derived class
Object	Instance or object

C++ does not include every concept proposed as part of object-oriented programming. As shown in Figure 2.1, persistence, delegation, and genericity are the three main aspects not encompassed by C++.

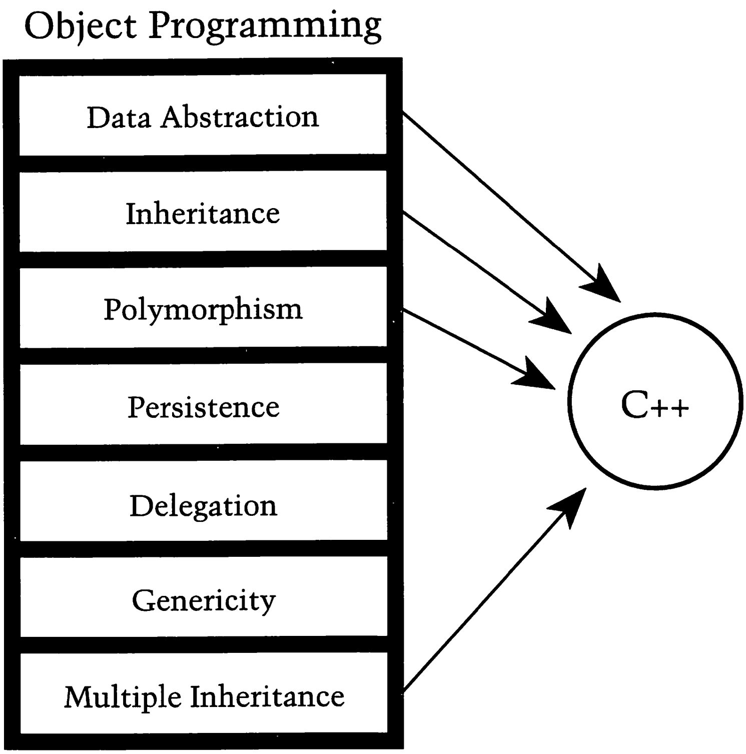


Figure 2.1 *Object-oriented programming aspects included in C++.*

Advances in Development Environments

Up to this point, there have been two major development environments for the Mac: MPW (Macintosh Programmers Workshop) from Apple and THINK from Symantec. Apple supplied C and Pascal compilers and an assembler for their MPW environment and later a C++ compiler. In addition, there have been third-party suppliers whose products run under MPW, namely Ada, Modula, FORTRAN and Zortech C++.

The MPW development environment is quite versatile and powerful, but it is a difficult system to use. MPW has many similarities to UNIX, so many of the line-oriented commands are analogous to UNIX.

THINK runs under the Finder. Unlike MPW, it is paradoxically more Mac-like: it has the Mac human interface built right into the environment. It does not have the same power as MPW, but is much easier to use.

Symantec has had two compilers under the THINK environment: THINK C and THINK Pascal. Over time, the company has modified both to support object-oriented programming. Symantec C++ implements some of the features of C++ not found in C; that is, polymorphism, multiple inheritance, friends, and overloading. In 1991, Symantec purchased Zortech, and Symantec C++ is the result of mixing Zortech C++ and THINK C.

With the advent of the Macintosh, Apple developed a set of routines (in a ROM chip inside of every Mac) that contain something called the *Toolbox*. The main purpose of the Toolbox is to provide a set of routines for consistent user interface. Apple also created a library of classes under MPW to interface with the Toolbox called *MacApp*. Symantec, in the meantime, developed its own class library, which is called Think Class Library (*TCL*). Unfortunately, since there were no standards set for developing class libraries, the Apple and Symantec libraries were incompatible. Subsequently, Apple and Symantec got together to develop an application framework, not just for the Mac but also for use on other platforms like the PC, probably to run with Windows. An application framework is a type of class library that you build on to develop your own application. This framework is called Bedrock. It will allow source code to be transferred to run on other platforms using Bedrock, but each platform will retain its individual interface characteristics: a window on the PC will be a PC window, and a window on the Mac will be a Mac window.

Objects and Classes

All OO programming involves objects (of course), the classes that objects belong to, and all the things that objects can do.

Objects

What is an object? An object is a programming construct that can do useful work. It is also a mechanism for modeling things in the real world, such as people, places, and things. All of these can be manipulated as objects in software. If you're writing a program to inventory different models of cars, for instance, you can use each model of car as an object.

To get an object to perform some operation, you send it a message telling it exactly what you want it to do. The object has methods that are used to respond to the messages. Each object has its own private memory (internal data) and local functions (methods). Every object has a name,

Note:

The information in an object will be hidden unless you are in the process of designing a class.

which is sometimes referred to as an *object reference variable.* An object is a self-contained unit of information (modularity), where the data is protected (information hiding); that is, the object determines the method that will be used, and that information is hidden from you—it's none of your business. The same thing is true for what data the object will use.

Additionally, the data that the object uses can be private information. In other words, the object's internal data is private and any "internal" methods that an object uses can also be hidden, so all you have to do is send it a message to do something. It is up to the object to figure out how to do it.

Classes

One useful way to think about objects and classes is to compare them with variables and structures. To see the comparison, examine the following code:

Class

```
class TMonster
{
  private:
    RGBColor fColor;
    short fNumOfEyes;
  public:
    void HideUnderBed();
    void MakeScaryNoises();
};
```

Type

```
typedef struct
{
  short top;
  short left;
  short bottom;
  short right;
} Rect;
```

Objects of the same class have identical properties. Objects of the same class will have their own copies of a set of common data, and they each share a common set of methods. So, all the monsters may have color and eyes (common data), and they could all have the same way of making scary noises (methods). When you create a new class of objects, you must define the internal variables and the set of messages, and you must write the methods used by the objects. The internal data (variables) contained in an object are called *instance variables.*

Variables and methods are defined for an entire class. Each object, which is an instance of a class, will have its own, unique instance variables. Each will "point" to a single copy of code for shared methods. Put another way, each object has unique internal variables but shares methods with other objects of the same class.

An object is an instance of a class. A class is to an object as a data type is to a variable. For example, you can say: *CookieMonster* and *GroverMonster* are each monster objects. They are each instances of the class *TMonster*. (In this book, the convention is that the names of classes begin with an uppercase T.)

Messages and Methods

To program with objects, you create an object of the class you need. You then send that object messages describing what you want it to do, and the object responds to those messages by performing some operation, which is the method. A message must be addressed to a specific object. In C++, a message is the name of a function with any associated arguments. For every message, there is an associated response. For example:

```
void TMonster::EatCookies (short type)
{
    .
    .
    .
}
```

You can think of messages as function calls, and methods as function definitions. A method is defined to be the function executed by the object in response to a message. Also, a method is a service that an object performs.

Encapsulation

Encapsulation is simply a method of packaging instance variables and method names together as an object. In other words, the data is encapsulated into the object, and messages are then used to manipulate the data. This encapsulation of data enables information hiding. It also defines an interface.

Class Diagrams

Class diagrams are useful for program design and program documentation because they allow you to see the structure of your application. They are used mostly at the beginning (at the time you lay everything out) and at the end of the project, when you test and document it.

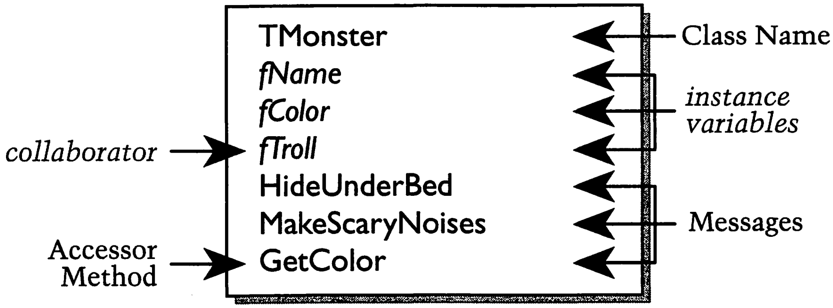


Figure 2.2 *Class diagram.*

In the class diagram shown in Figure 2.2, *TMonster* is the class name. In this book, the convention is to have class names begin with an uppercase *T* for type. They appear in bold letters, always at the top of the class diagram.

Things that begin with *f* are internal variables, in this example *fColor, fName,* and *fTroll.* These are only of interest to the person who has to write a message for an object. Sometimes the internal variables are called *instance variables;* at other times they're called *fields.* By convention, these are represented in the class diagram in italics.

Some internal variables may be references to other objects—here, *fTroll.* These references are sometimes called *collaborators.*

Messages represent jobs that the objects can be asked to do. Message names are also the names of the functions that implement the jobs, in this example *HideUnderBed* and *MakeScaryNoises.* These functions are called *methods,* which are represented by normal type and are always located at the bottom of the class diagram. Methods (which have the same name as messages) may be underlined. This underlining can be useful because the number of underlined methods defines how much work you will have in writing the corresponding routines.

Sometimes an object provides a copy of an instance variable. This is done through a special message called an *accessor method.* In Figure 2.2, *GetColor* is an accessor method.

It is important to remember that messages are not global procedures. Messages must be sent to instances or instantiated objects. (To instantiate means to make an object from a class.) These instantiated objects incorporate the compiled methods, as shown in Figure 2.3.

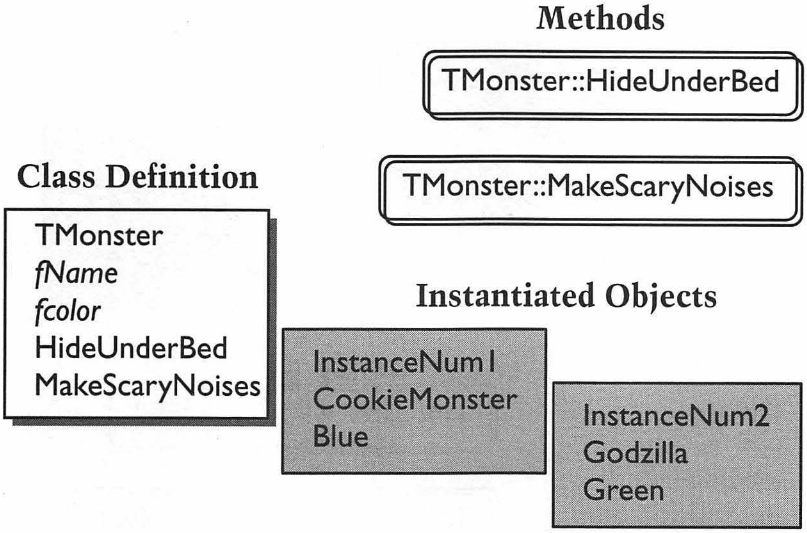


Figure 2.3 *Classes versus objects.*

In the above figure, we created a class called *TMonster* that contains internal variables and the names of the methods used by *TMonster.* Next we created two instances of this class (objects) called *CookieMonster* and *Godzilla.* We also wrote the methods (*TMonster:HideUnderBed* and *TMonster:*

MakeScaryNoises) that will respond to the messages sent to the objects. To send a message to *CookieMonster*, you write:

```
CookieMonster.HideUnderBed
```

Fields and methods are defined for the entire class. Each object (instance of a class) will have its own unique field data. Also, each object will "point" to a single copy of code for the shared methods.

Note that objects *send* messages, too, as shown in Figure 2.4.

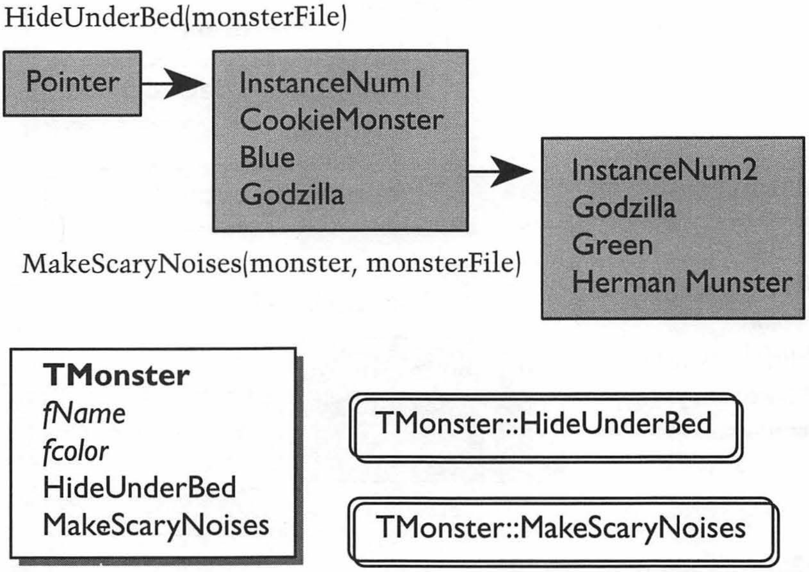


Figure 2.4 *Example of objects sending messages.*

Figure 2.4 illustrates one of the most powerful features of C++: the master pointer. Here, the pointer is a pointer to an object of class *TMonster.* It not only points to objects of class *TMonster,* it points to objects that are subclasses of *TMonster* as well and uses the *same* message.

Methodology

If you do not have the kind of object you need, you must define a new class of object. You also have to define the internal variables that the object is going to use and the messages that will be sent to the object, and you must write the methods that the object will use to implement the messages.

Specifying Fields and Methods

The fields of a class often correspond to real-world data. Sometimes they store temporary data used by the methods. Each message has to have a method. Since methods only perform one task, they should be small in size.

Subclassing and Inheritance

Classes have fields and procedures; they resemble an advanced form of data hiding, which is called *data abstraction.* But classes offer much more than just data hiding. You can create a new class from an existing class and reuse most of the class's methods and instance variables. The mechanism for this is called *subclassing*. Subclasses inherit their behavior from the parent class. The subclass will have all the instance variables that the parent class has but can add others. Subclasses can also alter their behavior without modifying the parent class by adding new methods or overriding old methods. Figure 2.5 shows an example of subclassing.

In the figure, the new class that is created is called a subclass, and the parent class is called a superclass. The reason the arrow points upward is that the subclass inherits variables and methods from the superclass; the superclass code does not even know that the subclass exists.

Subclassing offers a way to add new methods without affecting the behavior of the original class, so it is useful when you want to leave the original class alone. It is also a great way to make a versatile library (called a *Class Library*).

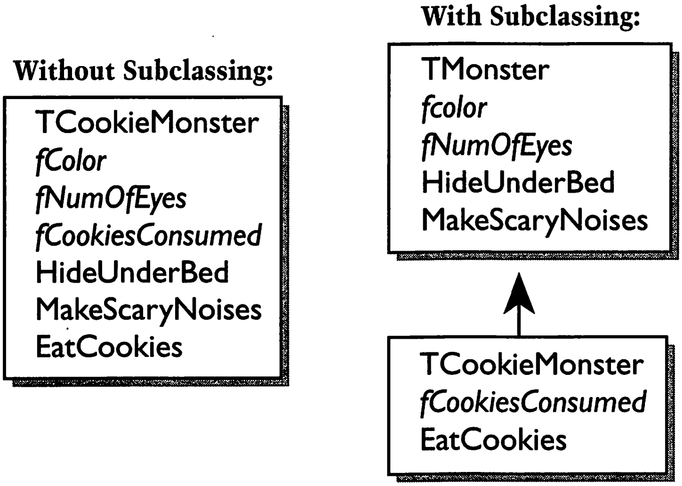


Figure 2.5 *Subclassing example.*

Abstract Class

An abstract class acts as a template for other classes but is one that will never be instantiated. It is usually used as the root of a class hierarchy; that is, its purpose is to promote reuse. You use it if you want to create a really general class like *automobile* or *window* with the intention of creating subclasses. A concrete class, on the other hand, is instantiated to create objects. Examples of concrete classes might be *ForeignAutomobile* and *DomesticAutomobile.*

Overriding

Overriding occurs when a method *replaces* an inherited method from a superclass. In overriding, one message sent to two different but related objects will invoke two different methods. You send the same message to different types of objects, but the resulting behavior is different for objects of different classes. (This override capability also allows you to write generic code and promote code reuse.) A method that can be overridden in C++ is called a *virtual function*.

Figure 2.6 shows an example of the override capability.

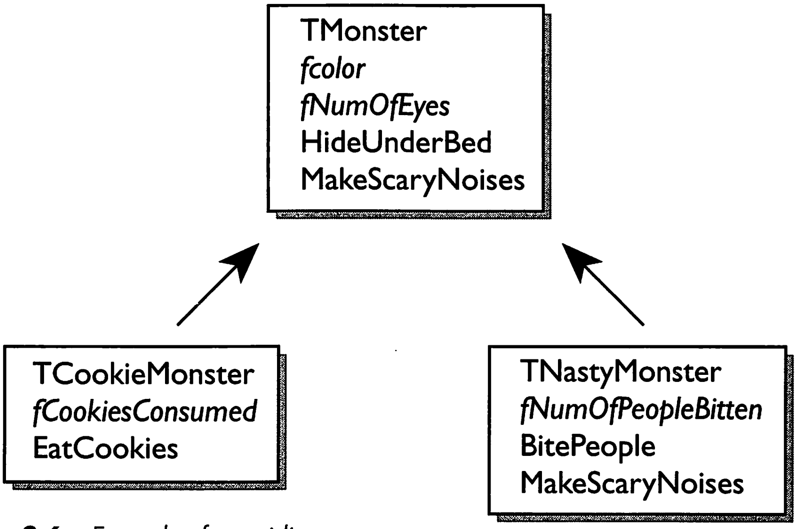


Figure 2.6 *Example of overriding.*

Multiple Inheritance

Multiple inheritance allows you to define classes that inherit properties from more than one superclass, but it greatly complicates designing reusable classes. Superclasses with variables or methods with the same names require complicated rules to avoid conflicts. Some programmers feel that you can write effective code without using multiple inheritance, and that you should, in fact, avoid multiple inheritance because it drastically complicates the design. Figure 2.7 illustrates an example of multiple inheritance.

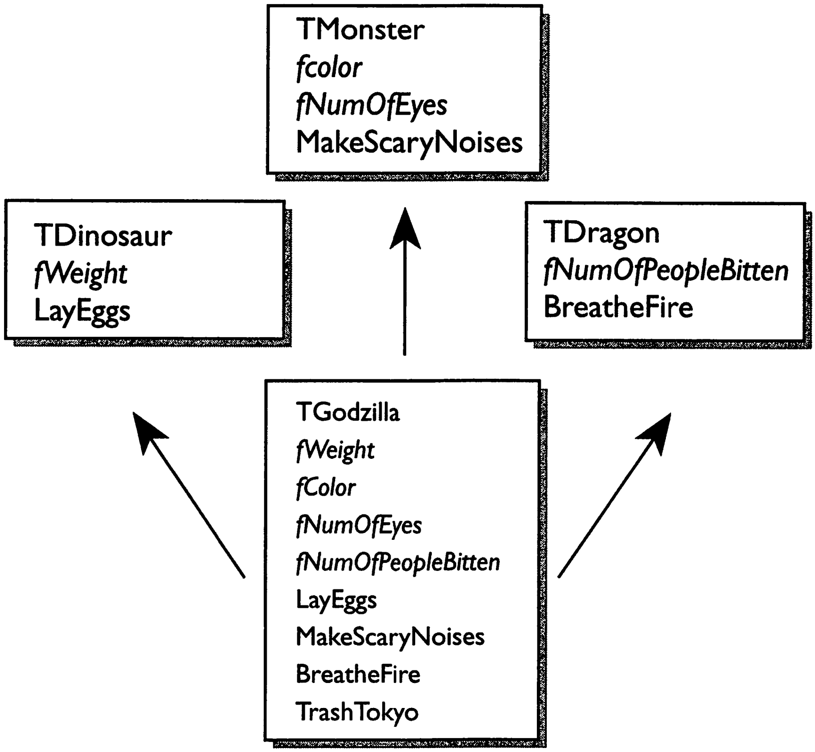


Figure 2.7 *Example of multiple inheritance.*

Polymorphism and Dynamic Binding

Objects from related classes use the same names for different methods. Object languages in general support sending messages to objects without worrying about which exact method will be used. This is called *polymorphism.* Figure 2.8 shows an example of polymorphism.

The message sent to an object can be invoked without knowing the object's actual class. Since messages invoke methods, and some of the methods cannot be resolved at compile time (i.e., we don't know which method will actually be used), they are resolved at run time. Because this operation is done at run time, it is called *dynamic binding.* The method that is chosen depends, naturally, on whether the method is an overriding one.

Dynamic binding requires a process known as *method lookup.* The property of having many routines with the same name (polymorphism) means that the compiler cannot always determine at compile time which method should be called. This results in the need for a method lookup table mechanism to find the correct method. A good object language creates this table by means of compiler-generated code, and this table is transparent to you. In C++, these tables are called *vtables*, with the letter *v* standing for virtual.

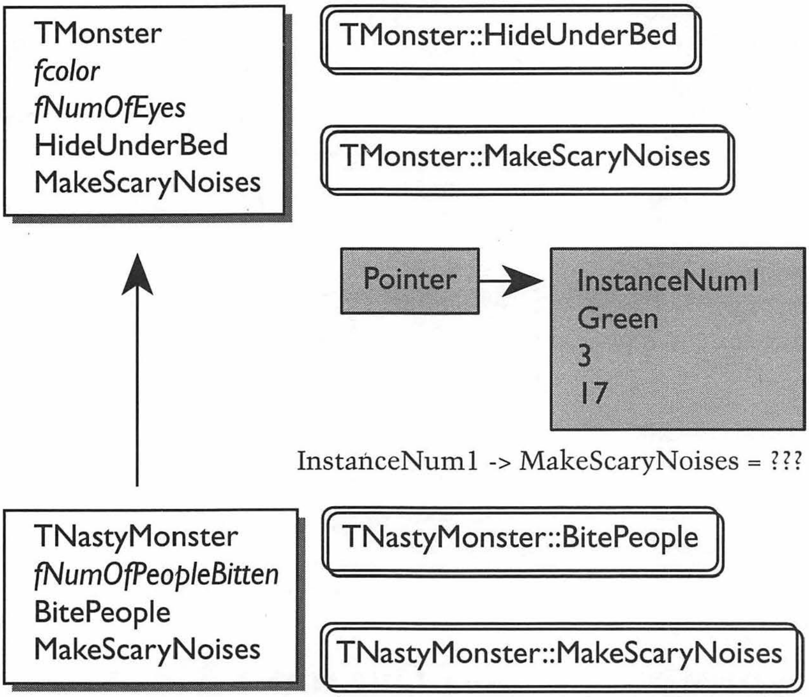


Figure 2.8 *Example of polymorphism.*

Iterator Methods

An iterator method can be set up to send the same message to every object in a collection. The difference between iterator methods and traditional methods is subtle but important. For example, if you want to add a new monster in object programming, you do not need to change the code. However, in traditional programming, you have to add a new case label to the switch statement to accomplish the same thing. The following code illustrates this:

Sending a Message to Each Object:

```
cookieMonster->HideUnderBed( );
oscarTheGrouch->HideUnderBed( );
godzilla->HideUnderBed( );
```

Writing a Switch Statement, Traditional Programming:

```
switch (theMonster)
{
  case cookieMonster:
    HideCookieMonsterUnderBed( );
    break;
  case oscarTheGrouch:
    HideOscarTheGrouchUnderBed( );
    break;
  case Godzilla:
    HideGodzillaUnderBed();  // Watch out!!
    break;
}
```

Using an Iterator Method:

```
ForEveryMonster(HideUnderBed);
```

Looking at the above examples of code, you can see that with a switch statement, you must make physical changes in the code; that is, you must create new names and methods for each case. With the iterator method, you can send messages to all of the objects with just one command. Of course, you have to write the iterator method, but no modification of code is necessary.

Where to Begin an Object-Oriented Program

After reading about all the wonderful things that OOP incorporates and can do, most programmers feel:

- Scared.
- Confused.
- Lost—not knowing where to begin.

Take heart. There is a comfortable, organized way to begin your OO program:

- Describe the problem your program must solve in English sentences.
- Identify nouns. These can be the class names.
- Find verbs. These are good candidates for messages.
- Look for adjectives. These may lead you toward instance variables.

After collecting all the ingredients, practice some visualization:

- Imagine a scene that involves a task.
- Imagine objects in the scene (classes).
- Imagine what the object can do (methods).
- Imagine things that describe the objects (instance variables).

As you begin to design your program, think in terms of small methods; they are easier to write and debug. Make sure that your methods have only a single purpose; this helps keep them small and makes it easier to reuse a class.

To create an object-oriented application, you must carefully design classes for your objects and code your design in an object language. Your coding will be influenced by the language you choose, but whatever the language, your code will be organized around a class hierarchy, and the flow of control will be based on sending messages to instantiated objects. Both your design and your code will look very different from those of a traditional application.

Summary

In this chapter, you've learned about:

- Objects
  Real-world modeling
  Modularity
  Information hiding

- Classes
  Code sharing and reuse
  Subclassing and inheritance
  Polymorphism and dynamic binding

Now you can put this knowledge to work with the following exercises.

Exercises—Programming with Objects

1. Imagine that you are writing an application to help manage a company's personnel and fixed assets. Building on the examples presented in this chapter, create a design (i.e., draw class diagrams) for an object-oriented program that meets the following requirements.

   The company has four types of employees: managers, programmers, secretaries, and bookkeepers. The first two categories are salaried and the latter two are paid hourly. Your design must allow the program to print paychecks for each employee that show the employee's name, gross pay, tax, and net pay. Inputs to the paycheck process for each employee include the employee's salary, tax rate, and (for hourly employees only) the number of hours worked. Give some object (e.g., one associated with a bookkeeper) the ability to print all paychecks. How will the bookkeeper object cause all of the individual paychecks to be printed? You may assume a fixed maximum number of employees if you wish.

   The company also has three types of assets: typewriters, adding machines, and computers. Make, model number, and serial number are of interest for all of these items. For computers, the amount of memory is also important. For simplicity, assume that each employee can be assigned at most one piece of equipment. Your design must allow the program to print a list that shows each employee's name followed by a description of the piece of equipment assigned to that employee.

Finally, each secretary has one boss and each programmer knows one programming language. Your design must allow the program to list the boss for each secretary and the language for each programmer.

2. Create a diagram of the instantiated objects in your program, assuming a company that consists of two programmers, one secretary, one bookkeeper, and one manager, and some reasonable distribution of equipment. You need not show the fields or methods for each instance, but do use arrows to show collaborations among instances; that is, the references from one instantiated object to another. You may use pencil and paper or a paint or draw program.
3. Your class diagram probably demonstrates many instances of inheritance. What possibilities for polymorphism does it contain?

3 The Symantec C++ Environment

If you have not used Symantec's THINK C development environment before, then you are in for a nice surprise—especially if you've been using UNIX or DOS. Symantec C++ is even friendlier than the earlier THINK C versions and has considerably more to offer. It supports MacApp and includes THINK C and the THINK Class Libraries. In the future, it will support the Apple/Symantec jointly developed class library.

If you are a first-time Macintosh programmer, you'll find that Symantec C++ is a fully integrated development environment that contains everything you need to begin developing your own applications, as well as desk accessories and device drivers. The three main components for development—editor, compiler, and linker—are all included in the package and work together to produce your project; that is, you don't have to jump from one application to another to edit, compile, and link. In addition, Symantec C++ has a source-level debugger to aid you in debugging your program.

The following sections introduce you to the Symantec C++ environment, show you how to create an application, and give you brief explanations of each menu, window, and dialog box that you will be using. Chapter 15 treats the components of Symantec C++ in much more detail. In this chapter, we highlight the nitty-gritty aspects of developing applications in this environment, with an emphasis on what to do when.

Getting Started with Symantec C++

If you followed directions in the Users Manual for installing Symantec C++ on your hard drive, you should be ready to write a simple application. The first thing you need to do is create a folder for your project, name it *MyNewProject,* and put it in your *Projects* folder on the hard drive. (You will have separate folders for all of your development projects.)

Symantec C++ is more lenient than earlier versions of THINK C, where you were required to keep everything—THINK C shell, projects, utilities, and library sources—within the *Development* folder. With Symantec C++, you can have the THINK Project Manager in one folder and the rest of the development environment in another folder. In fact, you can have the THINK Project Manager on a hard disk other than that where you keep your project files and it will still execute. Nevertheless, Symantec C++—probably more for logical organization reasons than anything else—has put everything into a folder called *Development.* In a folder called *Symantec C++ for Macintosh* (within the *Development* folder) are the THINK Project Manager, Debugger, libraries, tools, and translators. For ease of access, if for no other reason, you will probably want to keep each of your project folders within the *Projects* folder.

Another useful convention is to name each of your project folders with a *.f* extension. This way you can identify your Symantec C++ project folders at a glance.

Creating a Project

Now that you have a folder ready for your project, double-click on the THINK Project Manager icon to launch Symantec C++. A dialog box similar to that in Figure 3.1 will appear.

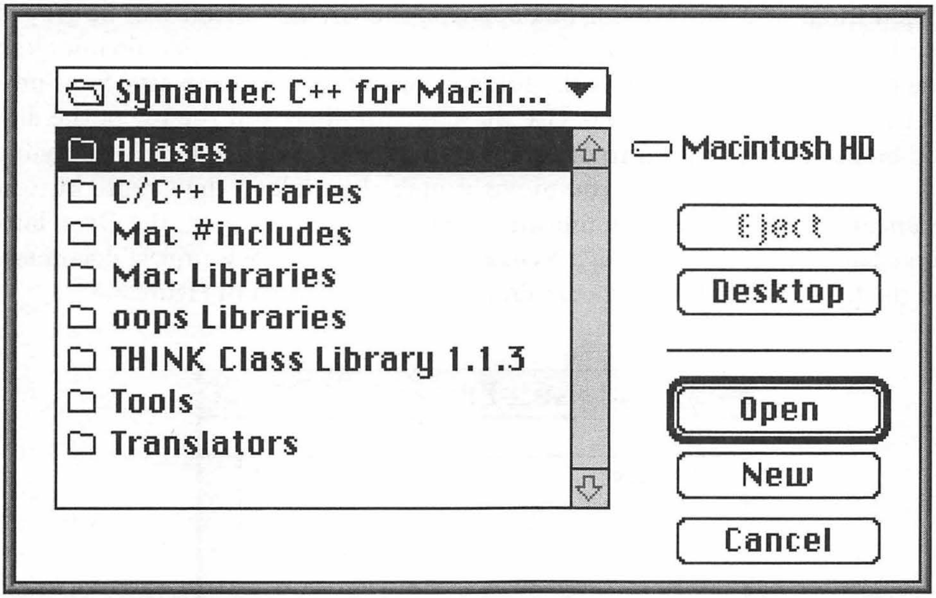


Figure 3.1. *THINK Project Manager dialog box.*

Click on the *New* button, and a dialog box like the one in Figure 3.2 will appear with the request for you to name your new project.

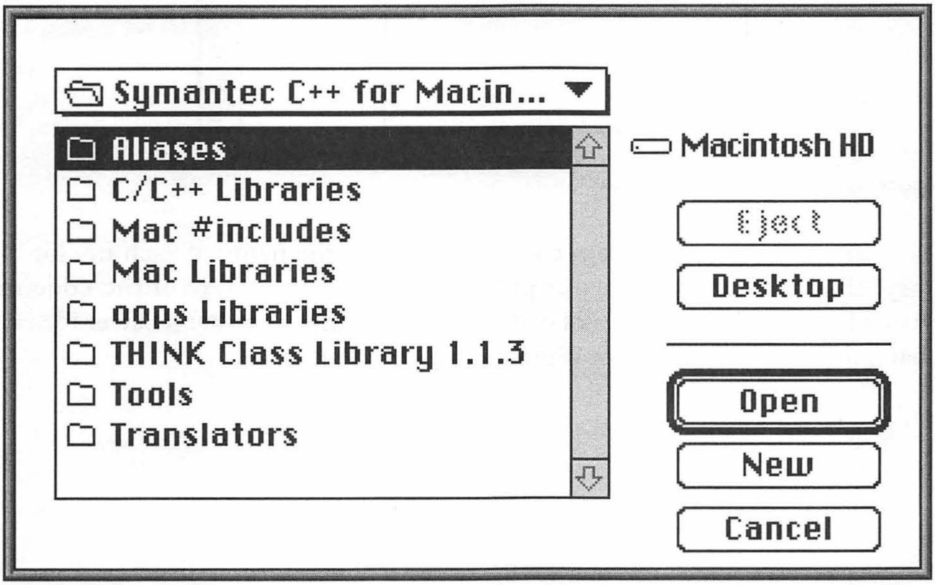
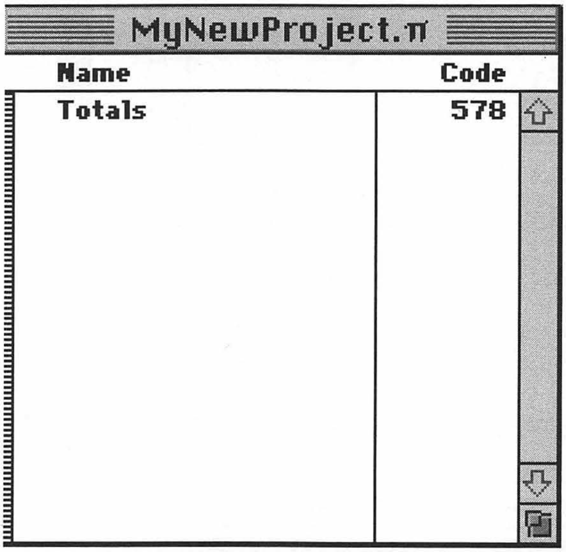


Figure 3.2. *The New Project dialog box.*

When the new project dialog box appears, the *Create* button will be grayed out until you enter text. Name your project *MyNewProject.π* but do not click the *Create* button yet unless the folder in which you want to save your project is the one currently open. Use the standard file box at the top of the dialog box to move around to different folders until you find and open your *MyNewProject* folder. As you move around the folders, the *Create* button changes to *Open*. When you find and open the folder you want, the *Open* button changes to *Save*. Press *Save*. Symantec C++ creates a new project document on the hard disk and displays the Project Window shown in Figure 3.3.

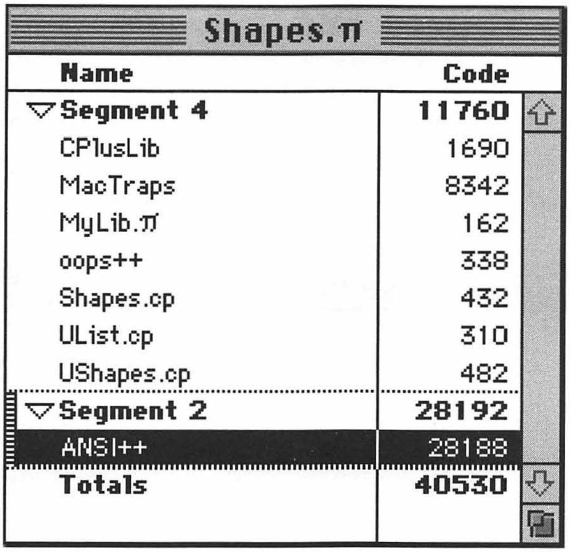


MyNewProject.π

Name	Code
Totals	578

Figure 3.3. *The Project Window.*

The two columns in the Project Window display the name of each file (or library) that you include in your project along with the size of the code in bytes, plus a total of all bytes of code. Look at Figure 3.4 to get a better idea of what a more complex Project Window might look like.



Shapes.π

Name	Code
▽ **Segment 4**	**11760**
CPlusLib	1690
MacTraps	8342
MyLib.π	162
oops++	338
Shapes.cp	432
UList.cp	310
UShapes.cp	482
▽ **Segment 2**	**28192**
ANSI++	28188
Totals	**40530**

Figure 3.4. *Project Window Showing Compiled Files with Number of Bytes*

The Project Window shown in Figure 3.4 contains the files and libraries that are included in the Shape example in Chapter 13.

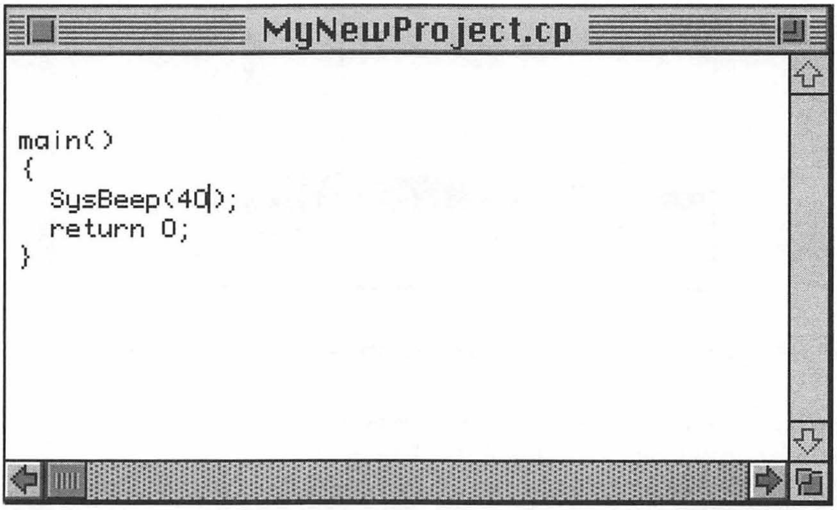
Segments

Symantec C++ now numbers the segments in which your files appear, and, when you have exceeded the segment limit of 32K bytes, automatically opens up a new segment. Since it makes sense to keep related routines in the same segment, if possible, Symantec C++ allows you to move items around from segment to segment.

Segments are units of object code that go in and out of memory. All of your code executes in a resource, which is restricted to 32K, and that resource may or may not be in memory. A code resource with an ID of 0 or 1 is always in memory. A code resource with an ID of 2 or more is the actual code that you compile. The code resources 0 and 1 figure out, when you run the program, which of the segments the code is in. That segment is loaded into memory for as long as you need it. The segment loader is part of the operating system of the Toolbox. (The Toolbox is divided into two parts: the operating system, which has the Memory Manager, and the user interface, which has the Menu Manager and other related managers.)

Creating a Source File

Now that you have created a project, you need to write the source code for it. To do this, pull down the File menu and select *New*. When the source code editing window appears (as shown in Fig. 3.5), type in the code as you see it in Figure 3.5.



```
main()
{
  SysBeep(40);
  return 0;
}
```

Figure 3.5. *Source Code Window for MyNewProject*

After you have typed in the source code, choose *Save As* from the File menu, name the file *MyNewProject.cp* (for C++) and press *Save*. Next, switch to the Project Window.

With the Project Window open, select *Add Files* from the Source menu. A dialog box like the one in Figure 3.6 will appear.

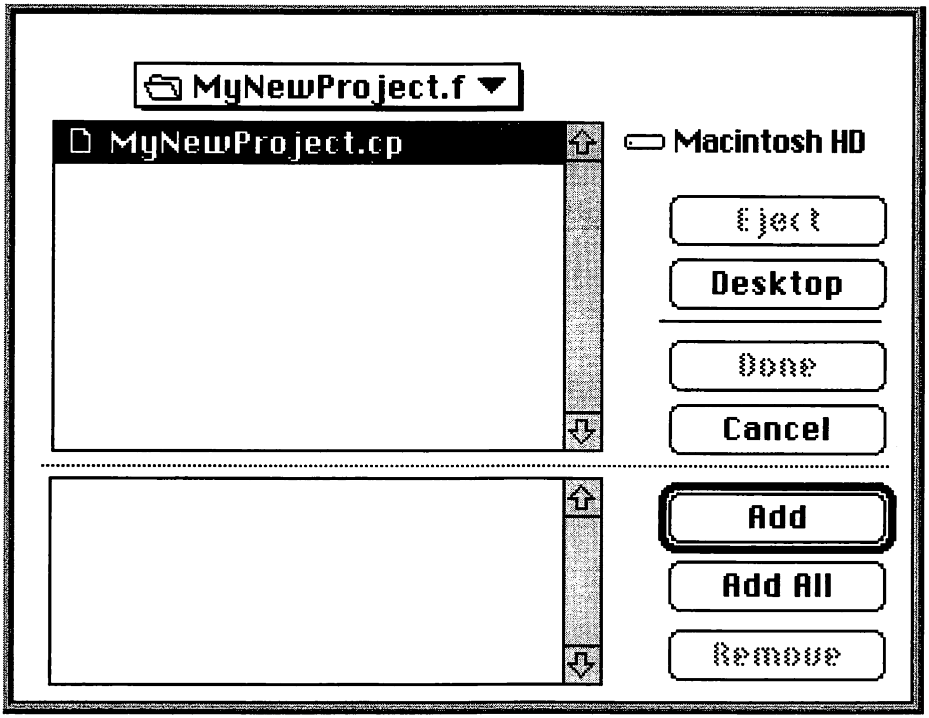


***Figure 3.6**. Add Files Dialog Box from the Source Menu*

Click on the *Add* button to add the file to the *MyNewProject.f* project window. You could run this project now because it does not require any special libraries, but it is a good idea to know from the beginning how to add libraries to your project.

Adding Libraries

Macintosh libraries are essential to most Mac programs. In fact, it is nearly impossible to write an application for the Mac without the Mac header files and libraries. The program disks that you received with Symantec C++ contain almost all of the library functions that you'll need. (But you may want to build your own libraries that contain routines and functions you'll use over and over again.)

Since you already have the *Add Files* dialog box open, move around the standard file box at the top of the dialog box until you see C/C++ Libraries within the *Symantec C++ for Macintosh* folder. Open the C/C++ folder, select ANSI (or *ANSI++*) and click the *Add* button. You will see the ANSI library appear in the lower window of the dialog box, as shown in Figure 3.7.

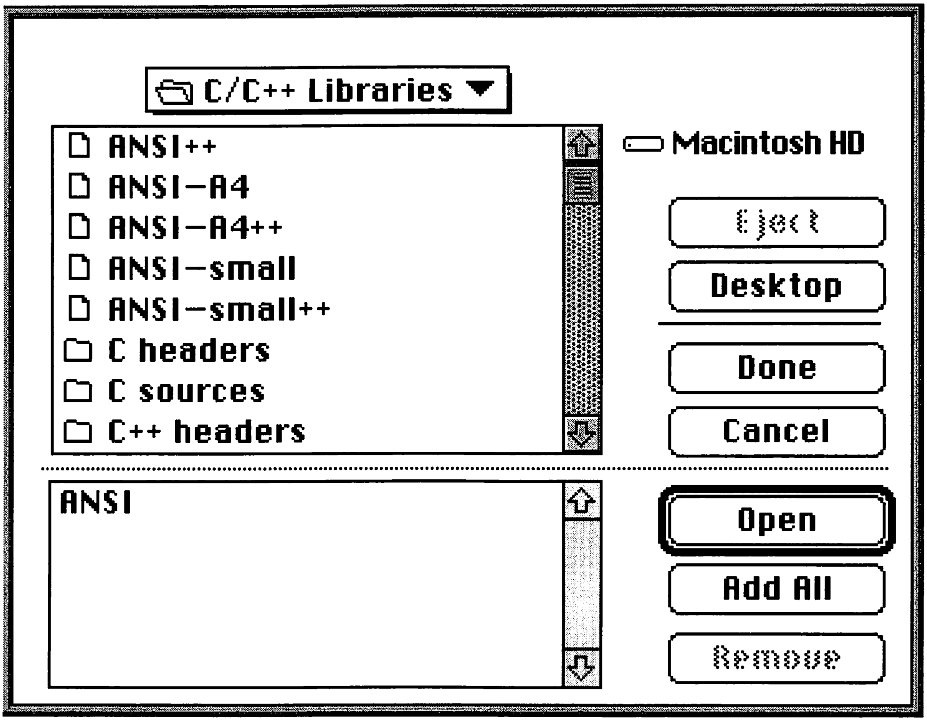


Figure 3.7. *Adding Libraries to MyNewProject*

You do not actually need the *ANSI* library to run this project, but you will need that and other libraries for most of your applications. Adding to this project will not affect the project.

Compiling the Program

There are a number of ways to compile your program, but the two that you will use most are the *Compile* command in the Source menu and the *Run* command in the Project menu. For the purposes of this project, choose *Compile* from the Source menu. A dialog box shows the number of files and the number of lines of code that are compiled (in this case, 1 file and 5 lines, if you started your code on the top line).

Remember that Symantec C++, unlike traditional compilers, does not create separate object files from your source files. Instead, it puts all the object code into the project document.

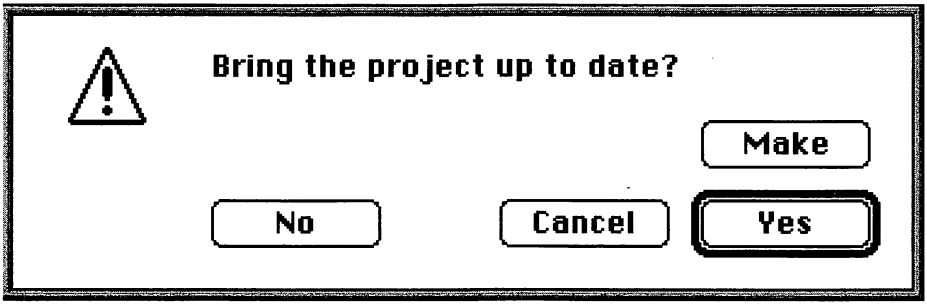
Running the Program

Now it's time to see if the program will run. If it compiled, you have half the battle won. The other half is linking all parts of the program.

The good news is that this program compiles and links. The bad news is that it is kind of a dumb program; all it does is play your system beep sound. You can, however, make it more exciting by changing your beep sound. If you have one of the newer Macs with a built-in microphone, or if you have sound-editing capability or a sound management program (like SoundMaster), you can assign something that you really like to be your system beep. Then, when you run this simple little program, you may even hear a Bach fugue. Just remember to change the number inside the parentheses after *SysBeep* in your source code. The number 40 only allows a 1-second sound to play. If the sound you want to play lasts about 6 or 7 seconds, you may want to write in the number 400 to make sure the entire sound plays.

Building an Application

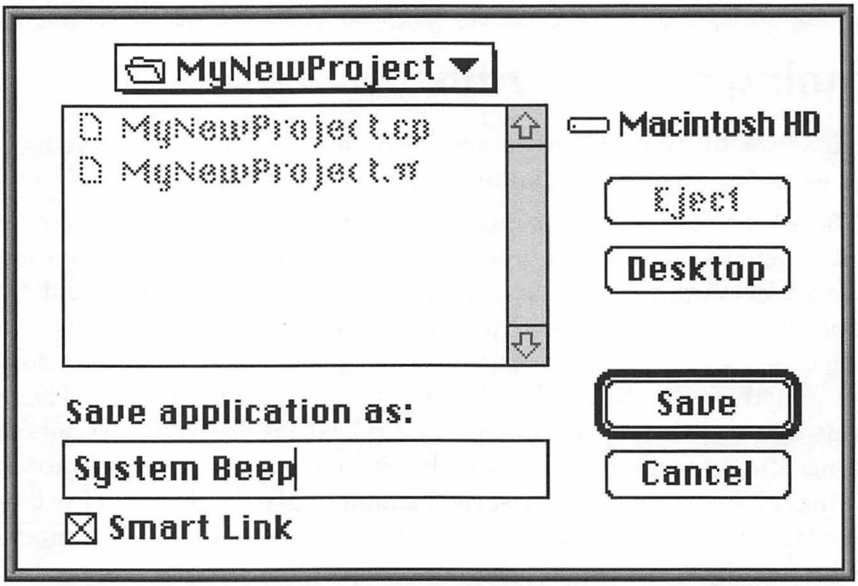
If you have done everything right to this point, you can now turn your project into a stand-alone application. Choose *Build Application* from the Project menu. If you have made changes in the project, the dialog box shown in Figure 3.8 will appear.



Bring the project up to date?

Figure 3.8. *Bringing MyNewProject Up to Date*

Bring the project up to date by clicking on the *Yes* button. As soon as the program is recompiled, the dialog box in Figure 3.9 will appear.

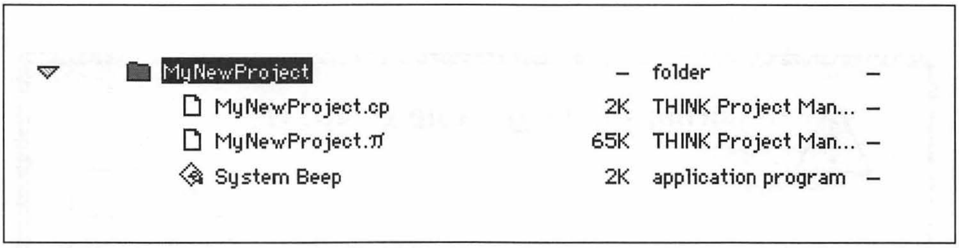


Save application as: System Beep

☒ Smart Link

Figure 3.9. *Saving MyNewProject as an Application*

To be really creative, save your application as *System Beep*. Hang around for a while and then open up the *MyNewProject* folder on your hard disk. You will see the application there just as it appears in Figure 3.10.



```
▽   MyNewProject                  –    folder                   –
        MyNewProject.cp          2K    THINK Project Man...     –
        MyNewProject.π          65K    THINK Project Man...     –
        System Beep              2K    application program      –
```

Figure 3.10. *System Beep, the Application Developed from MyNewProject*

Now, double click on the application icon, and voila! It may not be a spreadsheet, database management, or word processing program, but it is a program nevertheless.

Summary

In this chapter, you have learned how to:

- Create a project.
- Create a source file.
- Add files and libraries.
- Compile the program.
- Run the program.
- Build an application.

In Chapter 15, Using Symantec C++, we discuss the other features of the Symantec development environment in much greater detail. Now, however, you know the basics of the THINK Project Manager. In the next chapter, you will learn the fundamentals of the C++ language.

Exercises

1) Install Symantec C++ on your computer if you have not done so already.
2) Perform the tutorial exercises found in the Symantec C++ manual.

4 Fundamentals of C++

The C++ language was developed by Bjarne Stroustrup of Bell Laboratories and Apple's version has been available on the Macintosh since 1990. C++ (which is contrived to mean "C incremented by 1," or "1 better than C") is essentially a superset of ANSI C with many additional features. It supports object-oriented programming via class definitions, inheritance, polymorphism, and more.

To understand C++, you have to be familiar with the programming basics of the C language and how it relates to the Mac. This section introduces the elements of the language, describes the syntax, and identifies some differences between C and C++, to help you avoid the traps and pitfalls of C.

Comments

Comments represent an area where C and C++ differ in a significant way. C uses /*...*/ to define a comment. This expression is referred to as a multiline comment. For instance:

```
/*
This is a comment
*/
```

The embedded /* is ignored. Another example follows:

```
/* This /* is a comment */.
```

In the above example, the compiler will ignore the second /*. Another example is:

```
/* This /* is a */ comment*/
```

Multiline comment pairs do not nest. That is, one comment pair cannot occur within a second pair.

C++ uses both the star slash and the double slash—//—which is a single-line comment. Everything on the rest of the line is ignored. For example:

```
a = b;    //  Set a equal to b.
c = d;
```

The compiler ignores the words, *Set a equal to b.*

For debugging purposes, the // makes it very easy to comment out code. However, it is very poor style to leave commented out code in software. For instance:

```
    for (i = 1; i < 10; i++)
    {
        a = b;
//      c = d;
        e = f;
       g = h;
    }
```

In the above example, the line *// c = d* has been commented out; that is, the compiler has been instructed to ignore the expression *c = d*. This might be done for a variety of reasons, but most likely the programmer will comment out the line to see if the rest of the code works. The problem occurs when the programmer, in proofing the code, does not see the all-too-easily-hidden //.

Writing comments liberally will help you identify specific areas of your program more easily. It is always a good idea and good style to write a comment or comments for each function within the program. The double slashes are less prone to error than the /*...*/. However, the latter are useful for block statements.

Statements

All C++ programs derive from statements, which contain expressions. Statements are really lines of instructions that you give to the compiler. If it isn't a blank line, lines, or a comment, then it's a statement. A statement in C ++ always ends in a semicolon or a curly brace, but can span several lines. For example, you can say:

```
a = b;
```

or:

```
a

=

b

;
```

or:

```
a = b;

c = d;
```

or:

```
a = b; c = d;
```

As shown in the last example, it is possible to have several statements on the same line, separated by semicolons.

Note:

The last example contains a redundant statement; that is, a statement that does nothing. This is called a no-op.

Another more complicated example might be:

```
for (i = 1; i < 10; i++)
{
  a[i] = b;
}
```

or:

```
for (i = 1; i < 10; i++){ a[i] = b; }
```

or:

```
for (i = 1; i < 10; i++){ a[i] = b; };
```

White Space

White space describes blanks, tabs, carriage returns, and line feeds embedded in text. C++ ignores white space except inside identifiers, numbers, and any words or symbols that belong together.

Variables

Variables are names given to memory locations. They contain values that change with program dynamics and are written to or read from memory as required. Variables are identifiers that can have a maximum of 127 characters in length, and all of the characters are "significant." At one time, compilers such as Fortran allowed variables up to 30 characters, but these compilers only read the first seven.

Numbers, alpha characters, and underlines are valid symbols for variable names. It is not a good practice to begin a variable name with a number or an underline. Since the characters are case significant, always begin the first word of a variable's name with a lowercase letter, and then capitalize the first letter of each significant word that follows. (This convention helps you distinguish a variable from a function.)

Examples of the format of variables include:

```
myEventLoop

theControlRect

dialogPtr
```

Reserved Variables

There are some words that you may not use for variable or function names because these are reserved by C++. The full list of these 54 keywords follows:

asm
auto
break
case
catch
cdecl
char
class
const
continue
default
delete
do
double
else
entry
enum
extern
far
float
for
fortran
friend
goto
huge
if
inline
int
long
near
new
overload
operator
pascal
private
protected
public
register
return
short
signed
sizeof
static
struct
switch
this
template
typedef
union
unsigned
virtual
void
volatile
while

Predefined Variable Types

C++ has certain basic or predefined variable types. Some of the most important of these are:

char A byte that ranges from –128 to +127. An 8-bit number. The first bit is the sign, and the following bits are the number.

short A word that is 2 bytes in length. It ranges from –32,768 to +32,767.

int In Symantec C++, the same as a short. It can be set to either a short or a long.

long A long word; i.e., 4 bytes. It ranges from –2,147,483,648 to +2,147,483,647.

float Real 1.e that ranges from –36 to +36 (precision 7 digits).

double Real 1.e that ranges from –303 to 1.e +303 (precision 13 digits).

Figure 4.1 shows some predefined variable names and their positions in memory.

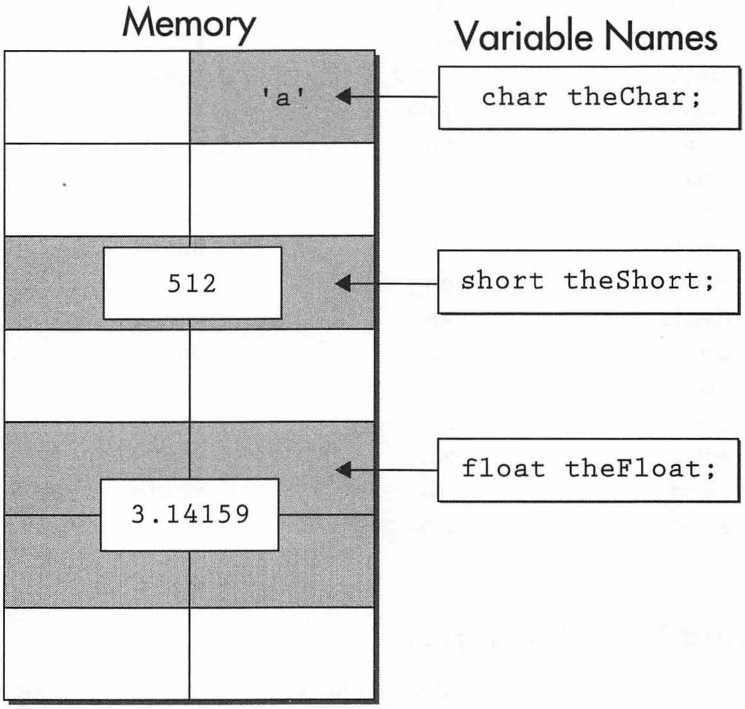


Figure 4.1 *Example of predefined variable names and their positions in memory.*

Another way to demonstrate the size of integers is in the following list:

Type	Size
char	1 byte
short	2 bytes
int	4 bytes (MPW)
	2 bytes or 4 bytes, depending on the option (THINK)
long	4 bytes

Typically, an *int* is of machine word size. However, on the 68000, this is ambiguous (32-/16-bit microprocessor). The Macintosh Programmers Workshop (MPW) has a 32-bit *int*, while THINK has a 16-bit *int*. The best way to avoid confusion is to use the words *short* or *long* and never use *int*.

The predefined types mentioned above are primitive data types. You can make up your own data types. Typical examples of variable types are:

```
main()
{
   short a;
   long b;
   float c;
   char d;

   a = 10;
   b = 7638465;
   c = 3.14;
   d = 'a';
}
```

Note:

Under THINK C 5.0, and Symantec C++, you can have an option set up to declare whether an *int* is a *short* or a *long*. Under 4.0, a dialog box asks you to declare whether an *int* is a 4-byte or a 2-byte *int*.

Declarations

C++ requires that you declare the type of every variable before you use it. You declare a variable by specifying the type first and then stating the name of the variable of that type that you want to declare. For instance, if you want to declare *a*, you must say:

```
short a;
```

Definitions

Definitions cause space to be allocated for a variable, or code to be generated for a function. (Declarations, on the other hand, do not cause space allocation or code generation.) Variable definitions should appear at the beginning of each program or function. Typical definitions are:

```
short i = 0, j = 0;

float pi = 3.14;

char cr = 0x0D;

char space = ' ';
```

Initialization

You can initialize the variable in the definition statement, and you can define a variable anywhere a normal statement may occur. Just remember that the variable must be defined before you use it. An example showing variable definitions directly after a declaration follows:

```
x = 5;

y = 6;


float t; t = x; x = y; y = t; //swap x and y.
```

A variable that has not been assigned a value is said to be uninitialized. An uninitialized variable's value is also undefined. The memory storage for the variable is not swept clean when allocated. C++ supports two forms of variable initialization, as shown here:

```
short myValue = 1024;  //explicit form
```

or:

```
short myValue (1024);  //implicit form
```

A variable may also be initialized with an arbitrarily complex expression. Remember that all variables used in the expression must be initialized. An example of a complex expression might be:

```
float price = 99.95;
float tax = 0.0825;
float total (price + (price * tax));
```

Signed and Unsigned

To further differentiate the variable types, you can put the modifiers *signed* or *unsigned* before certain types of variables. For example:

```
main()
{
  short a;
  unsigned short b;
  signed char c;
  unsigned char d;

  a = 32767;
  b = 65535;
  c = 'a';
  d = 'Σ';
}
```

The default for any of these variables is *signed*. The following list shows the ranges for *signed* and *unsigned* variables:

unsigned char	The range is 0 to 255 or 2^8 -1(still 1 byte). All 8 bits are used for the number.
unsigned short	The range is 0 to 65,535 (2^{16} -1).
unsigned int	Same as short or long, depending on the option.
unsigned long	This range is 0 to 2^{32} -1.

You can also have numerical constants. For instance:

```
long a;
a = 5;
```

Constants are *shorts* by default. However, you can change this in Symantec C++. If, in the example above, you change the *int* to a *long*, which is called a *literal constant*, the 5 will be a *long*. (The problem with the above example is that it has mixed types. First, you declare *a* as *long*, then set it equal to 5, which is normally a *short*. This is not good coding practice.)

Specified Constants

Following is a list of specified constants showing the individual formats:

OxFF	hex char
OxFFFF	hex short
OxFFFFFFFF	hex long
123L	long
128U	unsigned (You may use U or u)
1024UL	unsigned long (UL or LU)
"Macintosh"	string
'T'	char
'TEXT'	long
1.23	double float (double precision)
1.23e2	double float (scientific)
3.14F	float (single precision)
1.0L	extended precision

All numbers are assumed to be short and decimal unless otherwise specified. When you declare the variables, you may write:

```
    short a, b;
```

(You may have as many of these as you wish, each separated by a comma.)

```
    short a = 10;
```

or:

```
    short a = 10, b;
```

(a and b are both declared shorts and, in addition, a is set to the value of 10.)

In the statement *short a = 10, short* is the type, *a* the identifier, and *10* the initializer.

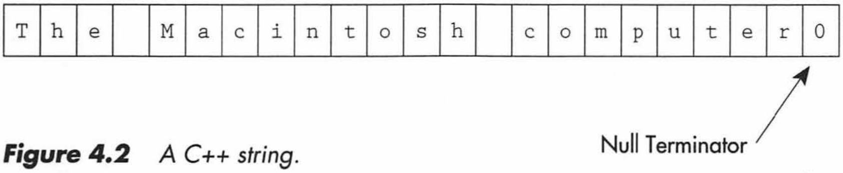
In the above example, we initialized *a* to be the value of *10*, thereby making *10* the *initializer*.

Logical Values

Any value that does not equal zero is assumed to be *true*. If a value is zero, then it is logically *false*.

Strings

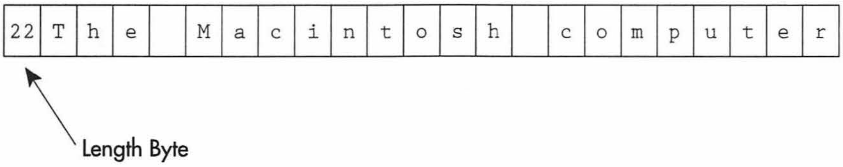
C++ strings are always terminated with a null (0) character (see Figure 4.2), and character constants—*a, b, c, d,* and *e*—are automatically null terminated. C++ strings are usually contained in an array of *chars*.



T	h	e		M	a	c	i	n	t	o	s	h		c	o	m	p	u	t	e	r	0

Figure 4.2 *A C++ string.*

Pascal strings are used with the Mac ToolBox (see Figure 4.3). The first byte in a Pascal string is the length of the string, with a maximum of 255 characters (Str255). Pascal strings are usually contained in an array of unsigned *chars*.



22	T	h	e		M	a	c	i	n	t	o	s	h		c	o	m	p	u	t	e	r

Figure 4.3 *A Pascal String*

String Constants

The C++ compiler places a null character at the end of every literal string constant. An example of a string constant might be:

```
char *myString = "abcde";
```

Here, *myString* points to six characters.

Scope

Statements can refer to variables that are only within the same scope. Scope is the space, domain, or world where an identifier is recognized. Figure 4.4 illustrates the areas of scope.

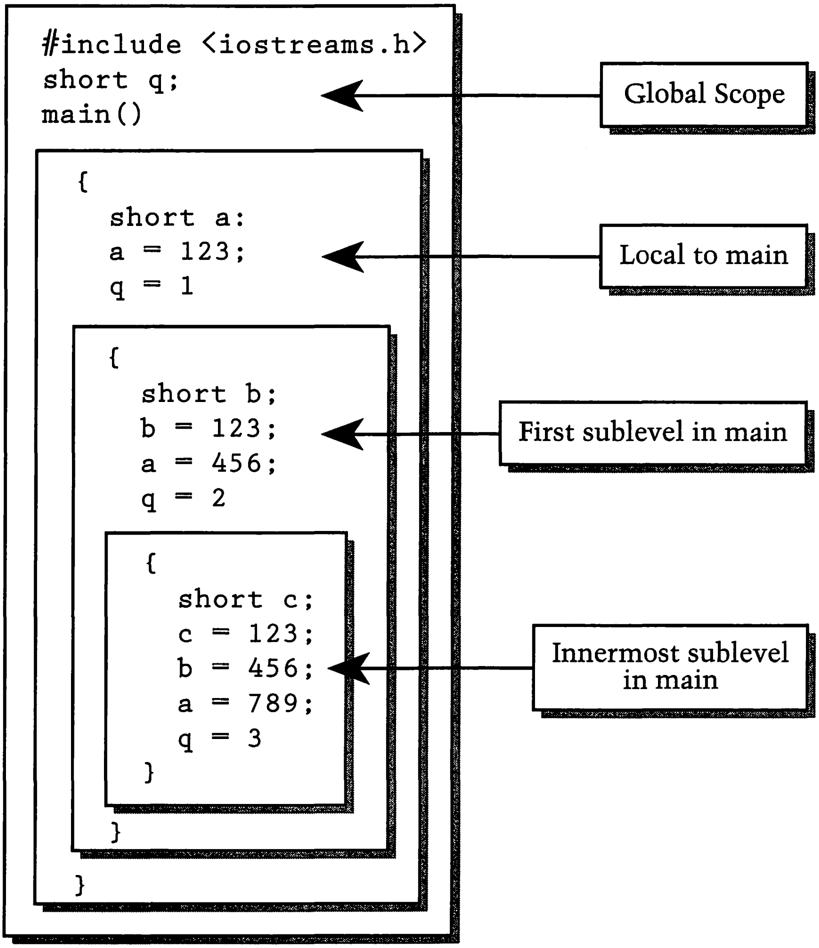
Operators

An operator is a unique character or set of characters that represent a specific computer operation. An operator works on something called an *operand*.

In C++, there are four basic types of operators:

1. Unary
   - Prefixed
   - Postfixed

2. Binary
   - Arithmetic and logical
   - Assignment
   - Comparison
3. Ternary
4. Comma



```
#include <iostreams.h>
short q;
main()
{
  short a:
  a = 123;
  q = 1
  {
    short b;
    b = 123;
    a = 456;
    q = 2
    {
      short c;
      c = 123;
      b = 456;
      a = 789;
      q = 3
    }
  }
}
```

Figure 4.4 *Scope.*

Operators have a hierarchical order something like the order of operations in mathematics. Table 4.1 shows the order or precedence for groups of operators in descending order:

Table 4.1

Operator	Description
::	Scope resolution
( )	Function call
[ ]	Array element
.	Direct selection
– >	Indirect selection
+	Unary plus
–	Unary minus
++	Increment
– –	Decrement
!	Logical **NOT**
~	Ones complement
*	Dereference
&	Address of
sizeof	Object size
(type)	Cast
new	New operator
delete	Delete operator
*	Multiplication
/	Division
%	Modulus
+	Addition
–	Subtraction
>>	Left shift
>>	Right shift

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Logical equals
!=	Logical **NOT** equals
&	Bitwise **AND**
|	Bitwise **OR**
&&	Logical **AND**
| |	Logical **OR**
?:	Conditional
=	Assignment
+=	Addition update
– =	Subtraction update
*=	Multiplication update
/=	Division update
%=	Modulus update
<<=	Left shift update
>>=	Right shift update
&=	Bitwise **AND** update
!=	Bitwise **OR** update
^=	Bitwise **XOR** update
,	Comma

Unary Prefixed Operators

Operators require one, two, or three variables. Unary operators perform functions on a single operand. A list of unary prefixed operators follows:

*	Dereferencing
&	Address of
+	Positive
–	Negative
!	Logical **NOT**
++	Increment
– –	Decrement
sizeof( )	Size of
(cast)	Cast to
new	New operator
delete	Delete operator

Dereferencing — *

An address is a reference to a memory location. When you dereference the address, you then have the contents contained at that address. For instance:

```
theValue = *thePointer;
```

In the above example, the statement takes the contents of the address contained in *thePointer* and places it in *theValue*. Actually, the statement sets *theValue* equal to **thePointer*. Another example might say:

```
newValue = *(&oldValue);
```

This is the same as saying:

```
newValue = oldValue;
```

If *x* is a pointer

```
*x = 10;
```

the above example puts *10* in the location pointed to by *x*. In other words, the contents of *x* equal *10*.

Address of — &

The operator & is placed before a variable name to indicate that we want the *address* (in memory) of that variable, not its current value. For instance:

```
theAddress = &myVariable;
```

The above statement sets the variable *theAddress* equal to the memory address of *myVariable.*

Negative — -

The negative operator turns a number into a negative number. For example:

```
short a;
short b;
a = 5;
b = -a;
```

In this example, the value of *b* is equal to -5.

Ones Complement — ~

Ones Complement inverts the bits in the variable. For instance:

```
char a;
char b;
a = 0xAA;
b = ~a;
```

In this case, the value of *b* will be *0x55.*

Logical NOT — !

A logical NOT takes the logical of its operand and inverts it. For example:

```
char a;
char b;
a = 0xAA;
b = !a;
```

In this case the value of *b* will be *0.*

Increment — ++

This operator increases the value of a variable by 1. For example:

```
short a;
a = 5;
++a;
```

The result of this is that *a* = 6. The above example says the same thing as:

```
short a;
a = 5;
a = a + 1;
```

We could also say:

```
short a;
a = 5;
a++;
```

In this latest example, *a* still has the value of 6, but the ++ is now a postfixed operator instead of a prefixed operator. (See section below on unary postfixed operators.)

Prefixed and postfixed operators have different effects in C++. For instance, look at the following code:

```
short a;
short b;
a = 5;
b = ++a;
```

This statement says the same thing as:

```
short a;
short b;
a = 5;
a = a + 1; b = a;
```

In the above example, a is incremented by 1 and put into b; that is, $b = ++a$, which is the same as $a = a + 1; b = a$.

Now take a look at an example with a postfixed operator:

```
short a;
short b;
a = 5;
b = a++;
```

Note:

C++ takes its name from the ++ operator. C++ is one step beyond its predecessor language, C.

The above example says the same thing as:

```
short a;
short b;
a = 5;
b = a; a = a + 1;
```

Here, the example is evaluated as $b = 5$ and $a = 6$ *(b = a++;)*. In other words, b is set equal to a , and then a is incremented.

Decrement — - -

This operator decreases the value of a variable by 1. For instance:

```
short a;
a = 5;
-a;
```

The result of this is that $a = 4$. This is the same as:

```
short a;
a = 5;
a = a - 1;
```

We could also say:

```
short a;
a = 5;
a--;
```

In the preceding example, *a* still has the value of *4*, but the -- is now a postfixed operator instead of a prefixed operator. See the explanation of the difference between prefixed and postfixed operators in the preceding section on the increment operator.

Size of — sizeof( )

The *sizeof* operator returns the size of a variable in bytes. For example:

```
short a;
short b;
b = sizeof(a);
```

Here, *b* is equal to *2* because *a* is a *short*, and a *short* takes up 2 bytes. You can also say:

```
short b;
b = sizeof(long);
```

In this example, *b* is equal to *4*.

Cast to — (cast)

Casting transforms a variable from one type to another. For instance:

```
long a;
short b;
b = 5;
a = b;
```

In this example, *a* may or may not be equal to *5*. You cannot predict this because there is a variable-type mismatch. To be safe, you cast it thus:

```
long a;
short b;
b = 5;
a = (long)b;
```

In this case, *a* = *5*.

New Operator — new

New is a unary operator that's available to access memory storage. *New* is a replacement for the library function *malloc* and is more convenient to use. *New* creates memory in an area called *free store,* which is a system-provided memory pool (located in the heap) for objects whose lifetime is directly managed by the program. When you want to create an object in memory, you call *new.* When you want to destroy that object, you use *delete.*

The *new* operator returns a pointer to the beginning of memory for the allocated variable or object. If you don't have any space in *free store*, the *new* will return a zero-value pointer. You can then use this feature to determine whether or not you have enough memory to allocate something. For example:

```
short *i;

i = new short;

*i = 20;
```

Here, the first line declares *i* to be a pointer to a *short.* The compiler only allocates space for the pointer, not for the contents that the pointer points to.

The second line uses the *new* operator to obtain a *short* object, which is located in the *free store* area. The *new* operator returns the address of the *short,* which we place in *i* with the = operator.

On the third line, we set the contents that are pointed to by *i* to the value of *20.*

We discuss the *new* operator in more detail later in the book.

Delete Operator — delete

The *delete* operator destroys the memory space that was allocated by the *new* operator. For instance, to delete the space we allocated above, write:

```
delete i;
```

The memory allocated by the *new* operator is no longer accessible. However, the value is still there. If you use the *delete* again, you will *delete* another word that is at the same address as *i,* because the address has not been eliminated. *Delete* will also delete memory in the *free store* area without your ever having allocated anything to it. You do *not* want to do this. To prevent it, never use *delete* without first using the *new* operator. However, if you delete a pointer of zero value, the *delete* command actually does nothing.

Unary Postfixed Operators

Following is a table of unary postfixed operators:

( )	Function call
[ ]	Array subscript
.	Direct selection
– >	Indirect selection (called an arrow)
++	Increment
– –	Decrement

Function Call — ( )

A function is like a piece of code or routine that you call up when you need it. You usually declare a group of code a function when you are going to use it many times. It makes things more understandable. In a function call (in between the parentheses), you pass to the function a group of arguments (variables) that are used or set by the function. Sometimes a function returns a value. Below is an example of a call to a function:

```
short a;
short b;
short c;
c = divide(a, b);
.
.
.
}
```

This function definition would be:

```
short divide(short x, short y)
{
  return(x / y);
}
```

A second type of call to a function might look like:

```
short a;
short b;
short c;
divide(a, b, c);
.
.
.
}
```

The function would be:

```
short divide(short x, short y, short & z)
{
 z = x / y;
}
```

For more information on and further examples of functions, see Chapter 6.

Array Subscript — [ ]

An array is a collection of variables that is contiguous in memory. An array has a name, a type, and an index. To use an array, you first have to declare it. For instance:

```
long a[100];
```

The above declares the array as 400 bytes and sets aside the memory for it. Even though this is declared as an array, an array in C++ is not a data type; it's an operator. To use the array in code, you would write:

```
long a[100];
a[0] = 1000;
a[1] = 1001;
a[2] = 1002;
.
```

```
    .
    .
    a[98] = 1098;
    a[99] = 1099;
```

Here, you've created an array that has 100 elements. The [ ] is an operator, just as the + is in addition, but in the function *a[i]* is the same thing as **(a + i)*. In the example, *a* is actually the address of the beginning of the array or of the first element in it. When *i* is added to *a*, the *a + i* combination is the *i*th element in the array. So, **(a + i)* is the contents of the *i*th element in the array. This is exactly the same as saying *a[i]*. Since the address of *a* is the address of the first element in the array, you can obtain the contents of that element by stating that the index is equal to *0*. Therefore, arrays always start with *a[0]* because that is the way you obtain the first element in the array. The last element in the array is *a[99]*, even though we dimensioned the array by 100, which is the maximum index value of the value by which you dimensioned the array. So if we addressed *a[100]*, we would be addressing a memory location of something that is out of the bounds of *a* and would be an invalid value.

Since the [ ] is an operator, you do not have to declare something as an array to use this operator.

Dynamic Arrays, New

If you wanted to create an array that is dynamic (i.e., its size is determined at run time, as opposed to compile time), you could write:

```
    long *a;
    short n;
    n = 100;
    a = new long[n];
    a[99] = 1099;
```

In the above example, the *new* in the statement *a = new long[n]* allocates an array of *n longs* and places the address of the first element of the array in *a*. This makes it dynamic. In contrast, a static array is dimensioned at compile time—once its size is declared, it never changes. For a dynamic array, there is no space allocated at compile time. Using the *new* operator, you request space for that array.

Dynamic Arrays, Delete

To use *delete* to wipe out the memory in the preceding *new* example, you would write the following:

```
delete [n] a;
```

This will delete *n longs* pointed to by *a* from the *free store.*

Direct Selection — .

To understand what direct selection is, you have to understand structure. A structure is a way to consolidate or encapsulate a group of variables. It's similar to an array, but the variables that we put into this group are not all of the same type. For instance:

```
typedef struct
{
  long a;
  char b;
  short c;
} myType;
```

The above code creates a new variable type called *myType.* The variable type that it creates is a type just as a *short* is a type. To use this new type, we need to say:

```
myType theType;
```

The above line declares a variable called *theType* of the type *myType.* To set the values contained in *theType,* we would use the *direct selection* operator. For example:

```
theType.a = 8201836;
theType.b = 'm';
theType.c = 512;
```

When we declared *theType,* we created a collection of variables in memory all associated with the identifier *theType,* which is a structure. It can contain any number and combination of variable types. In the above example, it contains a *long*, a *character*, and a *short*. To set the value of any of the variables in the structure, we used the *direct selection* operator. You not only set the values (as above) but you can use the values as well.

Indirect Selection — - >

Indirect selection uses the -> symbol, which is sometimes called an *arrow*. Indirect selection performs almost exactly the same function as direct selection, except that the variable is an address or a pointer. For instance:

```
myType theType, *pTheType;

pTheType = &theType;

pTheType->a = 8201836;

pTheType->b = 'm';

pTyeType->c = 512;
```

On the first line of this example, we declare a structure called *theType* of type *myType* and a variable called *pTheType*, which is to be used as a pointer to something that contains a record of type *myType*. On the second line, we put the address of the variable *theType* into *pTheType*. On all the other lines, we use indirect selection to gain access to the individual members contained in the structure.

Increment — + +

This operator increases the value of a variable by 1. For example:

```
short a;

a = 5;

a++;
```

The result is that *a* = 6. This says the same thing as:

```
short a;

a = 5;

a = a + 1;
```

Decrement — --

This operator decreases the value of a variable by 1. For example:

```
short a;
a = 5;
a--;
```

The result of this is that *a* = *4*. This is the same as the following:

```
short a;
a = 5;
a = a - 1;
```

Binary Operators—Arithmetic and Logical

Binary operators function on two expressions, one on the left side of the operator and one on the right side. A list of binary operators follows:

Arithmetic and Logical Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise **AND**
|	Bitwise inclusive **OR**
^	Bitwise exclusive **OR (XOR)**
<<	Left shift
>>	Right shift
=	Equal (or replacement)

Addition — +

This is used simply to add two numbers together. For example:

Note:

The + (plus) sign is also a unary operator.

```
short a;
short b;
short c;
a = 4;
b = 5;
c = a + b;
```

The result of the above is that *c = 9*.

Subtraction — -

This operator is used to subtract two numbers. For instance:

Note:

The - (minus) sign is also a unary operator.

```
short a;
short b;
short c;
a = 4;
b = 5;
c = a - b;
```

Here, *c = –1*.

Multiplication — *

This operator is used to multiply two numbers. For example:

Note:

The * (asterisk) sign is also used as the dereferencing unary operator.

```
short a;
short b;
short c;
a = 4;
b = 5;
c = a * b;
```

Here, *c = 20*.

Division — /

This operator is used to divide two numbers. For instance:

```
short a;
short b;
short c;
a = 20;
b = 5;
c = a / b;
```

Here, *c = 4.*

Modulus — %

This operator is used to show the remainder when the first number is divided by the second number. For example:

```
short a;
short b;
short c;
a = 22;
b = 5;
c = a % b;
```

Here, *c = 2,* which is really *a - (a / b).*

The modulus operator should be used with integers only; not with floating point numbers. It can be used to determine if the variable is odd.

Bitwise AND — &

A bitwise **AND** takes each bit of the left expression and performs an **AND**ing function with each bit of the right expression. For instance:

```
char a;
char b;
char c;
a = 0x55;
```

```
b = 0x0F;
c = a & b;
```

In the above example, *c = 0x05*. In the binary system, the numbers look like this:

```
a       0101 0101
b       0000 1111
```

Using the truth table for an **AND** function, the result will be:

```
c       0000 0101
```

or:

```
        0101 0101
&       0000 1111
        ---------
        0000 0101
```

Note:

When we ANDed the 55 with a 0F, the upper nibble of the 55 became 0, and we were left only with the lower nibble, which is a 5. In a sense, we masked out the upper nibble. When you want to mask out certain bits, use a 0 in the masking number. For those that you want to keep unmasked, use a 1 in the masking number.

Bitwise Inclusive OR — |

A bitwise **OR** takes each bit of the left expression and performs an **OR**ing function with each bit of the right expression. For instance:

```
char a;
char b;
char c;
a = 0x55;
b = 0xAA;
c = a | b;
```

In the above example, *c = 0xFF*. In the binary system, the numbers look like this:

```
a       0101 0101
b       1010 1010
```

Using the truth table for an **OR** function, the result will be:

```
c      1111 1111
```

or:

```
       0101 0101
|      1010 1010
       1111 1111
```

Bitwise Exclusive OR(XOR) — ^

A bitwise **XOR** takes each bit of the left expression and performs an **XOR**ing function with each bit of the right expression. For example:

```
char a;
char b;
char c;
a = 0x55;
b = 0xFF;
c = a ^ b;
```

In the above example, *c* = *0xAA*. In the binary system, the numbers look like this:

```
a      0101 0101
b      1111 1111
```

Using the truth table for an **XOR** function, the result will be:

```
c      1010 1010
```

or:

```
       0101 0101
^      1111 1111
       1010 1010
```

Left Shift — <<

A left shift takes all the bits in the expression and shifts them to the left side by the number of places indicated. For instance:

```
char a;
char b;
char c;
a = 0x55;
b = 1;
c = a << b;
```

In the above example, *c* = *0xAA*. Before the shift:

a   0101 0101, then shifted by *b* (1)

After the shift:

C   1010 1010

Right Shift — >>

A right shift takes all the bits in the expression and shifts them to the right by the number of places indicated. For example:

```
char a;
char b;
char c;
a = 0xAA;
b = 1;
c = a >> b;
```

In the above example, *c* = *0xD5*. Before the shift:

a   1010 1010, then shifted by *b* (1).

The right shift fills with 0s if the left operand is unsigned. Otherwise, the fill is a copy of the signed bit; in this case, a *1*.

After the shift:

```
c       1101 0101
```

Equal (or replacement) — =

The equal operator totals any preceding operation(s). For example:

```
char a;
char b;
char c;
a = 6;
b = 2;
c = a/b;
c = 3
```

In the above example, *c = 3* is the result of a division operation (*a/b*). The more complex use of the = operator as an assignment operator is discussed in the next section.

Binary Operators—Assignment

The *equal* is an assignment statement that has a low precedence. Assignments are evaluated from right to left. In C++, the = symbol is an operator, although in other languages it is not. The difference is that in C++ a statement can contain more than one = sign. For example:

```
short a;
short b;
short c;
a = b = c = 1;
```

The above statement assigns the value *1* to *c*. It sets *b* to be the value of *c*, and sets *a* to be the value of *b*. Typically, the = assignment operator produces assembly code for something like *a* = *b* :

```
MOVE b, al;
MOVE al, a;
```

Here, the assembler takes the value of *b* (right-hand side) and puts it into a temporary register. (A register is a working storage location that's inside the 68000 microprocessor.) It then places the contents of the temporary register into *a* (the left-hand side). Therefore, in C++, as in most high-level languages, the left-hand side of the assignment must be an expression that refers to storage in the machine, which is referred to as an *lvalue.* The right-hand side of the assignment can be a storage value, an expression, or a constant. It is referred to as an *rvalue.* (The *rvalue* may be read but not altered, so you can think of it as a *read value.* The *lvalue* is the memory location where the result is written, so you can think of it as a *location value.*) Figure 2.5 shows the location of the *rvalue* and *lvalue.*

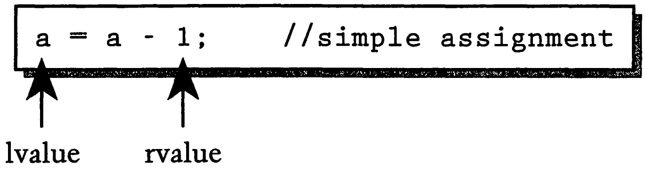


Figure 4.5 *l and r values.*

The code below illustrates the concept of the *lvalue* and *rvalue*:

```
short a;
short b;
a + 1 = b;
```

The above is not valid. *a + 1* is not an *lvalue*; that is, it is not a valid memory location. An error message will come up saying that you have an invalid *lvalue*. Instead, it's an expression. What would be acceptable is the following:

```
short *a;
short b;
*(a + 1) = b;
```

Here, *a* is a pointer to a *short*, and we're taking *a*, which is a memory address, adding *1* to it, and placing *b* in the contents of that memory address.

Also, you may not say:

```
short a;
0 = a;
```

You will get the same error message because *0* is a constant, not a memory location.

Assignment Operators

=	Assignment
+=	Addition update
– =	Subtraction update
*=	Multiplication update
/=	Division update
%=	Modulus update
<<=	Left shift update
>>=	Right shift update
&=	Bitwise **AND** update
|=	Bitwise **OR** update
^=	Bitwise **XOR** update

Addition Update — +=

If you want to increment a value by 1, you can do the following:

```
short a;
a = a + 1;
```

Another way to do this is to use the += operator, as follows:

```
short a;
a += 1;
```

All of the other operators that contain the = symbol work in the same way. If you write *a (operator) = b*, it results in *a = a (operator) b*. Another, more challenging example is below:

```
short a = 5;
short b = 7;
short c = 3;
a += b++ + ++c;
```

In this example, *c* is incremented by 1, so *c* now contains the value of *4*. Then, the value of *c* is added to *b*, and that result is added to *a* and stored back in *a*. So *a* will be equal to *4* + *7* + *5*. In addition, *b* will be incremented so that after this expression is fully executed, *b* will have the value of *8*. Another way to state the last line is:

```
a+=b+++++c;
```

However, this expression is confusing. That is why C++ rules include inserting a space between binary operators and no space between the operator and the operand for unary operators.

Binary Operators—Comparison

Comparison binary operators provide a logical result that is either true or false. These operators are used to compare things, as in the following example:

```
short a;
short b;
short c;
a = 2;
b = 5;
c = a < b;
```

The above example compares *a* and *b*. Since *a* is less than *b*, the result of the operation is a *true*. (A *true* in C++ is defined as something that is not *0*.) A *false* would be *0*. For all comparison operators, the result is either *1* or *0*; that is, true or false. If *a* is less than *b*, then the value of *c* is *1*, or true. Otherwise, *c* is *0*, or false.

Comparison Operators

==	Equality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Inequality
&&	Logical **AND**
||	Logical **OR**

The equality operator compares two variables to see if they are equal to each other. For instance:

```
short a;
short b;
short c;
a = 2;
b = 5;
c = a = b;
```

The above example sets the value of *a* to the value of *b*, then sets the value of *c* to the value of *a*. The result is that *a, b,* and *c* all have the value of 5. However, the statement does not perform a comparison to see if *a* is equal to *b* and set *c* to *true* or *false,* depending on the result of the comparison. To do such a comparison, write the following:

WARNING

A common mistake made in C programming is to use the = operator when the == operator is intended. This is especially true when the == operator is used in the control statements; that is, an if or while statement.

```
short a;
short b;
short c;
a = 2;
b = 5;
c = a = = b;
```

In this example, *a* is compared to *b*. Because they are not equal, it sets the value of *c* to *false.*

The <, >, <=, >=, and != operators all have the same function as the == operator; that is, they compare two variables and provide a result that is either true or false.

Logical AND and Logical OR

These operators take the logical value of the two operands and state either, "If a is this AND b is that, then c will be *true* (or *false*)," or "If a is this OR b is that, then c will be *true* (or *false*)."

```
short a;
short b;
short c;
a = 2;
b = 5;
c = a && b;
```

In the above example, if *a* is not equal to *0* (i.e., *a* is *true*) and *b* is not equal to *0,* then *c* will be set to *true.* This is quite different from saying:

```
short a;
short b;
short c;
a = 2;
b = 5;
c = a & b;
```

The above example takes the hex number *0x0002* and performs a bitwise **AND** with the number *0x0005.* The result placed in *c* will be *0x0007.*

The && operator is normally used in statements like the following:

```
short a = 2, b = 5, c = 7, d = 1, e;
e = (a < b) && (c > d);
```

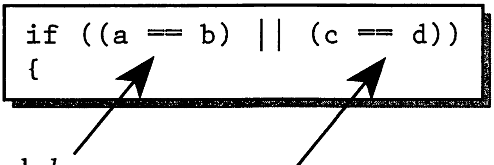
The result of the above operation is that *e* is equal to *true.*

The **OR** comparison (||) operates similarly, except that it states, "This or this," instead of "This and this." For example:

```
short a = 2, b = 5, c = 7, d = 1, e;
e = (a == b) || (c == d);
```

Here, if *a* is equal to *b* or *c* is equal to *d,* then *e* will be equal to *true.* However, that is not the case, so *e* is equal to *false.*

The && and || operators are evaluated from left to right; that is, as soon as the result of the left-hand variable is known, the operation will determine whether it needs to evaluate the right-hand variable. If not, it will set the assignment to true or false immediately. The advantage of this is speed. For efficiency's sake, you should put the variable most likely to affect the outcome in the left-hand position. Figure 2.6 shows this short-circuit evaluation.



```
if ((a == b) || (c == d))
{
```

If *a* equals *b,*
then the expression (*c* == *d*) is not evaluated

Figure 4.6 *Short-circuit expression evaluation.*

Ternary Operator — ?:

The ternary operator provides a choice between two alternatives. For instance:

```
short a, b, c, d;
d = c ? a : b;
```

The above example can be expressed as *d* equals *a* if *c* is *true,* otherwise *b.* Thus, the value of *d* will be either *a* or *b.* It is not a logical value. A typical example of this is:

```
short a,b,c;
c = (a < b) ? a : b;
```

This statement performs a minimum (min) function; that is, it takes the smaller of two values, either *a* or *b*. Another example might look like the following:

```
 short a,b,c;
c = (a > b) ? a : b;
```

The above example describes a maximum (max) function because it takes the larger of the two. Another example might look like:

```
short a,b;
b = (a < 0) ? -a : a;
```

Here, if *a* is less than *0* (is a negative number), then set *b* to *-a* (which is a positive *a*); otherwise (*a* is positive), set *b* to *a*. This is referred to as an *absolute value function.*

Comma Operator

The comma operator is used to separate a series of expressions. These expressions are evaluated from left to right. (It is important not to confuse the comma operator with the statement/end separator—the semicolon.) Here is an example of the use of the comma operator in a complex statement:

```
short i = (ia !=0) ?
      ix = 5, ia[ix] =, 1:
      ix = 6, ia[ix] = ix - 1,0;
for (short i = 1, short j = 1; i < 10; i++, j++)
{


a = 100; b = 200; c = 300;
```

Summary

In this chapter, you have learned the elements of the C++ language:

- Simple statements.
- Variables.
- Basic data types.
- Operators.

In the next chapter, we will introduce you to the basics of program flow and will discuss C++ standards of style.

Exercises

1) Which comments are valid?

a) `// Macintosh`

b) `/* Macintosh`

c) `/* Macintosh */`

d) `/* Macintosh /* Computer */ System */`

e) `// Macintosh // Computer`

f) `#define pi 3.14 // π`

g) `y = a*x/*p; /* simple equation */`

h) `“/* Macintosh */”`

i) `“// Macintosh”`

j) `// /* Macintosh */`

k) `/* // Macintosh */`

2) Debug the following:

```
main()
{
  short library;
  short public = 1;
  short private = 2;
  short i;
```

```
    cin >> i;
    if (i == 0)
    {
      library = public;
    }
    else
    {
      library = private;
    }
    return (0);
  }
```

3) Debug the following:

```
main()
{
  char system[9] = "Macintosh";
  cout << system;
  return (0);
}
```

4) What is the value of x on each line of the code below:

```
main()
{
  short x;

  x = 5;
  x -= (4 * 2) - 6;
  x /= 5;
  x-;
  x = short(25.0);
  x /= 3 * 2;
  x = (3 * 4 * 5) / 9;
  x  -= (3 + 4) * 2;
  x = sizeof(long) + 1;
  x = (3 + 4) * 2;
  x = sizeof(char);
  x /= (3 + 4) / 6;
}
```

5) Which of the following are *true* or *false*:

a) `10 == 5 * 2`

b) `0 && 0`

c) `12 || 0`

d) `short a = 1, b = 2; a > b || b > a;`

5 Controlling the Program Flow

C++ programs derive from *control structures* that allow you to determine which operations the computer will perform and in what order. The structures determine the flow of control of the program.

C. Böhm and G. Jacopini stated in *Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules* that any algorithm could be coded in a computer language using only three control structures: sequential execution, conditional execution, and looping. Sequential execution is the most common and is usually part of a simple statement or block of statements. Conditional execution and looping are more complex. All are covered in the following subsections.

Statements

All C++ programs are made up of statements, which are followed by a semicolon. The several types of statements are described below.

Expression and Null Statements

The most common statement is the *expression statement*. An expression is a statement that describes the relationship between variables and operators. For example:

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

The above expresses *a*'s relationship to *b* and *c* using the = and + as operators.

C++ can also have a *null statement*, which is a statement for which no code is generated. It is equivalent to an assembly language *No-op*. For instance:

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

In the above example, nothing happens between the two semicolons. This example is not a useful one, but there are instances where a null statement can be useful.

Beyond the *expression* and *null* statements, there are other statements in C++ that use keywords.

Blocks

A *block*, or *compound statement*, is a collection of statements enclosed by a pair of curly braces. Thus, from the beginning of an open curly brace to the closed curly brace, all of the statements contained therein constitute a block. We have used blocks in all of our code examples, although we have not always shown the curly braces because most of the examples are code fragments. The following is an example of a compound statement using the curly braces:

```
main()
{
  short a, b, c, d;
  c = a + b;
```

```
    d = a + c;
  }
```

The three lines between the curly braces make up a block.

If

The *if statement* is the simplest of all the control statements. An *if* takes the following form:

```
  if (expression) statement;
```

The *if* statement simply means that the statement executes if the expression is true; that is, if everything inside is not equal to 0 (the C++ definition of *true*). However, you can say in the expression that something is equal to 0, and it will execute because the expression is true. The following is a simple example of an *if* statement:

```
  short a;
  if (a < 0) a = -a;
```

The above example says that if *a* is less than *0*, then a is equal to *–a*. In other words, it takes the absolute value of *a*. Another way to write this is:

```
  short a;
  if (a < 0)
    a = -a;
```

This is exactly the same as the first example but shows that C++ ignores intervening lines and white spaces.

The statement in the *if* statement can be a block statement. For instance:

```
  short a, b, c;
  if (a != 0)
  {
    b = 10;
    c = 20;
  }
```

The example above has a compound (block) statement between the curly braces. A variation of the above example might be:

```
short a, b, c;
if (a)
{
  b = 10;
  c = 20;
}
```

This code does exactly the same thing as the previous code because the expression still says, "if *a* is not equal to *0.*"

It is important to note that the logical expression inside the parentheses can be interpreted differently for the =, &, and | operators. For example, if you are comparing variables—*a* equals *b,* and *c* equals *d*—the rational thing to write would be:

```
short a, b, c, d, e;
if ((a = b) & (c = d)) e = 0;
```

This appears to say, "If *a* is equal to *b* and *c* is equal to *d,* then set *e* to *0.*" What actually happens is that *a* is set equal to *b,* clobbering *a,* and *c* is set equal to *d,* clobbering *c,* and if the bitwise **AND**ing of *a* and *b* is not equal to *0, e* will be set to *0.* The correct way to write that statement is:

```
short a, b, c, d, e;
if ((a == b) && (c == d)) e = 0;
```

By using the == and &&, the above example evaluates the logical expressions this way: "If *a* is equal to *b* and *c* is equal to *d,* then set *e* equal to *0.*" It compares *a* to *b* and *c* to *d* rather than setting values.

Even if you intend to do an assignment in the expression part of an *if* statement, the best way to write this would be:

```
short a, b, c;
a = b + 1;
if (a != 0) c = 0;
```

A crackerjack C++ programmer might look at that and, for conservation of keystroke purposes, might express it in the following manner:

```
short a, b, c;
if (a = b + 1) c = 0;
```

This will execute the same as the first example, but it is poor style. If at some future time you wanted to modify your code, you might not know what was originally intended.

C++ has unusual rules for curly braces. The language allows you to write an *if* statement with an else statement without braces—if there is only one line of code after each statement. However, if you do not use braces, you stand the chance of making a disastrous mistake. Thus, a good example of code is:

```
if (i < 0)
{
  do one line
}
else
{
  do another line
}
```

A bad example might be:

```
if (i < 0)
  do one line
else
  do another line
```

The reason that the preceding example is considered poor style is that it lends itself to error. That is because almost every line of code in C++ has a semicolon after it, and it is remarkably easy to insert a semicolon where one does not belong. For instance, it would be effortless to write:

```
if (a = b);
  c = d;
```

Let's assume that in this example you wanted to say, "If *a* equals *b*, then *c* equals *d*." Instead, you are saying, "If *a* equals *b*, then do nothing and always set *c* to *d*." The best way to avoid this error is to use braces, as follows:

```
short a, b, c, d;
if (a != b)
{
  c = 10;
  d = 20;
}
```

Here, if you inadvertently put a semicolon after the *if (a != b)* expression, the brace on the following line automatically flags the error. (This is true **only** in the Symantec compiler.)

If –Else

An *if–else statement* includes the alternative condition; that is, *if* the expression is true, execute the first statement, or *else* execute the second statement. The syntax of the *if–else* is:

```
if (expression) first statement;
else second statement;
```

A simple example would be:

```
short a, b;
if (a < 0) b = -a;
else b = a;
```

An alternative way of saying the above would be:

```
short a, b;
if (a < 0)
  b = -a;
```

```
else
  b = a;
```

Even though this example is acceptable syntax in C++, it is poor style for the reasons discussed in the previous section: Since most lines are terminated by a semicolon, it's easy to insert one in the wrong place. You might inadvertently write:

```
if (a < 0);
  b = -a;
else
  b = a;
```

What would happen above is that if *a* were less than *0,* nothing would happen (this is a null statement) and *b* would always be set to *–a.* A better way to say this is:

```
short a, b;
if (a < 0)
{
  b = -a;
}
else
{
  b = a;
}
```

This makes the statement clearer and more goof-proof. It also assures that the compiler will pick up any mistakes.

Else–If

The *else–if statement* is used to implement multiple-choice statements. In essence it says, "If the first expression is true, execute the first statement; else if the second expression is true, execute the second statement."

The form for an *else–if statement* is:

```
if (first expression) first statement;
```

```
else if (second expression) second statement;
.
.
.
else last statement;
```

A simple example would be:

```
char  answer;
short result;

if (answer == 'A') result = false;
else if (answer == 'B') result = false;
else if (answer == 'C') result = true;
else if (answer == 'D') result = false;
else if (answer == 'E') result = false;
else result = false;
```

The above example says: "If *C* is the answer, the result is true. If *A, B, D, E,* or anything else is the answer, then the result is *false.*"

Switch

A *switch statement* executes a particular action depending on whether an expression matches one of a number of constant values. A *switch* statement has the following form:

```
switch (expression)
{
  case first constant:
    first statement;
  case second constant:
    second statement;
  .
  .
  .
```

```
    default:
      last statement;
  }
```

In the above example of code, the lines with a case and constant (or default) are called case labels. These end in a colon. The case and default labels can occur in any order, but by convention the default case is usually last. You cannot have two instances of the same case label; each must be unique.

When the *switch* expression matches the case constant, the statement following that case label is executed. In addition, every statement from the first matched statement to the last is executed. For example:

```
short i;
char theCharacter;
i = 0;
switch (theCharacter)
{
  case ('4'):
    i++;
  case ('3'):
    i++;
  case ('2'):
    i++;
  case ('1'):
    i++;
  default:
}
```

WARNING

Although the default case is optional, it is good programming practice to have a default case in the event that none of the other cases matches. This way you know that the *switch* statement has executed properly. By convention, the default case is usually the last case in the *switch* statement.

In this example, we have two variables: *theCharacter*, which is some ascribable ASCII value, and *i*, which is a short number. In the first line of code, we set *i* to *0*. In the second line, we have the variable *theCharacter*, which is what we will be testing on. Each of the case expressions contains an ASCII constant. For example, *case ('4')* is *0x34*, *case ('3')* is *0x33*, and so on. Let us suppose that *case ('4')* is a match, then the line following that case label—*i++*—gets executed. When that statement is executed, *i* is now equal to *1*. The next thing that happens is that the program falls through and executes the statement following the next case label and ignores the case label; in this *case ('3')*. Each additional statement under each case label, including the default, is also executed, thereby changing the value of *i* in each case.

If you do not want the fall-through feature for a particular case, you must use the *switch–break statement*, which is covered in the next section.

Switch–Break

The *switch–break statement* is just a *switch* statement with the addition of a *break* statement. When a *break* statement is executed, it forces the program to branch out of the *switch* statement; that is, it does not fall through. The format for a *switch* statement using breaks is:

```
switch (expression)
{
  case first constant:
    first statement;
    break;
  case second constant:
    second statement;
    break;
  .
  .
  .
  default:
    last statement;
    break;
}
```

For performance purposes, put the case that is most likely to match first in the *switch–break* statement (may vary from complier to compiler). An example of a *switch–break* statement might be:

```
short i;
char theCharacter;
switch (theCharacter)
{
    case ('9'):
    case ('8'):
    case ('7'):
    case ('6'):
    case ('5'):
    case ('4'):
    case ('3'):
    case ('2'):
    case ('1'):
    case ('0'):
      i = theCharacter - 0x30;
      break;
    case ('A'):
      i = 10;
      break;
    case ('B'):
      i = 11;
      break;
    case ('C'):
      i = 12;
      break;
    case ('D'):
      i = 13;
      break;
    case ('E'):
      i = 14;
      break;
    case ('F'):
      i = 15;
      break;
    default:
      i = 0;
      break;
}
```

In this example, the first 10 lines after the open curly brace are fall-through cases; that is, they are multiple case labels attached to one statement. For each of those cases, we have an ASCII character from which we subtract *0x30*, thereby setting *i* to the decimal equivalent. After that, we hit the *break* statement, which forces us to branch out of the *switch* statement and not execute any of the other cases in the block. In the other cases—*A* through *F*—we assign *i* to the decimal of the hex digits. If no match is found, the default is executed.

For sanity's sake, it is best to avoid fall-throughs except to prevent inefficient duplication of code lines. Fall-throughs are best used when you have multiple labels for a single computation.

Which Do I Use—Switch or If-Else?

Since these two statements do almost the same thing, it is sometimes difficult to know which to use. We suggest that you use a *switc*h statement if at all possible. The following rule will help: If you are matching an expression to a constant, use a *switch* statement. If you are matching an expression to an expression, use an *if-else* statement. For example:

Switch:

```
case ('A'):
  i = 10;
  break;
case ('B'):
  i = 11;
  break;
case ('C'):
  i = 12;
  break;
```

If–Else:

```
if
  a = b
else
  a < b
  a > b
```

and so on.

While

The *while statement* executes a statement as long as a specific expression is true. The test for a logical true is made each time before the statement is executed. If the expression is always false, the statement is never executed.

The format for the *while* statement is as follows:

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

The statement can be compound. The test for the expression is always done at the top of the loop. An example of a *while* statement might be:

```
i = 0;
while (i < 100)
{
  i++;
}
```

Note:

Variables that retain a value based on the number of times through a loop are called counters. Typically, programmers designate these counters as *i, j, k, l, m,* and *n.* Originally, in Fortran (the mother of all programming languages), the first letter of any integer variable had to begin with one of those letters. This tradition has been handed down to C++.

The above statement changes the value of *i* by 1 each time you go through the loop. In this example, the statement will be executed 100 times, but the test for the expression will be executed 101 times.

The statement has to affect the expression in some way. Otherwise, you will have an endless loop, because the expression would always be true. At some point, the statement has to render the expression false.

Another way to write the above example might be:

```
i = 0;
while (i < 100)
  i++;
```

Note:

For the most part, the only way out of an infinite loop on the Macintosh is to restart the machine.

The problem with the style of this example is the tendency to put a semicolon after *while (i < 100).* Doing this puts the program into an infinite loop.

Do–While Loop

The *do–while statement* is similar to the *while* statement, except that the test and evaluation of the expression are done at the bottom of the loop. The format is:

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

The statement can be single or compound.

Notice that the test and evaluation for the expression come at the end of the loop. An example of a *do–while* statement is shown in the following:

```
i = 0;
do
{
  i++;
} while (i < 100);
```

Here, the statement will still be executed 100 times, and the test for the expression will be executed 100 times as well. You only use the *do–while* statement when the problem you're trying to code dictates that the statement be executed at least once before the expression is evaluated.

For Loop

The *for loop,* while similar to the *while* statement, can improve the readability of your code. It does this by confining the initialization, testing, and evaluation of the loop counter on a single line. The format for the *for loop* is:

```
for (initial exp; test exp; evaluate exp) statement;
```

or:

```
initial exp;
while (test exp)
{
  statement;
  evaluate exp
}
```

The test of the expression comes at the top of the loop, while the evaluation of the expression always comes at the bottom.

To see how a *for loop* improves the appearance of a complicated *while* statement, look at the following:

```
i = 0;
while (i < 100) //This is a while statement
{
  y[i] = a[i] * x[i] + b[i];
  i++;
}

for (i = 0; i < 100; i++)  //This is a for loop
{
  y[i] = a[i] * x [i] + b[i];
}
```

The *for loop* in this example keeps all the operations of the loop counter on one line, and it also reduces the code by two lines.

If the initial expression is left blank, no initialization takes place. If the evaluation expression is left blank, no evaluation takes place. If, however, the test expression is left blank, the loop executes forever. An example of an infinite *for loop* would be:

```
for (;;)
{
  do stuff here ;
}
```

Another thing that C++ allows inside a *for loop* is use of the comma operator. This lets you use multiple expressions on the same line. For example, look at the following *for loop* without the comma operator:

```
j = 0;
for (i = 0; i + j < 100; i++)
{
  do stuff here ;
```

```
    j++;
  }
```

This same loop using the comma operator would be written this way:

```
  for (i = 0, j = 0; i + j < 100; i++, j++)
  {
    do stuff here;
  }
```

You use a comma instead of a semicolon because a compiler uses the semicolon to delineate the initialization, test, and evaluation of expressions. In addition, the compiler uses the semicolon to separate multiple statements on a single line. If you want to use multiple statements for the initialization, test, and evaluation expressions, the use of a semicolon would be confusing to the compiler. Therefore, to separate multiple statements in the *for* expressions, use the comma.

WARNING

The initialization and evaluation expressions do not have to be connected to the test expression in any way, although it is good practice to connect them. The test expression, however, has to be able to terminate the loop or an infinite loop will be created.

While versus For

For code optimization and efficiency, it is sometimes more desirable to use a *while loop* instead of a *for loop* and vice versa. Here are a couple of rules of thumb:

- If you have a *for loop* with the first and third expressions omitted, use a *while loop*.
- If a loop depends on a simple comparison for repetition and does not use an index variable, use a *while loop*.
- Otherwise, use a *for loop*.

Break

A *break statement* inside a *while, do,* or *for* loop terminates the loop. The format is:

```
  break;
```

WARNING

Use the *break* statements to terminate a loop sparingly, because such a construction is not conducive to structured thinking; that is, there is probably another way to write the loop and express the same logic without the use of a break. However, the *break* statement is a necessary evil in a *switch* statement because it is the only way to avoid the fall-through.

An example of a *break* statement is:

```
for (i = 0; i < 100; i++)
{
  do stuff here;
  if (this is true) break;
  do more stuff here;
}
```

WARNING

Use the *continue* statement sparingly for the same reasons that you use the *break* statement sparingly.

The break can occur anywhere inside the loop. Note that a *break* statement inside a *switch* statement that is inside a loop will terminate the *switch* statement only; it will not terminate the loop.

Continue

The *continue statement* causes a loop to recycle by branching to the place in the loop where the evaluation and test occur. For example:

Note:

The *continue* statement has no effect on *switch* statements.

```
for (i = 0; i < 100; i++)
{
  do stuff here;
  if (this is true) continue;
  do more stuff here;
}
```

In the example above, anything after the *continue* statement will not be executed if the *if* statement is true; the loop will then test and evaluate the expression again for continuation or termination.

Labels and Goto

Any statement may be preceded by a label. The format for a label is:

```
identifier:
```

The only use for a label is to be the target of a *goto statement*. A *goto* is a way to transfer unconditionally to a label. The format of a *goto* is:

```
goto identifier;
```

WARNING

The *goto* statement should be avoided as much as possible because it misses the underlying syntax of the problem and represents one-step thinking.

An example might be:

```
for (i = 0; i < 100; i++)
{
  for (j = 0; j < 100; j++)
  {
    do stuff here;
    if (an error detected) goto fixit;
  }
}
     fixit:
handle error here;
```

In this example, we have nested loops—that is, a loop containing a loop. Inside the loops, we executed statements, and we have an *if* test to determine if an error occurred in the calculation of those statements. If it did, we executed the *goto,* transferring control unconditionally to the label *fixit,* where we execute more statements to handle the error.

The above example is probably the closest to a valid use of the *goto* statement. However, that same example without the *goto* statement could be written:

```
for (i = 0; i < 100 && error == false; i++)
{
  for (j = 0; j < 100 && error == false; j++)
  {
    do stuff here;
    if (an error detected) error = true;
  }
}
if (error = = true)
{
  handle error here;
}
```

In this example, we have the advantage of retaining a structured way of thinking; the disadvantage is that we have introduced a new variable.

Note:

In the above example, you could not use *break* statements to accomplish what the *goto* accomplishes. If you place a *break* statement in the innermost loop of the two nested loop statements, you would break out of the inner loop but remain in the outer loop.

Style

Although C++ is a popular, widely used language, it can be terse and demanding, and programmers must follow prescribed conventions. For that reason we devote this section to a discussion of C++ programming standards.

Error from Fortran

What was intended:

```
   DO 10 I=1, 23
     do stuff here
10 CONTINUE
```

What was coded:

```
   DO 10 I1.23
     do stuff here
10 CONTINUE
```

What the compiler saw:

```
DO10I=1.23
do stuff here
10CONTINUE
```

What happened:

```
     DO10I = 1.23
     do stuff here
10   CONTINUE
```

This error was found in a program that was used to compute the trajectory of a multimillion-dollar communications satellite. After launch, no trace of the satellite was ever found.

Errors like this can happen with an old dinosaur like Fortran but can never happen with an advanced language like C++...can they? Let's take a look.

What was intended:

```
a = b/*p          /* Div b by the contents of p */;
```

What happened:

```
a = b             /*/* Div b by the contents of p */;
```

What should have been done:

```
a = b / *p;       /* Div b by the contents of p */
```

The point of all this is to urge you to be consistent in your methods for C++ code.

Rule 1: Placement of Curly Braces

Even the experts disagree on where the curly braces should go, as seen in the following.

The curly brace rule according to Kernighan and Ritchie:

```
if (expression) {
  statements
}
```

According to Plum:

```
if (expression)
{
  statements
}
```

According to Whitesmith:

```
if (expression)
  {
    statements
  }
```

Rule 2: Use Curly Braces on All If Tests

This:

```
if (i < 0)
{
  do one line
}
else
{
  do another line
}
```

Not this:

```
if (i < 0)
  do one line
else
  do another line
```

Okay:

```
if (a == b)
  c = d;
```

Wrong:

```
if (a == b);
  c = d;
```

Best:

```
if (a == b)
{
  c = d;
}
```

What the programmer thought he/she had:

```
if (a <= b)
  if (a == b)
    i++;
else
{
  a = b;
  i = 1;
}
```

What the programmer really had:

```
if (a <= b)
  if (a == b)
    i++;
  else
  {
    a = b;
    i = 1;
  }
```

The fix:

```
if (a <= b)
{
  if (a == b)
  {
    i++;
  }
}
else
{
  a = b;
  i = 1;
}
```

Rule 3: Space Between Statement Keywords and Parentheses

This:

```
for (i = 1; i <= 10; i++)
if (a == b)
```

Not this:

```
for(i = 1; i <= 10; i++)
if(a == b)
```

Rule 4: No Space Between Function Name and Parentheses

This:

```
MyFunction(variable)
```

Not this:

```
MyFunction (variable)
```

Rule 5: Use Spaces Between Binary Operators (No Spaces Between Unary Operators)

This:

```
a = i++ + ++j * *k
```

Not this:

```
a+i+++++j**k //Huh??
```

Rule 6: Use Spaces After Commas and Semicolons

This:

```
MyFunction(theVar1, theVar2, theVar3)
```

Not this:

```
Myfunction(theVar1,theVar2,theVar3)
```

This:

```
for (i = 1; i <= 10; i++)
```

Not this:

```
for (i = 1;i <= 10;i++)
```

Rule 7: Capitalize Every Main Word in a Function Name

This:

```
MyFunction(theVar)
```

Not this:

```
myFunction(theVar)
```

Or this:

```
myfunction(theVar)
```

Or this:

```
MYFUNCTION(theVar)
```

Or this:

```
My_Function(theVar)
```

Rule: 8 Capitalize Every Main Word in a Variable Name Except the First

This:

```
short eventRecord;
```

Not this:

```
short EventRecord;
```

Rule 9: Use Blank Lines Only When They Convey Meaning

For example:

```
HLock(theHandle);
thePointer = *theHandle;


pi = 3.14;
f = 1.0/ (2.0 * pi * SquRoot(f * c));
*thePointer = 1.0 / f;


HUnlock(theHandle);
```

Rule 10: Go Easy on the Use of Underscore (_)

This:

```
theWindowDefProc = MyDefinitionRoutine;
```

Not this:

```
The_Window_Def_Proc = My_Definition_Routine;
```

Rule 11: Use a Break on the Last Case of a Switch Statement

This:

```
switch (theVar)
{
  case (1):
    do stuff here
    break;
  case (2):
    do more stuff here
    break;
  default:
    break;
}
```

Not this:

```
switch (theVar)
{
  case (1):
    do stuff here
```

```
        break;
      case (2):
        do more stuff here
  }
```

A case label that deliberately omits a *break* statement should in most cases provide a comment stating that the omission is deliberate.

Rule 12: Operators in Definition and Declaration Statements

This:

```
    char *p1;
```

Not this:

```
    char* p1;
    char* p1, p2; // Could be a problem
```

In the above example, *p1* is a pointer to a character, where *p2* is a character.

Summary

- Be consistent: Choose a style and stick with it!
- Your style should help you program defensively.
- Code for readability: Be kind to those programmers who follow you.

Exercises

1) Write a program that will create an array on the heap, then initialize that array to zero.

2) Debug the following function:

```
void PrintCanine(short canine)
{
  switch (canine)
  {
    case 1: cout << “Doberman”;
    case 2: cout << “German Shepherd”;
    case 3: cout << “Weimaraner”;
  }
}
```

3) Debug the following program:

```
main()
{
  short array[1000];
  short i;

  for (i = 0; i <= 1000; i++);
    a[i] = i;
}
```

4) Write your own routines to:

   a) compute the minimum and maximum of two variables.

   b) compute the absolute value of a variable.

5) What is wrong with the following code fragments:

   a) `do (i++) until ( i == 100);`

   b) `while () i++;`

   c) `for (i = 0; i < 100; i++) i–;`

6) Create a file call MyStyle. In that file define a set of rules that describe your coding style. The file should include:

   a) how much white space you use to indent code.

   b) how much white space you use before and after operators.

   c) how you use characters in variable and function names.

   d) any rules that apply to control statements.

6 Functions and Variables

We covered variables (and to some extent functions) in Chapter 4. In this chapter, we show you how these elements interact.

Functions

A function is a collection of statements that perform a particular task. In a well-written program, a function will perform only one task. Functions break a program up into parts that are reusable and can be saved in a library, which keeps you from reinventing the wheel every time you want the same task performed. Functions also make your code more readable and easier to maintain.

When you write a function, you might ask yourself, "Does it make the code more readable and does it hide the code (along with all the thought processes that go on behind it)?" If the answer is yes, write the function.

Note:

Common programming practice on the Macintosh is to capitalize the first letter of each major word in the function name. This is different from a variable, which has a lowercase letter for the first letter of the first word but an uppercase letter for the first letter of each major word following. An example of a function name is EventRecord. Some programmers like to use underbars to replace spaces in names of functions and variables. For example, Pen_Normal. This style is not really popular on the Mac, probably because the Toolbox does not use it.

You can think of a function as a kind of black box, with data going in and out. The inside of the box is invisible to the rest of the program. It is not necessary to know what is going on; you need only know what goes in and what comes out.

For every program, you must have at least one function, and it must be called *main*. The main function controls the execution of the program and calls other functions, which in turn call still more functions.

When a program calls a function, control is passed to that function; that is, when the program makes a call to the function, it stops executing operations and passes them on to the function, which executes until it encounters a return or the end of the function. Figure 6.1 depicts a function call, execution, and return to the next statement.

You call a function by stating its name followed by the function operator. For example:

```
PenNormal();
```

When designing functions, try to keep them short. A function should be limited to one printout page, about 50 to 60 lines. Small functions are easier to maintain.

A Function Definition

A function that does not indicate a return type is presumed to return an *int* value. For example:

Functions

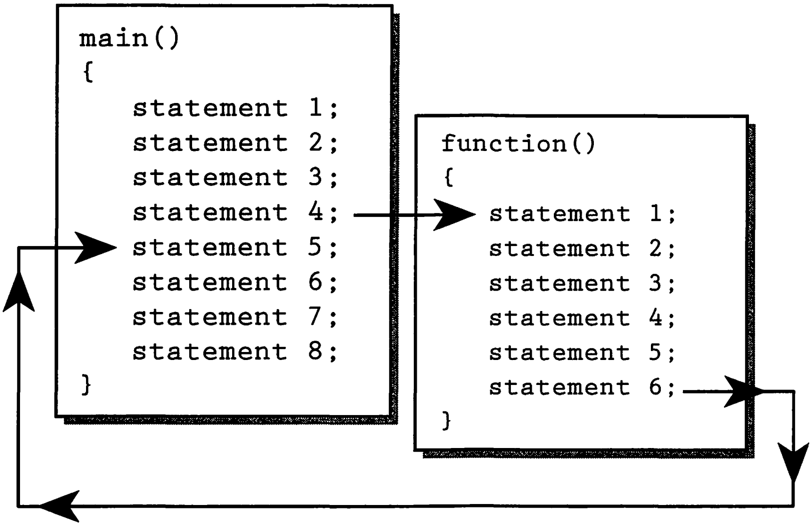


Figure 6.1 *Function call, execution, and return.*

```
return-type Name (argument list)
{
  declarations
  statements
}
```

A *return* statement provides a method for terminating the execution of a function. The return of a zero in the main function indicates the successful completion of the main. The form for a *return* statement is as follows:

```
return;
```

or:

```
return variable;
```

or:

```
return (variable);
```

A Function Definition Example

```
short Name (short top, short bottom)
{
  short temp2Bytes;
  long temp4Bytes;

  temp4Bytes = bottom - top;
  temp2Bytes = (short)temp4Bytes;
  return (temp2Bytes);
}
```

Function Prototypes

A function prototype is a mechanism used in C++ to improve program reliability. All functions must have their type and arguments explicitly listed before they are used or defined. If a function is not declared to the program before it is used, a compile-time error will result.

Prototypes are also known as *forward declarations*. They have the following form:

```
type name (argument-declaration list);
```

A typical example of a prototype might be:

```
void PenNormal(void);
void SetPort(GrafPtr thePort);
void SetPt(Point thePoint, short h, short v);
short StringWidth(Str255 theString);
```

or:

```
float squ(float x);  //Prototype

main()
```

```
{
  float pi;
  float radius;
  float area;

  radius = 5.0;
  area = pi * squ(radius);
```

You may omit parameter names from the prototype (only the types are important). For instance:

```
void setRect(Rect theRect, short top, short left,
                    short bottom, short right);
```

or:

```
void SetRect (Rect, short, short, short, short);
```

Note that in these examples the keyword *void* is used in prototypes and function definitions for empty argument lists and in prototypes and function definitions for null returned values. (This non-use of *void* in function definitions is specific to Symantec.)

Variable Number of Arguments

An ellipsis (...) can be used to specify an unknown number and type of parameters. However, argument checking is turned off when a function is declared to have an unspecified number of arguments. Because of this, it is best not to use this capability unless it is absolutely necessary.

An example of code with a variable number of arguments might be:

Prototype:

```
int printf(char *format, …);
```

Use:

```
printf("%f is sqrt of 4\n", sqrt(4));
```

Note:

The library *stdarg.h* contains a set of macros for accessing unspecified arguments.

Passing Function Arguments

The code between the left and right parentheses in a function is called *arguments* (or *parameters*). When you make a function call, the arguments that you have placed between the parentheses are automatically available to the function. This operation is called *passing arguments*. For example:

```
 main()
{
  float a,b;


  a = 6;
  b = Square(a);
  do more stuff here;
}


float Square(float x)
{
  return(x * x);
}
```

In the above example, *a* is passed in to *square*, and inside of *square* it is referred to as *x*.

You may pass in function arguments by three different methods: value, pointer, and reference. Examples of each method appear below.

It is important to note that under C++ and the new ANSI standard for C, the declaration of the arguments must be included between the parentheses of the function declaration. Previously, the code in C would have been written as follows:

```
float Square(x)
  float x;
{
  return(x * x);
}
```

Do not use the above style in your programming because C++ does not support it.

Passing by Value

```
main()
{
  short a, b;

  a = 5;
  MyFunction(a);
  b = a;
}

MyFunction(short x)
{
  if (s == 5)
  {
  x = 6;
  }
}
```

When this routine is complete, *b* will be equal to 5.

Passing by Pointer

```
main()
{
  short a, b;

  a = 5;
  MyFunction(&a);
  b = a;
}

MyFunction(short *x)
```

```
{
  if (*x == 5)
  {
    *x = 6;
  }
}
```

When this routine is complete, *b* will be equal to 6.

Passing by Reference

```
main()
{
  short a, b;

  a = 5;
  MyFunction(a);
  b = a
{

MyFunction(short &x)
{
  if (x == 5)
  {
     x = 6;
  }
}
```

When this function is complete, *b* is equal to 6.

Default Arguments

A default argument is usually a constant that occurs frequently. By using a default argument, you save writing in a default value at each call.

```
short Exp(short n, short k = 2);

main()
{
  short i, a, b;

  i = 5;
  a = Exp(i + 5, 1);
  b = Exp(i + 5, 3);
}
short Exp(short n, short k = 2)
{
  if (k == 2)
    return (n * n);
  else
    return (Exp (n, k - 1) * n);
}
```

Remember that only trailing arguments may have a default value, as shown in the following code:

```
void foo(long i, long j = 7)                        //legal
void goo(long i = 3, long j)                        //illegal
void hoo(long i, long j = 3, long k = 7)            //legal
void moo(long i = 1, long j = w, long k = 3)        //legal
void noo(long i, long j= 2, long k)                 //illegal
```

Passing Multiple Values

You can also pass in multiple values. For example:

```
main()
{
  short a = 3, b = 4, c;
  c = MyFunction(a, b);
}

short MyFunction(short a, short b)
{
  return(a + b);
}
```

You can pass in any number or combination of variable types.

Explicit Void

You can explicitly ignore the result of a function by placing a *void* typecast in front of the function call. You use an explicit void when you do not care about the returned value. Note the typecast *void* in the following examples:

```
short MyFunction(short &A);          //Prototype
Y = MyFunction(X);                   //Normal
(void) MyFunction(Z);                //Explicit
MyFunction(Z);                       //Implicit
```

The Stack

When you call a function, the address where you need to return is *pushed* into a queue (waiting line) called the *stack*. The stack holds the return address, function arguments, and local variables. The stack is a LIFO; that is, the Last thing that is put Into the queue is the First thing that comes Out. When the function hits a return or comes to an end, it *pops* that address out of the stack and returns to the address of that statement plus one additional statement.

By convention, the stack grows from high to low memory address. When a function has finished executing, the stack consumed by the function is released, restoring the stack to the state it was in before the function was called. In C++, all stack management is automatically done by the compiler.

Figure 6.2 shows how memory is allocated in the stack.

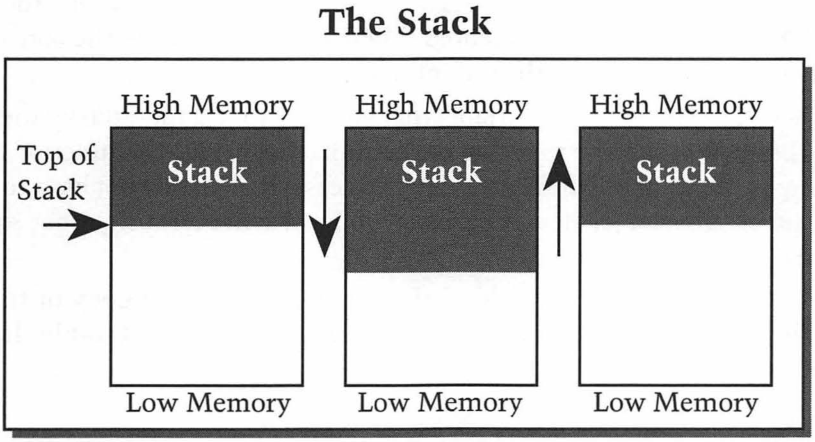


Figure 6.2 *Memory in the stack.*

To see how the stack manages memory in a function call, examine the following code:

```
main()
{
  do stuff here;
  MyFunction();  \\ This is a function call
  do more stuff here;
}

void MyFunction()
{
  do my function's stuff here;
}
```

When the program starts this example, it executes statements (*do stuff here*) in the *main*. Embedded within the statements of the *main* is the call to the function (*MyFunction*). In the process of making that call, the address where you need to return (*do more stuff here*) is pushed on the stack. Next, the statements in MyFunction are executed (*do my function's stuff here*). Note that the line *void MyFunction( )* is called a function declaration. When the end of the function is encountered, the return address is popped off the stack, and the statement (*do more stuff here*) is executed. The handling, and even the concept, of the stack are transparent to the C++ programmer.

You do not have to call a variable that you pass in to a function by the same name that you use in the function declaration. This gives the function a general-purpose capability, which means (in this case) that you do not have to write a routine to square a particular variable; you can write a routine that squares *any* variable.

When you pass in a variable to the function, you get a copy of the value of the variable; you do not get the variable itself. For example, look at the following:

```
main()
{
  short a = 0, b;
  MyFunction(a);
  b = a + 1;
}

void MyFunction(short a)
{
  a = 5;
}
```

First you declare *a* and *b* and set a to *0*. Then you call *MyFunction*, which changes the value of *a*. Then you compute *b* as being the value of *a* + *1*. From a quick examination of the program, it appears that *b* is equal to *6*, and *a* is equal to *5*. However, *a* is actually equal to *0*, and *b* is equal to *1*. The reason is that when we called *MyFunction*, it created a local variable called *a* on the stack. A copy of the value of the *a* argument being passed in from the *main* was placed in that local variable. Then the local variable *a* was set to *5*, but not the *a* that was declared in *main*. When the end of *MyFunction* is reached, any memory created for *MyFunction* on the stack is released. Therefore, the value of the *a* that was set to *5* is now lost.

In order to make the program work, you must do the following:

```
main()
{
  short a = 0, b;
  MyFunction(&a);
  b = a + 1;
}

void MyFunction(short *a)
{
  *a = 5;
}
```

In this example, we passed the address of *a* as an argument to *MyFunction* (which is 4 bytes). Now, in the function declaration, we declared *a* to be a pointer to a *short*. Where **a = 5*, we are saying that the contents of *a* are equal to 5. It works because we passed in the address of *a*, a copy of which was stored as a local variable. In other words, if you want the routine to change a value, you have to pass in the address. (This only applies to arguments being passed in as values.)

The *void* means that *MyFunction* does not return a value. The following is an example of a function that returns a value:

```
main()
{
  float a;

  do stuff here;
  a = GetPi();
  do more stuff here;
}
```

```
float GetPi()
{
   return(3.14);
}
```

Note:

A function that returns nothing is called a *void* function. In some languages, a *void* function is referred to as a procedure, and a function that returns a value is referred to as a function.

Here, the function returns the value of pi; that is, the function returns a *float.*

The returned value is not placed on the stack; it is stored in the 68xxx microprocessor's registers. The register (D0) is only 4 bytes long, so the value of whatever is returned cannot be more than 4 bytes. (See the subsection entitled Register Variables later in this chapter.)

The Heap

The heap, which is located at the low end of memory, contains quite a variety of data objects. The system heap, which you will not use in your programming, contains the Operating System code, INITS, fonts, DAs, and other management data that are part of the Mac environment. The application heap, which is the one that you will use, contains your application resources, including the code segments of your applications. Among other things, it is used for dynamic memory allocation. The free store operators *new* and *delete* act on the heap. Figure 6.3 shows how memory is allocated to the heap.

Remember that you have to allocate a block of memory in the heap before you can use it, and only one application can use a block of the heap at any given time. After you have finished with the block, you deallocate it so that another part of your program can then use it.

C and Pascal on the Macintosh

C programmers must have some knowledge of Pascal, specifically in the areas of procedures, functions, and parameters. This is because the Mac is a native speaker of Pascal. All of the Mac ROM (read-only memory) routines are defined as if they were being called from Pascal, so users of other languages must compensate. Note the following declarations of toolbox routines in Pascal:

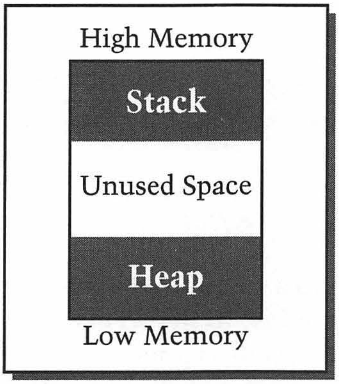


Figure 6.3 *Memory in the heap.*

```
PROCEDURE FrameRoundRect
  (r: Rect; ovalWidth, ovalHeight: INTEGER);
FUNCTION StringWidth(s: Str255): INTEGER;
```

Procedures and Functions

Pascal has two kinds of subroutines: procedures and functions. In C, every subroutine is a function, and a *void* function is essentially a procedure, as shown in the code below:

Pascal declaration:

```
PROCEDURE FrameRoundRect
  (r: Rect; ovalWidth, ovalHeight: INTEGER);
```

C++ equivalent:

```
pascal void FrameRoundRect
  (const Rect *r, short
  ovalWidth, ovalHeight);
```

Pascal declaration:

```
FUNCTION StringWidth(s: Str255): INTEGER;
```

C++ equivalent:

```
typedef const unsigned char *ConstStr255Param;
pascal short StringWidth(ConstStr255Param s);
```

Order of Parameters Pushed

Pascal pushes parameters to a subroutine in order from first to last. C pushes them in reverse order, from last to first. This allows C to support a variable number of arguments and default arguments. When calling a Pascal function from C, you must push parameters in Pascal order. Figure 6.4 shows parameter orders in Pascal and C stacks.

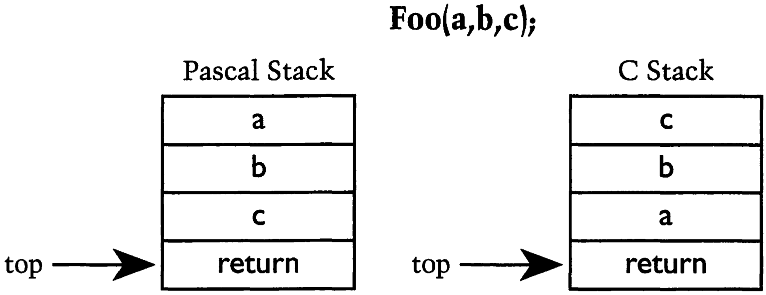


Figure 6.4 *Order of parameters pushed in Pascal and C++.*

Used in function declarations (i.e., prototypes), the *Pascal* keyword tells the compiler to push parameters in forward order just as Pascal would. Used in function definitions, the *Pascal* keyword tells the compiler to expect its parameters in forward order, as shown in the following code:

```
pascal void ScrollUp(ControlHandle theControl,
                     Int16 thePart)
{
  Int16 startingValue;
  if (thePart == inUpButton)
  {
  startingValue = GetCtlValue(theControl);
  ... // more code here
```

```
        }
    }
```

Type Conversion

At the machine level, all data types in memory meld into a contiguous stream of bits carrying types of information that represent a kind of prescription: Take *x* number of bits and interpret them using the following pattern.

Converting from one type to another will change the type but not the underlying bit pattern. The size of the new type may be wider or narrower, and the interpretation of the bits will change. Some type conversions are not safe; for example, it is not safe to convert from a wider data type to a narrower one or vice versa. Note the following inconsistencies:

```
    float fval = 3.14159;
    double dval;
    dval = double(fval);
```

This example requires bits beyond the size of a *float*.

```
    unsigned char ucval = 255;
    char cval;
    cval = char(ucval);
```

Here, the interpretation of the bits changes.

```
    short sval;
    sval = 3.14159;
```

In this example, the fractional part is lost.

Variable Storage Types

Variables store data in the form of characters, numbers, strings, pointers, and data structures. This section covers five variable storage types: automatic, static, external, register, and const.

"auto" Variables

In C++, local variables are known as *auto* (automatic) *variables* because C++ automatically creates memory for them on the stack each time the function is entered. However, that space is removed from the stack after the function is executed.

You can put the word *auto* in the declaration, but it is pointless to do so. All local variables that you might declare are *auto* by default. Note the following code:

```
main()
{
   short a = 0;
   MyFunction(a);
   MyFunction(a);
}

void MyFunction(short &a)
{
   auto short b;

   if (a == 0)
   {
      b = 0;
   }
   a = a + 1;
   b = b + 10;
}
```

The second time *MyFunction* is called, the variable *b* is garbage.

Static Variables

Static variables, which can be internal or external, are another type of storage. An internal static variable is local to a particular function, just as an automatic (local) variable is. Unlike an automatic, it remains in existence in permanent data storage rather than coming and going each time the function is called. An example of a function with a static variable might be:

```
main()
{
  short a = 0;
  MyFunction(&a);
  MyFunction(&a);
}

void MyFunction(short *a)
{
  static short b;
  if (*a == 0)
  {
    b = 0;
  }
  *a = *a + 1;
  b = b + 10;
}
```

In this example, we have set *a* to *0* and have given *MyFunction* the address of *a*. The first time we call *MyFunction*, we pass in the address of *a*. Here, if the contents of *a* are equal to *0*, which is true in this case, we set *b* to *0*. Then we bump the contents of *a* by *1*, so that the contents of *a* are equal to *1*, and we bump *b* to *10*. The second time we call this function, the variable *a* is now equal to *1* and we have failed the *if* test; we do not set *b* to *0*. When the program is finished (i.e., after the second call is made to *MyFunction*), *a* will be equal to *2*, and *b* will be equal to *20*. The significance is that the value of *b* is retained because it is stored in the private *data* area; it is not popped off the stack.

External Variables

You can declare a variable to be available to every function in your program by making it global or *external*. For instance:

Note:

External variables are stored in the data area, not on the stack.

```
short b;
main()
{
    short a = 0;
    MyFunction(a);
    MyFunction(a);
  }

  void MyFunction(short &a)
  {
    external short b;
    if (a == 0)
    {
      b = 0;
    }
    a = a + 1;
    b = b + 10;
  }
```

By declaring *short b* outside of any function block, you make it external and, therefore, accessible to any function that follows. In the example above, both the *main* and *MyFunction* know of the existence of *b*. If you moved the declaration between the *main* and *MyFunction*, only *MyFunction* would know about *b*. When you place the declaration of a variable inside a function block, the existence of that variable is known only to the function itself; that is, it is local to that function. However, you can place the variable inside the function block and make it global by inserting the word *external* before the variable. The external declaration in *MyFunction* is required only if *MyFunction* is declared before *b*.

Register Variables

Register variables offer a fourth class of storage. When you declare a variable a *register variable*, you ask the compiler, whenever possible, to store that variable in a register. You may want the variable put into a register because you will be using it frequently, and a CPU does its fastest computations on variables that are in registers. However, the compiler may not always be able to store the variable in a register for two reasons: (1) There are only eight 4-byte data registers in a 68xxx CPU, and these may already be in use; and (2) the variable that you want to store is greater than 4 bytes. Generally, it is best to avoid declaring register variables.

Figure 6.5 shows the user's registers in a 68xxx microprocessor. Typical register usage in C++ is as follows:

A7	Stack pointer (SP)
A6	Pointer to function's locals (base register)
A5	Pointer to application globals
A4	Pointer to driver or code resource globals
D0	Return value from function

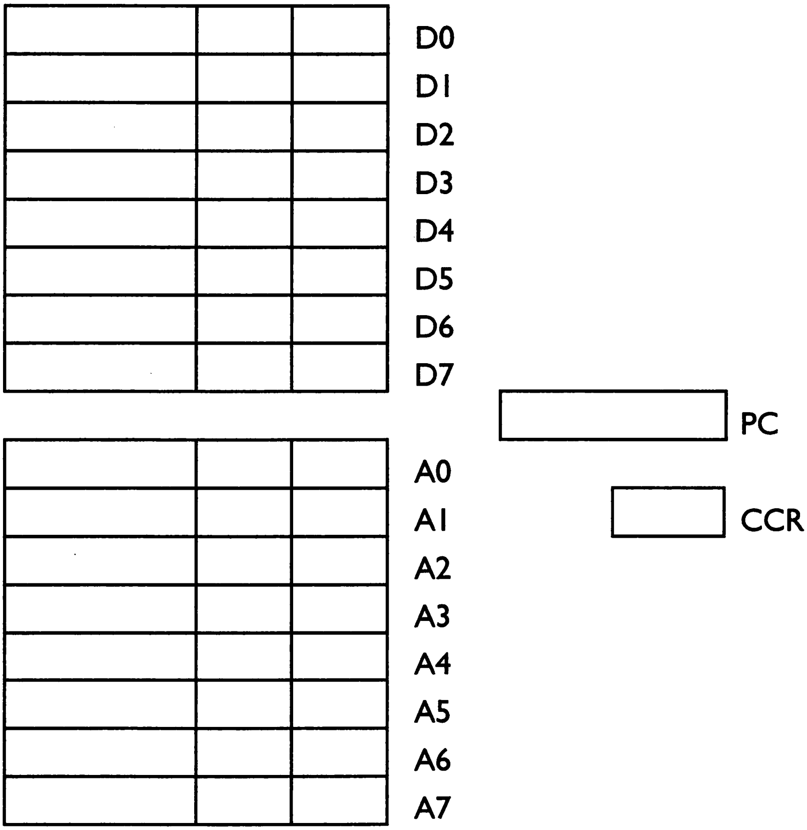


Figure 6.5 *68xxx user's registers.*

WARNING

Compilers are better able to optimize now than ever before. Therefore, if you force the compiler to store a variable in a data register, you may take away that optimization. It may be best to leave the choice to the discretion of the compiler.

A0–A1 and D0–D2 are trashable registers; that is, they are not guaranteed to remain the same after a ROM call. A2–A7 and D3–D7 are protected registers; that is, they are never corrupted or changed by the action of a ROM routine. That leaves only three address registers and five data registers available for register variables. Even so, it is not likely that the compiler could maintain more than one address register and two or three data registers for register variables.

"const" Variables

The *const* keyword is a type specifier. When used alone, it implies an *int* type. Any variable declared a *const* cannot be changed.

If you do not initialize a *const*, you will get a compile-time error. The same will happen if you try to assign the address of a *const*.

An example of a *const* declaration is:

```
const false = 0;
const double pi = 3.14;

ComputeArea(float radius, const float pi);
```

You may declare a pointer to the address of a *const*, but the pointer itself is not a *const*. The pointer can be changed to address a different variable of the same type at any time, but the contents of the pointer cannot be modified through the pointer. Note the following code:

```
double x;
const double *pc;    //OK
...
*pc = &x;            //OK
```

You can define a pointer that is a *const*. You can also define a *const* pointer to a *const*, as shown below:

```
short i = 10;
short *const cpi = &i; //Constant pointer to short
```

```
const short j = 20;
const short *const cpj = &j;
```

Note:

A literal string is a *char, not a *const char**.**

Reference Declarations

Reference declarations provide a way to have a multiple number of names refer to the same object. Modifying one is the same as modifying any other. As is the case with all variables, reference variables must be initialized. A reference type is sometimes referred to as an alias. The format for reference declarations is shown in the following example.

```
short &theAlias = theName;


unsigned char theString[256];
unsigned char &length = theString[0];
unsigned char &last = theString[255];
```

Another example might be:

```
short val = 10;
short &refVal = val;
short *pVal = &refVal;


if (*pVal == refVal && pVal == &refVal)
{
```

Right–Left Rule

The right–left rule provides a method for you to see how and in what order a function operates. Here is how it works:

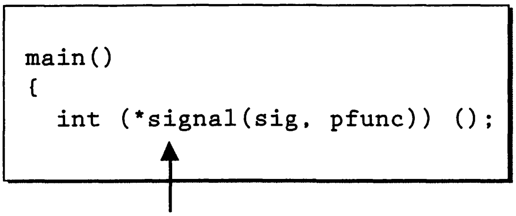
1. Start with the identifier.
2. Look to the right for an attribute.
3. If none is found, look to the left.
4. If found, substitute an English keyword.
5. Continue right–left substitutions as you work your way out.
6. Stop when you reach the data type in the declaration.

English Keywords

( )	Function returns
[n]	Array of *n*
*	Pointer to
&	Reference to

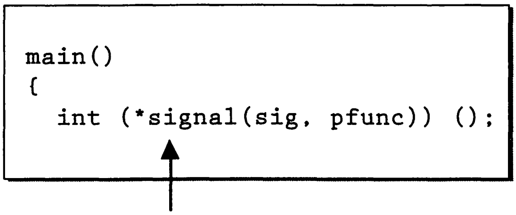
Now, let's look at the following walkthrough of the right-left rule:

Signal is a...



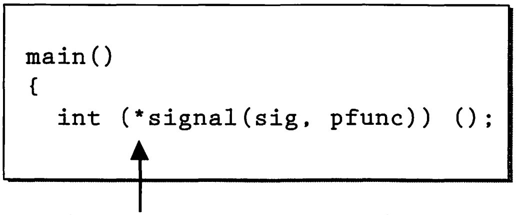
```
main()
{
  int (*signal(sig, pfunc)) ();
```

Signal is a function that returns...



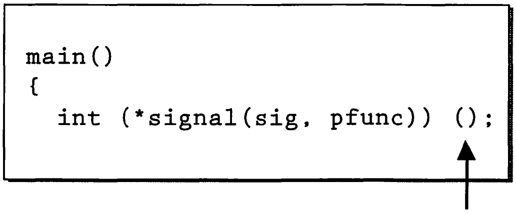
```
main()
{
  int (*signal(sig, pfunc)) ();
```

Signal is a function that returns a pointer to a ...



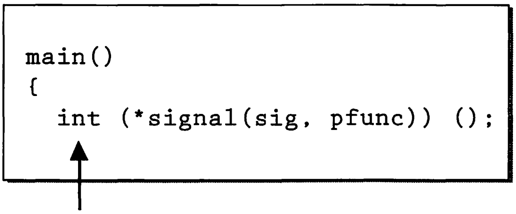
```
main()
{
  int (*signal(sig, pfunc)) ();
```

Signal is a function that returns a pointer to a function that returns...



```
main()
{
  int (*signal(sig, pfunc)) ();
```

Signal is a function that returns a pointer to a function that returns an *int.*



```
main()
{
  int (*signal(sig, pfunc)) ();
```

Another example might be:

```
long *p[2];
```

Here, *p* is an array of two *pointers* to a *long.*

Function Overloading

In C++, it is possible to overload functions; that is, more than one function within the same program can be given the same name. The correct one will automatically be called during the execution of the program. Use of function overloading can make a program more readable. The following code makes ample use of the function-overloading capability:

```
main()
{
  short a, b, c;
  short sum;

  a = 2; b = 3; c = 4;
  sum = add (a, b);
  sum = add (a,b,c);
}
short add(short a, short b)
{
  return (a + b);
}
short add(short a, short b, short c)
{
  return (a + b + c);
}
```

In this example, the number and the type of arguments determine which function gets called. The reserve word *overload* could be placed in front of each function declaration that is overloaded, but it is not required or recommended.

As mentioned before, the correct function to be invoked is determined by the type and number of arguments that are being passed to the function by the call. The return value—if any—is not taken into account. All functions overloaded with the same name should have the same return type.

Scope Resolution Operator

In C++, a function can declare an automatic (local) variable that has the same name as a global variable. It is important to note that in that function, the *local* variable will be referenced, not the *global.* If you want to access the global variable, you can do this by using the scope resolution operator (::). Note the following code:

```
short sameName = 5;


main()
{
  MyFunction();
}
void MyFunction()
{
  short theValue;
  short sameName = 4;

  theValue = sameName * 2            //local variable
  theValue = ::sameName * 2;         //global variable
}
```

Inline Functions

Every call to a function slows the execution of your program to some extent. Functions that are invoked many times may be placed *inline,* avoiding the overhead of a function call. The penalty you pay is that your program consumes more memory. You simply call an *inline* function like you would any other function, as shown in the following example:

```
inline char LoByte(short x)
{
  return (x & 0x00FF);
}
```

or:

```
inline char HiByte(short x)
{
  return ((x >> 8) & 0x00FF);
}
```

C++ Preprocessor

A preprocessor operates on your C++ source code before presenting it to the compiler. The preprocessor looks for a set of keywords that begin with the oglethorpe (#) symbol. The following list shows the preprocessor statements:

#include
#define
#if
#else
#endif
#ifdef
#ifndef

Files may be read into your source code with the *#include.*Files enclosed in " " are read in from your current folder, and files enclosed in <> are read in from a specified folder. For example:

```
#include "MyFile.h"
#include <TheirFile.h>
```

The *if, else, endif, ifdef,* and *ifndef* are used for conditional compiles. The format for this is:

```
#ifdef THINK_C
  do this code
#else
  do this code for everyone else
#endif
```

Conditional Directives

Conditional directives can be used to guard against the multiple processing of a header file. For instance:

```
#ifndef _MyHeader_
#define _MyHeader_
.
.
.
  MyHeader.h contents go here
.
.
.
#endif
```

C++ Preprocessor Examples:

The *define statement* implements macros and supports arguments in C++. It can also be used to define constants. Examples of macros that can be useful in Toolbox programming follow:

```
#define SetPt(pt,hor,vert) {(pt)->h = (hor);\
  (pt)->v (vert);}
#define SetRect(rect, l, t, r, b)\
  {(rect)->top = (t); (rect)->left = (l);\
   (rect)->bottom = (b); (rect)->right = (r);}
#define SetRGBColor(rgb,r,g,b)\
  {(rgb)->red = (r); (rgb)->green = (g); (rgb)->blue = (b);}
#define RectWidth(rect) ((rect)->right - (rect)->left)
#define RectHeight(rect) ((rect)->bottom - (rect)->top)
#define abs(x) ((x)<0?-(x):(x))
#define min(x,y) ((x)<(y)?(x):(y))
#define max(x,y) ((x)<(y)?(y):(x))
```

```
#define HiByte(x) ((x) >> 8) & 0x00FF)
#define LoByte(x) ((x) & 0x00FF)
#define Swap (x,y) ((x)^=(y)^=(x)^=(y))
#define arraySize(x) (sizeof(x) / sizeof
((x)[0]))
#define infinity ;;
```

Note:

In the preprocessor, the backslash (\) allows you to continue a macro expression to a new line. In C, macros had to be on one line only.

#define versus const and inline

The advantage of defining *inline* functions and *const* definitions rather than using the *#define* statement is that the C++ compiler can check the same code you see for errors. The disadvantage of defining *inline* functions is that it takes more work to support arguments of various types. Note the following code:

```
short abs(short x)
{
  return (x < 0 ? -x : x);
}
long abs(long x)
{
  return (x < 0 ? -x : x);
}
float abs(float x)
{
  return (x < 0 ? -x : x);
}
double abs(double x)
{
  return (x < 0 ? -x : x);
}
```

The Preprocessor and Comments

Other C++ compilers may not recognize the single-line preprocessor comment, which is:

```
#define pi 3.14  // pi is π
```

Symantec C++ does recognize the double-slash preprocessor comment. If you use the double slash, just be aware that it may not be compatible with other compilers when you attempt to port your code.

Summary

The features discussed in this chapter were:

- Using functions in C++.
- Function prototypes.
- The program stack.
- Pascal functions.
- Variable storage types.
- The right–left rule.
- Function overloading.
- Rules of scope.
- Inline functions and preprocessor statements.

Exercises

1) Recode the following using inline declarations:

    a) `#define Min(a,b) ((a)<(b)?(a):(b))`

    b) `#define Max(a,b) ((a)<(b)?(b):(a))`

    c) `#define Abs(a) ((a)<0?-(a):(a))`

    d) `#define HiByte(x) (((x) >> 8) & 0x00FF)`

    e) `#define LoByte(x) ((x) & 0x00FF)`

    f) `#define cube(x) (x) * (x) * (x)`

    g) `#define arraySize(a) (sizeof(a) / sizeof ((a)[0]))`

2) Using the right-left rule, explain the following:
   a) `char (*(*p)) () [10];`
   b) `char *(*p) () [10];`
   c) `char **p () [10];`
   d) `char* **p () [10];`
   e) `char* *(**p) () [10];`

3) Given the following function:

```
void Swap (short *x, short *y)
{
  short temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

Rewrite the function using call-by-reference.

4) Create a header file. Place in the header file the following:
   a) macros or consts that you feel that you will commonly use.
   b) your favorite inline functions.
   c) the necessary code to make sure that your header file will not generate an error if it is included more than once.

5) Rewrite the following as an inline function. Overload the function to support shorts and longs.

```
#define swap(x,y) short t; t = x; x = y; y = t;
```

7 Input/Output Streams

The simplest examples of input/output devices on your computer are the keyboard (input) and the screen (output). You use these devices to get information into and out of the computer in the same way that you use pens and books. These are easy concepts to understand when you are just a computer user. What is not quite as easy to comprehend is the way in which the computer handles your input and output when you write a program.

Neither C nor C++ contains any predefined input and output operators. Both support the infamous *stdio* (standard input/output) library, and C++ supports a new library called *iostream*. The iostream library is far more robust than the stdio library. You should use the iostream library for any new projects, because C++ will not support the stdio library in the future.

Streams

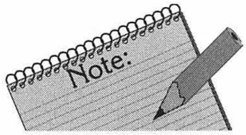
A stream is a sequence of bytes. You can extract data from a stream and place it into a program variable with the *extraction operator* (>>). Conversely, you can inset data into the stream with the *insertion operator* (<<).

I/O Channels

The name of the input channel associated with the user's keyboard is called *cin* (standard input). The output associated with the computer screen is called *cout* (standard output). Error statements may go to the user's screen or to an error file, which is called *cerr* (standard error). An example of a string going to an output channel might be:

```
cout << "This goes to the output channel \n";
```

You may have noticed that the output line includes the character sequence *\n*. This is an escape sequence of control characters that instructs C++ to move to a new line. This will be a common resident in the source code of your programs. Other escape sequences are as follows.



The term *escape sequence* refers to escaping from a string and going into another mode. The backslash represents the escape mechanism and the character after the backslash determines the action taken.

Escape Sequences

\\	Backslash
\?	Question mark
\a	Sound bell
\b	Backspace
\f	Formfeed (new page)
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quote character
\"	Double quote character
\0	Null

An example of an escape sequence might be:

```
cout << "\a";
cout << "\nError - press any key to continue\a\n";
```

The first statement sounds the system beep. The second statement moves to a new line, displays a warning on the screen, sounds the beep, and then moves onto another line.

Predefined I/O Stream Manipulator

The term *endl* insets a new line character into the output stream and then flushes the output buffer. An example might be:

```
cout << "\n";
```

use instead

```
cout << endl;
```

Generalized Escape Sequence

The format of the generalized escape sequence is *\000,* where *000* represents a sequence of up to three octal digits, as shown in the following code:

```
\7        //bell
\0        //null
\12       //newline
\062      //'2'
```

The format for a hex escape sequence is *\xhh,* where *hh* represents any number of hex digits. An example is:

```
\x7       //bell
\x0       //null
\x0a      //new line
\x32      //'2'
```

Formatted Output

C++ provides the capability for you to alter the format of the data you want to display from an unformatted output to a formatted one. There are five simple formatted output functions: *chr( )*, *dec( )*, *oct( )*, *hex( )*, and *str( )*.

WARNING

Not all C++ compilers support hex escape sequences.

Characters:

```
char letter;
letter = 'a';
cout << letter;                // a or 97?

cout << chr(letter);           //will output a
```

In the above example, the format *cout << letter* would most likely output *97*. By using the format *chr(letter)*, you ensure that the output will be *a*.

Decimal Numbers:

```
float number;
number = 12.345;
cout << dec(number);           // output will be 12

cout << dec(number, 20);       // twenty chars wide
```

In the above example, the number will be right-justified (to 20 places).

Hex and Octal Numbers:

```
short number;
number = 16;
cout << hex(number);           // 10
cout << oct(number);           // 20
```

Here, if you added the statement *cout << dec(number)*, you would get the decimal number *16* as the output.

Strings:

```
cout << str(string);
cout << str(string, 30);       // 30 characters
```

When the number of characters for the field is omitted or equal to zero, the correct amount of space required to display the contents of the variable will be allocated automatically. If the number allocated for the field is too small, the output will be truncated without any warning. If the number is negative, the output will be left-justified.

Note:

The strings discussed above are C strings, not Pascal strings.

Advanced Formatting

A more complex formatted output is available through the *form( )* function. This function allows formatting sequences called *conversion commands*. An example might be:

```
short number = 123;
cout << form("%x%s", number, "\n"); //hex
cout << form("%o%s", number, "\n");  //octal
cout << form("%d%s", number, "\n");  //decimal
```

As shown in this example, you must place the appropriate formatting sequence or sequences between double quotation marks before the values to which they refer.

The conversion commands are as follows:

%c	Single character
%d	Decimal integer
%e	Scientific notation
%f	Floating point value
%g	General numerical format
%o	Octal integer
%p	Pointer value
%s	String
%u	Unsigned integer
%x	Hexadecimal integer

The following code is another example of advanced formatting:

```
float fraction;
fraction = 123.456;
cout << form("%3.2f%s", fraction, "\n");
```

In the last line of the example above, the numbers immediately before and after the decimal point specify the number of digits before and after the decimal point, respectively.

Input Stream Example

With the function *cin* and the operator >>, you can input variables of any type in sequence, as follows:

```
float fraction;
char letter;
short number;

cin >> fraction >> number >> letter;
```

However, *cin* does have a disadvantage, as you'll see in the examples below.

If your input was:

```
This is a string
char str[80];
cin >> str;
```

the input variable will contain:

```
str is "This"
```

If your input was:

```
This is a string
char str1[80], str2[80], str3[80], str4[80];
cin >> str1, str2, str3, str4;
```

The input variables will contain:

```
str1 is "This"
str2 is "is"
str3 is "a"
str4 is "string"
```

The variable *string* will hold up to 80 characters, so you might think that by assigning the string 80 characters you would have enough field width to print "This is a string." The reason that only the word "This" will be assigned to the string is that *cin* recognizes a space character as the end of the variable.

I/O on the Macintosh

Almost all input and output on the Mac should be accomplished through the Toolbox or a class library. The C++ I/O can be useful for debugging, but the use of a debugger is more efficient, and you don't risk the danger of leaving *cins* and *couts* in your code.

Summary

In this chapter, we've covered:

- Using stream operators.
- What I/O channels are.
- Escape characters.
- Formatted output.

The next chapter, Chapter 8, is on advanced data structures. It is in this chapter that you will begin to see firmly the relationship between data structures and member functions in object-oriented programming.

Exercises

1) Write a program that will prompt a user for a temperature in Fahrenheit or Celsius, then display the temperature in both scales.

2) Write a program that will accept a book title, author, publisher and copyright date, then display all of the information.

8 Advanced Data Structures

Computers only know about bits, period. But how those bits are interpreted (and what is done with them) is accomplished through data structures. C++ differs from C (and Pascal) in that it allows structures to contain member functions as well as data, and these member functions manipulate the data contained in those structures.

To understand the function of structures fully, you must comprehend pointers and arrays and know something of dynamic memory allocation. This chapter covers those topics, then moves on to enumerating variables, structures, and unions; operator overloading (as opposed to function overloading); and encapsulation.

Pointers

A pointer is a variable that contains the address (memory location) of another variable. Use of a pointer is called *indirection* because the pointer is getting information indirectly. Getting the contents of a pointer is called *dereferencing*.

The size of the pointer has to be large enough to contain the address for a particular machine. For instance, a pointer on a Cray would have to be 8 bytes (a 64-bit word), but on the Macintosh, a pointer is 4 bytes. Figure 8.1 illustrates a pointer to memory.

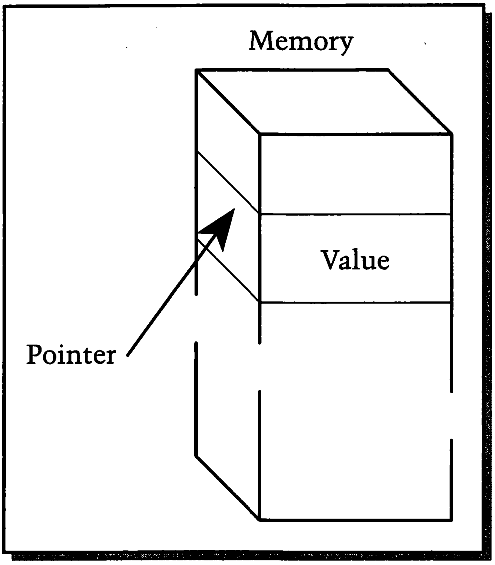


Figure 8.1 *Pointer to an address in memory.*

If you want to declare something to be a pointer you, might say:

```
short *p;
```

Using the right–left rule, this example declares that *p* is a pointer to a *short*. The only memory that is assigned here is the 4 bytes for the pointer, not the memory that it is pointing to. Right now, the *p* is pointing to garbage.

If you wanted to assign an address to *p*, you would write:

```
short *p;
short a;
```

```
p = &a;
```

Here, we declared a pointer to a *short*, we declared the *short*, and we set the pointer to the address of the *short*. As you can see, *p* now points to a.

Another example of using a pointer in code might be:

```
long a, b, *p;

p = &a;

*p = 10L;      //same
a = 10L;

b = *p:        //same
b = a;
```

Initializing a Pointer

There are three ways to initialize a pointer:

By variable address:

```
short value = 55;
short *p1;

p1 = &value;
```

From another pointer:

```
short value = 55;
short *p1, *p2;

p1 = &value;
p2 = p1;
```

WARNING

Pointers direct you to an area in memory. You must make sure that those areas are safe to use; that is, they are not used for some other purpose for which you have no knowledge. In all the previous examples, we've declared a pointer and something that it will point to. Then we did the following assignment:

```
p = &a;
```

This is legal. However, a very dangerous use of a pointer is:

```
short *p;
*p = 10;
```

In this case, *p* has not been initialized and can therefore be pointing anywhere in memory. At the memory location of *p* we assign the value of *10*, thereby overriding anything else that may be in that memory location. That memory location may hold part of your executable code, an I/O device, or anything else imaginable.

By using *new* operator:

```
short *p1;


p1 = new short;
*p1 = 55;
```

Note: It is illegal to take the address of a register variable; the compiler will not allow it.

Void Pointers

Void pointers can be used to point to variables of any type. The only way a void pointer can be initialized is by setting its value from another pointer. To dereference a void pointer, you must cast it first.

```
short value1 = 55, value2, *p1;
void *vp1;


p1 = &value1;
vp1 = p1;            //Both point to the same place
value2 = *short(vp1);
```

In the above example, we declared *value1* equal to *55*, *value2* as just a *short*, and **p1* as a pointer to a *short*. Then we declared **vp1* as a pointer to *void*, which is of unknown origin. Now we say that *p1* is equal to the address of *value*, and *vp1* is equal to *p1*. Now *vp1* and *p1* both contain the same address. The problem is with the interpretation: *p1* will always be interpreted as a pointer to a *short*, and *vp1* is a generic pointer. It points to memory, but at the same time it points to *void*. (It knows not what it points to!) There may be some reason for using this void pointer—to change the value from a *short* to a *long*, for instance—but the only way to use it is to cast it. This is done with the expression *value2 = *short(vp1)*. Casting takes precedence in the order of operations here; that is, the cast *vp1* is converted to a *short*, whose contents are then placed in *value2*. (Remember that the right–left rule applies only to definitions. It does not apply here.)

Arrays

An array is an accumulation of memory set aside for like-kind variables. For example, you can have an array of 100 *chars* or any other valid variable type. To declare an array, you might write:

```
    char a[100];
```

This declaration reserves 100 *chars* in memory for your use. If you want to access one of the *chars* in that array, you would write:

```
    a[10] = 5;
```

In this example, *a* will be indexed by the number *10*, and in that number *10* slot, you set the value *5*. From our discussion of the bracket operators, another way of stating the line *a[10] = 5* would be:

```
    *(a + 10 ) = 5;
```

Here, you take the *a*, which is an address or pointer, and add an offset to it. The offset is *10*, which is the index. This is automatically multiplied by the size of a *char*. Then you take the contents of the combination of the address and offset and put a *5* in that memory location. As you can see, an array is much more concise. (See Figure 8.2.)

In the first array example, when we declared *a*, we set the size of *a* to *100*. This is known as *dimensioning a*. We dimensioned *a* to the size of *100*, but we can only index *a* from *0* to *99*. To get to the very first element in *a*, we must write *a[0]*, which is the same thing as saying **(a + 0)* or, simply, **a*. To get to the last element in *a*, we write *a[99]*, which is the same thing as saying **(a + 99)*.

Note:

An example of the use of pointers can be found in the passing of parameters to a function. See the section called *Passing Function Arguments* in Chapter 6.

WARNING

If you index the array by any number smaller than 0 or greater than 99, you will be addressing a memory location outside of the area that was reserved for the array. C++ does not have array bounds checking, which means that there are no safeguards to prevent an array from being overwritten. If your program has many variables, you take a chance that the exclusive storage allocated to these other variables may be overwritten by the excess characters in the array. As you can see, this is fraught with the same dangers as using a pointer indiscriminately.

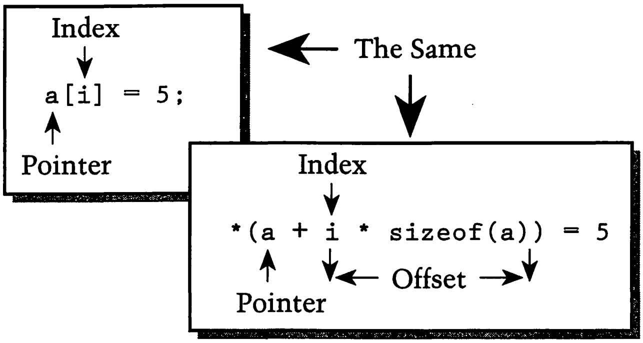
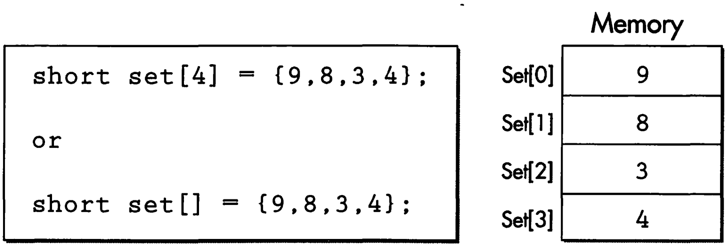


Figure 8.2 *Ease of Using an array.*

Initializing Array Values

Values can be assigned to arrays by listing the values inside curly braces separated by commas, as shown in Figure 8.3.



```
short set[4] = {9,8,3,4};

or

short set[] = {9,8,3,4};
```

	Memory
Set[0]	9
Set[1]	8
Set[2]	3
Set[3]	4

Figure 8.3 *Initializing array values.*

Initializing String Arrays

You can initialize a string array by putting brackets after the type and name and then declaring the value. For instance:

```
char string[6] = "Hello";
```

or:

```
char string[] = "Hello";
```

In the first example, we declared and initialized an array string of size 6. "Hello" is only five characters, but the extra array element is required for the null character used to terminate C++ strings. In the second example, where we have not declared an array size, the array will automatically be set to 6 because it is equated to a string literal.

Indexing Arrays

The definition of an array contains the number of elements in the array. The index of the following array is from *0* to *9*, the array size:

```
short a[10]; //10 is the array size

a[0] = 1;    //[0] is the array index, and 1 is the
             //value
.
.
a[9] = 10;
```

The *off by one error* is a common error in C++ arrays. Just remember that your index number will always be one behind your element number.

Array Assignment

C++ does not allow you to initialize or assign an array with any other, as shown in the following example:

```
short a = {1, 2, 3, 4};
short b[4];

b = a;       //error
```

Also, C++ does not provide any compile-time or run-time range checking of the array. The compiler would allow the following code:

```
short a[10];
```

```
for(short i = 0; i < 100; i++)
{
   a[i] = 0;
}
```

Here, you have declared an array size of *10,* but your code allows for *100* elements. As the program goes through and executes the code, it will write over anything in memory that is in the way. It may, in fact, clobber some of your code. The point here is to be careful that your code matches the size of your array.

Multidimensional Arrays

Multidimensional arrays, which can be useful for scientific and graphics work, are an extended feature in C++. Such an array declared as a formal argument must specify the size of all its dimensions beyond the first one. Following are examples of multidimensional arrays:

```
float large[10][10][10][10];          //4 dimensions
b = sizeof(large);                    //10000
```

Here, the four-dimensional array will have *10,000* elements; that is, *10 x 10 x 10 x 10 = 10,000.*

```
short a[4][3] = {
      {0,  1,  2},
      {3,  4,  5},
      {6,  7,  8},
      {9, 10, 11}
};
```

or:

```
short a[4][3] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

Free Store Operators

Free store operators are used to create and destroy variables in memory. This operation is called dynamic memory allocation. Most other variables are created on the stack, but free store operators create memory for variables on the heap (free store). You create this heap variable with the *new* operator, which returns a pointer to the beginning of the memory allocated to the variable. For instance:

```
char *word = new char[20];
```

Here, we have reserved *20 chars* on the heap pointed to by *word,* which is both an array name and a pointer.

Testing for Space

If you do not have enough memory to create the variable(s) you asked for, *new* will return a *null (0L)* pointer, as shown below:

```
char *word = new char[20];
if (!word) error
```

The last line in the example above means simply, "If word is not valid, then error."

Destroying Heap Variables

A heap variable can be destroyed by using the *delete* operator. The space consumed by the variable will be returned to the heap. To destroy the variable in the previous example, you would write:

```
delete [20] word;
```

or:

```
delete word;
```

Deleting a null pointer—that is, one with a 0 value—is always safe because it does nothing. You can say:

```
delete [20] word;
word = 0L;
```

Here, you have deleted *word* and then set its value to *0*. Just after the delete, the variable *word* will still have the value pointing to memory. Something else may now be there, but *word* will still have that address. However, by setting the value to *0*, if you at some future time need to delete *word* again, it will be okay. You will not have the horrendous error that you would otherwise have if you attempted to delete the same memory twice.

WARNING

For every *new*, there should be a corresponding *delete*. If you attempt to delete a variable that has already been deleted, the error will not be detected by the compiler, but the bug will show up later on. The *new* and *delete* operators are replacements for *malloc( )* and *free( )*.

There are other dangers in using the *new* and *delete* operators. Look at the following code:

```
main()
{
  for (short i = 1; i <= 10; i++)
  {
    MyFunction();
  }
}

void MyFunction()
{
  char *word;
  word = new char [256];
}
```

In the above example, the function *MyFunction* creates a pointer on the stack and then returns to the *main*, thus destroying the pointer. There is no way to go back and delete this memory. More danger:

```
short *x = new short;
short *y = new short;

*x = 26;
*y = 32;
```

```
y = x;         //clobbers old y
*x = 97;
```

Here, the pointer's value is overwritten, and there is no way to go back and delete its associated memory.

When to Use Dynamic Memory Allocation

It is best to treat variables containing user data as dynamic (heap) variables. This is especially true if you cannot foretell the amount of space they will require. Variables that you need for housekeeping while the program is running—loop counters, flags, and so on—should be on the stack.

Enumerated Values

The *enum* statement creates constants, which are assigned a numerical value starting with *0*. Enumerators differ from *const* declarations in that there is no addressable storage associated with an enumerator. For this reason, it is an error to apply the *address-of* (&) operator to an enumerator.

You declare an enumeration with the *enum* keyword and a comma-separated list of enumerators enclosed in curly braces. An example of an *enum* statement might be:

```
enum
{
  false,
  true
};
result = false;
result = 0       //same as above
```

The *enum* constants can be assigned numerical values starting with *0*. You can also force the values, as shown in the following code:

```
enum
{
  simm1 = 1,
    simm2,
    simm4 = 4
};
memory = simm2;
```

Here, instead of *simm1* being equal to *0*, we have forced it to be equal to *1*. The incrementing starts after that.

By the way, the *simms* in the example above refer to the memory cards that you have in your Mac.

Enumerated Variables

We stated above that you can assign numeric values to enumerated constants. However, if you assign any value other than the enumerated constants to an enumerated variable you will get a compile-time error. Look at the following example:

```
short hop = 4;
enum
{
    jump,
    run,
    skip
} simonSez;

simonSez = skip;     //OK
simonSez = hop;      //Compile error
```

In this example, we have declared a variable—*hop*—which is a *short* and which we have set equal to *4*. Next, we have enumerated type with *jump*, *run*, and *skip* and a variable called *simonSez*. We can assign *simonSez* a value of the enumerated type, but we cannot assign it any other value, such as *hop*.

Enumerated Types

In addition to the "built-in" data types such as *char, short,* and *long,* C++ allows you to create *enumerated types.* For instance:

```
enum TrafficLight
{
  red,
  yellow,
  green
} MarketAndGeary, MissionAnd5th;

MarketAndGeary = red;
MissionAnd5th = green;
```

Here, the variables *MarketAndGeary* and *MissionAnd5th* are created within the *enum.* These variables are known as *anonymous* variables because no new type has been defined.

"typedef"

C++ allows you to create your own variable types, which you use in the same manner as built-in types, with the term *typedef.* For example:

```
typedef Byte char;
typedef Boolean char;
typedef Integer int;

Boolean flag;
Byte status;
Integer i;
```

The expression *typedef* can serve as a program documentation aid. It can be used to reduce the notational complexity of a declaration and to improve the readability of definitions of pointers to functions and class member functions. A typical example might be:

```
typedef float wages;

wages johnsPay, marysPay;
```

The terms in the above example tell you exactly what you're going to get. When you use *typedef* with a structure, it gets more complicated. For instance:

```
typedef struct
{
   short top;
   short left;
   short bottom;
   short right;
} Rect;
```

Here, we defined the *typedef* to be a rectangle rather than a predefined type. To use this rectangle, you might say:

```
Rect theRect;

SetRect(&theRect, 10, 15, 100, 150);
```

Here, we have created a new variable called *theRect* of type *Rect*. Then we called a function and passed the function the address of that rectangle. The function called sets the coordinates of the rectangle to the four additional values that we passed in.

The variable type *Rect* has already been defined for you. It is used by the Macintosh Toolbox and is defined in a file called *QuickDraw.h*. You can open up that file to see what the definition of a rectangle is. You can also see some other data structures used by the Toolbox and their definitions as well. We also called a function called *SetRect,* which is a Toolbox call. The definition of that function is also in *QuickDraw.h*. You can find an explanation of how this function works in *Inside Macintosh* or the *THINK Reference™*.

Structures

A structure provides a way to declare new variables and variable types. It consists of a number of variables that are collected under one name. This is one of the most powerful features in C++. Both C and Pascal allow you to create structures for containing specific information, both numeric and nonnumeric. In Pascal, this capability is called a *record*, and in C it is called a *structure.* C++ differs from C and Pascal in that it allows structures to contain *member functions*, which we will discuss later in this chapter. The form for a structure is:

```
struct identifier
{
  structure declaration member list
} declared variables;
```

Declaring a Structure

To declare a structure, you might write:

```
struct Automobile        //type
{
  char make[20];
  char model[20];
  short numDoors;
  long mileage;
};

Automobile usedCar;      //instance of type
```

In this example, we declared a new type called *Automobile.* We then created a variable called *usedCar* of type *Automobile.* Now look at the next code fragment:

```
struct Automobile        //type
{
  char make[20];
  char model[20];
  short numDoors;
  long mileage;
} usedCar, *pUsedCar;
```

This example also creates a pointer to an *Automobile* type.

Anonymous Structures

C++ allows you to create anonymous structures in much the same way that you create anonymous types. For example:

```
struct Automobile       //type
{
  char make[20];
  char model[20];
  short numDoors;
  long mileage;
} usedCar, newCar;
```

Here, the structure creates two variables called *usedCar* and *newCar*. No new named type is defined; therefore, the structure is anonymous.

Referencing Data Elements of Individual Structures

To access the elements of individual structures, you must use the *dot* (.) operator. For instance:

```
struct Automobile       //type
{
  char make[20];
  char model[20];
  short numDoors;
  long mileage;
} usedCar;

usedCar.numDoors = 4;
usedCar.mileage = 250000;
```

However, if you have a pointer to the structure, you may use the -> (arrow) operator:

```
struct Automobile       //type
{
  char make[20];
  char model[20];
  short numDoors;
  long mileage;
} usedCar, *pUsedCar1;

pUsedCar = &usedCar1;

pUsedCar->numDoors = 4;
pUsedCar->mileage = 250000;
```

You may also use indirect selection (the *dot* operator) if you dereference the pointer, as shown in the following code:

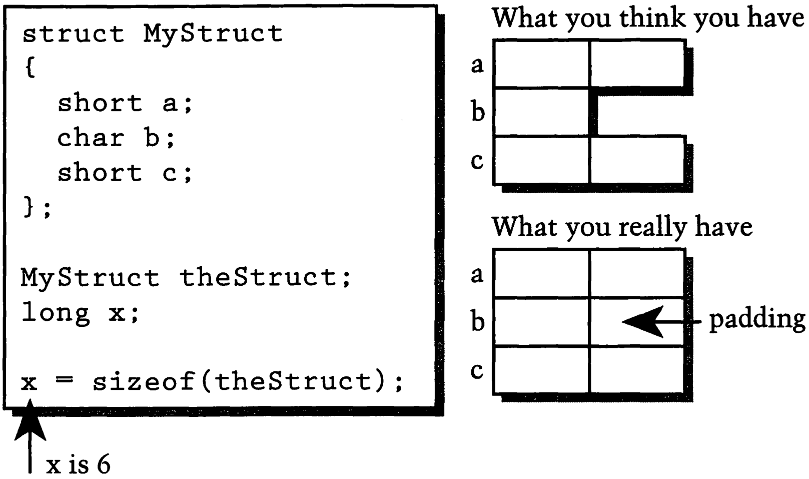
```
pUsedCar->mileage = 250000;
```

or:

```
(*pUsedCar).mileage = 250000;
```

Padding

An important thing to remember about memory assignment on the Mac is that the 68xxx is a "word" (2-byte) machine. This means that anytime you get or use any possible variable, the address always has to begin on an even-byte boundary. If you attempt to address an odd-byte variable, you will get a bomb. To ensure that the address falls on an even-byte boundary, the machine pads the odd byte, as shown in Figure 8.4.



```
struct MyStruct
{
  short a;
  char b;
  short c;
};

MyStruct theStruct;
long x;

x = sizeof(theStruct);
```

Figure 8.4 *Example of padding.*

Creating an Array of Structures

Suppose that you want to create a table showing, for example, the number of days in each month. Perhaps the easiest way to do this is to create an array of structures. You initialize the array of structures and enclose each member in curly braces, as in the following code:

```
struct
{
  char *month;
  short days;
} theMonths[] =
  {
    {"Jan", 31},
    {"Feb", 28},
    {"Mar", 31},
    {"Apr", 30},
    {"May", 31},
    {"Jun", 30},
    {"Jul", 31},
        {"Aug", 31},
        {"Sep", 30},
        {"Oct", 31},
        {"Nov", 30},
        {"Dec", 31},
    };
```

In this example, we have a structure that contains a pointer to some *chars* (months). We have also declared some *shorts* (days). We then declare an array of months and days and set them equal to the number of days in each month.

Structures and Bit Fields

Bit fields give you the ability to cut down on the kind of memory that an ordinary structure might consume. Look at the following code for a table showing the day, month, and year:

```
struct date
{
  short day;
  short month;
  short year;
};
```

Here, each *short* consumes 2 bytes for a total of 48 bits (6 bytes). Now look at the next example:

```
struct date
{
  unsigned day : 5;
  unsigned month : 4;
  unsigned year : 7;
};
```

By declaring unsigned bit fields, you have allocated an *int*, (2 bytes). The day consumes 5 bits of the *int*, the month 4, and the year 7, for a total of 16 bits (2 bytes), a saving of 32 bits (4 bytes) of memory. The disadvantages are that you cannot access the address of bit fields, and things run more slowly than when you access ordinary variables.

Unions

Unions are a way to allocate items that use the same storage area. The format of a union is similar to a structure and looks like:

```
union example
{
  short i;
  float f;
};
```

As illustrated in the example above, the word *union* replaces *struct*. All members of a union occupy the same memory space. If you address *short i* in memory, you will get 2 bytes, and if you address *float f*, you will get 4 bytes.

Suppose that you want to describe a point in a coordinate system that is sometimes described in Cartesian coordinates and at other times in polar coordinates. You cannot use both coordinate systems at the same time. Instead, you use a union to conserve space and to describe this point. For instance:

```
union point
{
  struct cartesian
  {
    short x;
    short y;
  };
  struct polar
  {
    short radius;
    short theta;
  };
};
```

Here, we created two structures inside a *union*. Use the first case whenever you want to access or update the value in Cartesian coordinates and the second case when you want to do so in polar coordinates.

To access the components of a *union*, you use the *dot* operator, as shown in the following code:

```
point.cartesian.x
point.cartesian.y
point.polar.radius
point.polar.theta
```

Operator Overloading

To improve the extendibility of the language, C++ allows operator overloading (in a similar fashion to function overloading). To overload an operator, you must use the keyword *operator*. In a way, you create more uses for operators. Figure 8.5 shows the C++ operators that can be overloaded.

+	-	*	/	%	^	&	|	-	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	>>=	<<=	==	!=	<=	>=	&&
||	++	--	[]	()	new	delete			

Figure 8.5 *Operators that can be overloaded.*

An example of operator overloading might be:

```
struct complex
{
  float r;
  float i;
};

complex a, b, c;
```

```
a.r = 3.0; a.i = 4.0;
b.r = 7.4; b.i = -5.6;

c = a + b;
complex operator +(complex x, complex y)
{
  complex temp;

  temp.r = x.r + y.r;
  temp.i = x.i + y.i;
  return (temp);
}
```

In this example, we have *float r* and *float i,* and we have declared three complex variables: *a, b,* and *c.* We then say that the real part of *a* is equal to 3, the imaginary part of *a* is equal to 4, the real part of *b* is equal to 7.4, and the imaginary part of *b* is equal to –5.6. Next, we set *c* equal to *a* + *b.* The plus (+) symbol works for *shorts* and *floats,* and so on, but it does not ordinarily work with a structure; the compiler will give us an error saying that we cannot add to structures together. To get around this, we declare a *new* operator for the + symbol, which will return a type *complex* and take as its operand on either side of the + symbol *x* and *y.* Next, we create a temporary where *temp.r* is equal to the sum of the real components and *temp.i* is equal to the sum of the imaginary components, and then we return the *temp.*

Be careful in choosing the appropriate operator to be overridden. If the + operator is overridden to mean multiply, the compiler will not care, but the next person to look at your code will!

Member Functions and Structures

Member functions are functions that are added to a structure, and they have access to the data members in the structure to which they belong. Using member functions allows you to access the data elements that form part of the structure without the need to use code that does not belong to the structure. This operation is known as *encapsulation.* Look at the following example:

```
struct automobile
{
  char model[20];
  long year;
  void InData(void);
  void OutData(void);
};

void automobile::InData()
{
  cout << "Enter model\n";
  cin >> model;
  cout << "Enter year";
  cin >> year;
}

void automobile::OutData()
{
  cout >> model >> "\n";
  cout >> year >> "\n";
}

main()
{
  automobile mercedes, ford;

  mercedes.InData();
  ford.InData();
  mercedes.OutData();
  ford.OutData();
}
```

In the example above, we have the model and year of the automobile and have added two prototype functions: *InData( )* and *OutData( ).* We then declare those functions: *void automobile::InData( )* and void *automobile::OutData( ).* The first function prompts you to input the model and year, and the second function prints out what you have entered. Note that we did not need to pass in *model* and *year* as arguments. They are inside the structure, so we have access to them.

In *main,* we declared two automobiles, a mercedes and a ford (of automobile type). Where we say *mercedes.InData( )* and *ford.InData,* the routine will input the data in the proper place. (The same is true with *mercedes.OutData( )* and *ford.OutData( ).*) Now look at another example of the *main*:

```
main()
{
  automobile *mercedes = new automobile;
  automobile *ford = new automobile;
  mercedes->InData();
  ford->InData();
  mercedes->OutData();
  ford->OutData();
```

In this *main,* we have declared automobiles on the heap instead of the stack. In other words, mercedes and ford are pointers to automobiles. That requires us to use the indirect (->) operator.

We can also write the *main* using a reference operator:

```
main()
{
  automobile &mercedes = *new automobile;
  automobile &ford = *new automobile;
  mercedes.InData();
  ford.InData();
  mercedes.OutData();
  ford.OutData();
}
```

Here, we have both mercedes and ford as references to an automobile. Even though we declare them as references, they are really pointers. The compiler will automatically deference them, allowing us to use the *dot* operator.

Summary

In this chapter, we have covered most of the advanced features of data structures:

- Using pointers and arrays.
- Dynamic memory allocation.
- Enumerated variables, structures, and unions.
- Operator overloading.
- Encapsulation.

These concepts lead us nicely into classes, which the next chapter discusses in detail.

Exercises

1) Define a structure that will be able to store the following information:
   book title
   author
   publisher
   copyright date

2) Write a program that declares an array of books on the heap. The array should be dynamic. Include the code you wrote in Chapter 7 to assign values to a book.

3) Rewrite the program using member functions.

9 Classes in C++

In Chapter 8, we explained that structures can contain both data elements and member functions, satisfying some elemental requirements of object-oriented programming (OOP). As powerful as C++ structures are in organizing data and functions (representing a significant advance on C structures), they still have some disadvantages. These can be overcome by the use of classes, which are more advanced forms of abstract data typing.

Defining a Class

A class definition is made up of two parts: (1) the name, composed of the keyword *class*; and (2) the declaration list enclosed in curly braces. For instance:

```
class TAutomobile
{
  private:
    char fModel[20];
    long fYear;
  protected:
    long fStickerPrice;
  public:
    void InData(void);
    void OutData(void);
};
```

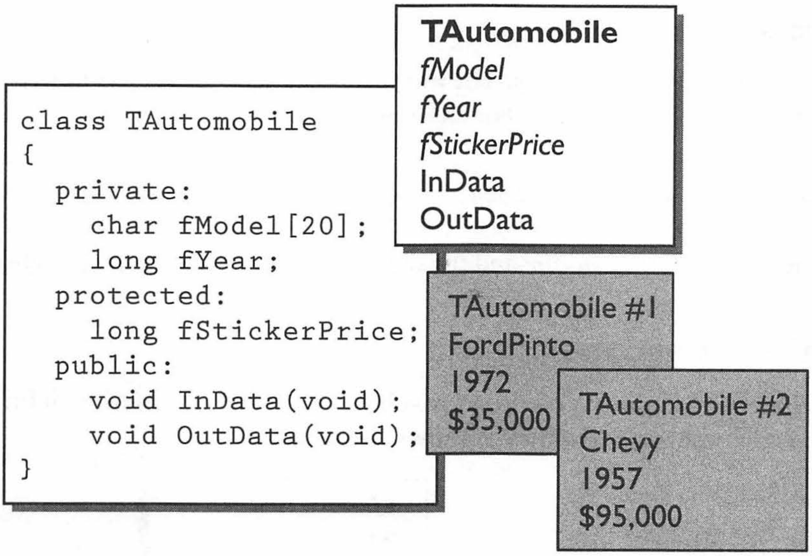
The class diagram for the above example would look like this:

```
TAutomobile
fModel
fYear
fStickerPrice
InData
OutData
```

Data members are like variables in a structure. Here, they are called instance variables. Each instance of the class will have its own storage area for the data members. Figure 9.1 shows the relationship of data members and instance variables.

Data members are typically private to the class; that is, code that is not part of the class cannot access this data. Member functions are functions added to a class. (You can also think of member functions as methods.)

Member functions are different from ordinary functions in that they have access to private members of their own class. They are defined only within the scope of their class, not within the global program scope. The class definition contains a prototype for each member function. In Figure 9.1, these are *void*



```
class TAutomobile
{
  private:
    char fModel[20];
    long fYear;
  protected:
    long fStickerPrice;
  public:
    void InData(void);
    void OutData(void);
}
```

Figure 9.1 *Relationship Between Data Members and Instance Variables*

InData(void); and *void OutData(void)*;. One of the most important aspects about classes is that they permit data hiding.

Classes and Structures

Classes look very much like structures except that they use the keyword *class* rather than *struct*, and nothing in the *private* section can be altered or even used except by means of the *public* member functions. Perhaps the major difference between classes and structures is that all data members and data functions in a class are private by default; everything in a structure is public by default.

Data Hiding

Data hiding is done by placing member functions and data in one of three sections: *private, protected,* and *public.* Data and functions declared private are accessible only to functions declared in the class. Those that are declared protected are only accessible to the class and its subclass. Those that are declared public are available to any code that has a reference to an object of this class.

Note:

It is a common mistake to forget the keyword *public* when defining a class. Remember that if no keyword appears, the default is private.

Objects

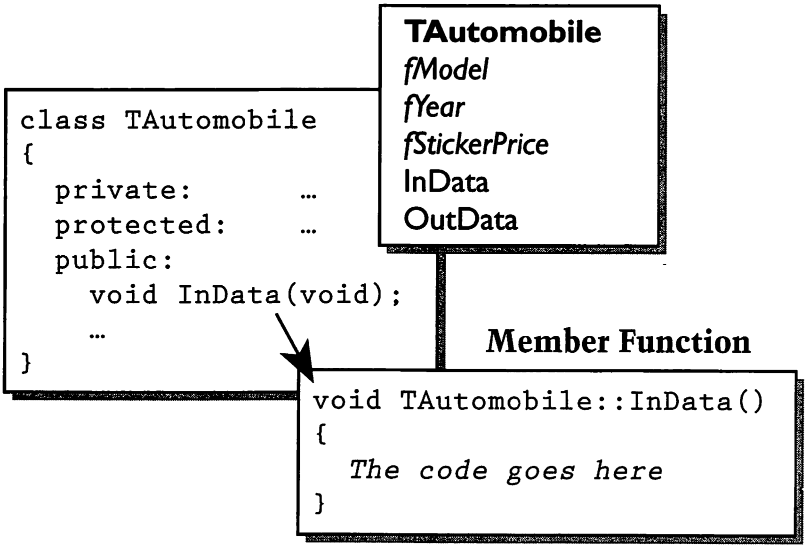
Objects are instances of a class. They are declared just like any other variable, except that the type is a class. For example:

```
TAutomobile mercedes, ford;
```

Here, the class is *TAutomobile* and the objects (instances) are *mercedes* and *ford.*

Member Functions

Somewhere in the source code you must have the member function definition; that is, its source code. Figure 9.2 shows this relationship.



```
class TAutomobile
{
  private:       ...
  protected:     ...
  public:
    void InData(void);
    ...
}
```

```
void TAutomobile::InData()
{
  The code goes here
}
```

Figure 9.2 *Member Function Definition*

Note that in the figure above, the :: operator identifies the class to which the member belongs. In this case, it is *TAutomobile.*

Constructors

A constructor is a special type of member function. It is called automatically when an object is created and does not need to be called explicitly. You should first declare a constructor in the *public* area of a class. An example of use of a constructor might be:

```
class TAutomobile
{
  private:
    char fModel[20];
    long fYear;
  protected:
    long fStickerPrice;
  public:
    TAutomobile(void);          //constructor
    void InData(void);
    void OutData(void);
};
```

As you can see from the above example, the constructor *(TAutomobile(void);)* has the same name *(TAutomobile)* as the class of which it is a member. If an array of objects is created, the constructor will be called once for each array element.

When you call *new* to create an instance of a class, two things happen: (1) memory is allocated, and (2) one or more constructors are called.

A constructor with no arguments passed is called a *default* constructor. It is possible to pass one or more arguments to a constructor, as shown below:

```
TAutomobile ford(arguments);
or
TAutomobile ford = new TAutomobile(arguments);
```

You use a constructor typically to initialize data and to create space for data on the heap. For example:

```
TAutomobile::TAutomobile()
{
  fModel = new char[20];
  fYear = new long;
  fStickerPrice = new long;
}
```

In the above example, *TAutomobile()* is a default constructor because we have not passed any arguments into it. We then say that we are getting a character array *(new char[20])* and making *fModel* point to it. The same is true with *fYear = new long* and *fStickerPrice = new long.* We now have memory on the heap for all three variables.

Destructors

A destructor is generally used for any type of cleanup operations. A destructor cannot take an argument nor can it be overloaded. You declare a destructor just as you would a constructor except that you precede the name by a tilde (~). An example of the use of a destructor might be:

```
class TAutomobile
{
  private:
    char fModel[20];
    long fYear;
  protected:
    long fStickerPrice;
  public:
    TAutomobile(void);
    ~TAutomobile(void);        //destructor
    void InData(void);
    void OutData(void);
};

TAutomobile::TAutomobile()
```

```
{
   delete fModel;
   delete fYear;
   delete fStickerPrice;
}
```

When you call *delete* to destroy an object, two things happen: (1) one or more destructors are called, and (2) memory is deallocated.

You may have noticed that constructors and destructors have no return values, and they are not preceded by the word *void*. The reason that *void* is not used is that both constructors and destructors *do* in fact return values. The returned value is implicit rather than explicit. It consists of a special pointer *this->*, which is the address of the object being created or destroyed. The implicit *this->* pointer provides a means by which a member function can know which object (instance of a class) it is dealing with. For now, you need only know that both constructors and destructors return a value, but you do not need to declare it.

Constructors and destructors are not inherited since they serve as class-specific initialization functions. More on inheritance later.

Constructors and destructors are useful but not absolutely necessary. You do not need a corresponding destructor for every constructor and vice versa. MacApp classes typically use an explicit initialization method rather than a constructor.

Initialization Functions

You may have initialization functions in addition to or instead of constructors. The convention for naming initialization functions is to begin the function name with the letter *I*. (The standard MacApp method of initializing objects of a given class is with an *I* function.) Call the *I* function for each object as soon as you have created it. You pass to your *I* function whatever parameters it needs. For instance:

```
class TAutomobile
{
   private: …
   protected: …
   public: …
      void IAutomobile(const char *model,
                       const long mileage);
```

```
    ...
};


main()
{
  TAutomobile ford, chevy;


  IAutomobile("Pinto", 24000);
  IAutomobile("57Chevy", 251925);
  ...
}
```

As you can see in the above example, the initialization function *void IAutomobile* looks like a constructor, except that the *T* has been changed to an *I*. The values *const char *model, const long mileage* are declared constants, which means that when we pass in values, they cannot be changed by that function; the function can only use them, it cannot change them. (Declaring the *const* is a safety measure, a form of defensive programming.) Note that the *I* functions are called as soon as the objects *ford* and *chevy* were created.

I functions are used mainly to initialize variables, not to allocate memory.

Accessor Methods

One way to give other parts of your program access to *private* data is through an *accessor*. For example:

```
class TAutomobile
{
  private:
    char fModel[20];
    long fYear;
  protected:
    long fStickerPrice;
  public:
    TAutomobile(void);
```

```
        ~TAutomobile(void);
        long GetYear(void);
        void SetYear(long year);
        void InData(void);
        void OutData(void);
    };

    long TAutomobile::GetYear()
    {
      return (fYear);
    }

    void TAutomobile::SetYear(long year)
    {
      fYear = year;
    }
```

Note:

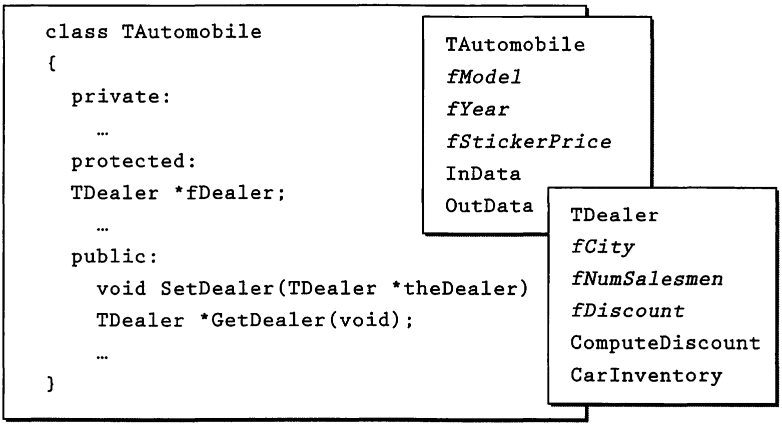
Constructors, destructors, and accessor functions are often omitted from class diagrams.

In this example, we have the variables *fModel* and *fYear* in the *private* section, but we allow members of the *public* section to get a certain amount of access to the *private* section by giving them the routines *GetYear* and *SetYear*. The first returns the value of the year, and the second allows them to set the year. By the way, accessors should return data, not the address to data.

It is a good idea to provide accessor functions for each data member that must be accessed from outside the class. As a first step, you may want to provide accessor methods for all data members, and then remove those that are not needed.

Collaborators

Data members do not have to be just simple variables. They can be references to other objects; for example, collaborators. Collaborator data members can have accessor methods just like any other member. Figure 9.3 shows the relationship between collaborators and classes.



```
class TAutomobile
{
  private:
    ...
  protected:
  TDealer *fDealer;
    ...
  public:
    void SetDealer(TDealer *theDealer)
    TDealer *GetDealer(void);
    ...
}
```

Figure 9.3 *Collaborators*

Collaborator data members can have accessor methods just like any other member.

Friends

A *friend* is called just that because it has access to the *private* section of the class to which it is a friend. (You can have both *friend classes* and *friend functions.*) Although friends can come in mighty handy at times, it is best to exercise restraint when creating friends. After all, the creation of friends compromises the whole idea of data hiding.

Friend Class

To declare a class as a friend, you place a statement within the class definition. For instance, the following declaration might be used in conjunction with our *TAutomobile* examples:

```
friend class TUsedCarSalesman;
```

It does not make any difference which section you place the *friend* statement: *public, protected,* or *private.* Member functions of the *friend class* will be able to manipulate the private data elements of the class to which it is a friend.

Friend Functions

Functions may also be friends of a class, and you declare a *friend function* for the same reasons that you declare a *friend class.* You can declare a *friend function* by placing the following statement in the class:

```
friend void GetSalesQuota(short quota);
```

As you can see, the key word in the declaration is *friend.*

Data-Hiding Convention

By convention, *public* members are listed first, followed by *protected,* then *private,* as shown in the following code:

```
class TAutomobile
{
  public:
    void InData(void);
    void OutData(void);
  protected:
    long *fStickerPrice;
  private:
    char *fModel;
    long *fYear;
};
```

Summary

In this chapter, we covered:

- Defining classes and objects in C++.
- Data members and member functions.
- Data hiding.
- Constructors and destructors.
- Initialization functions.
- Accessors.
- Collaborators.
- Friends.

Now it's time to put together everything that you have learned up to now into a good example. That is what Chapter 10 does with the list example.

Exercises

1) The following is a code to add two vectors:
void VAdd(float *a, float *b, float *c, short n)

```
{
  for (short i = 0; i < n; i++)
  {
    c[i] = a[i] * b[i];
  }
  return (c);
}
```

Write the corresponding function for a vector class:

```
void vector::add(vector &a, vector &b);
```

where the result of the addition is the implicit vector argument.

2) Modify the above code to support other vector operations (subtraction, multiplication, division, absolute value, etc.). Use operator overloading wherever possible.

3) A complex number is a number made up of two other numbers. One of the numbers is called a real number (these are the same numbers that you learned in arithmetic); the other an imaginary number. Both numbers are floating point. A complex number type could be defined using a structure as:

```
struct Complex
{
  float r;    // real part
  float i;    // imaginary part
};
```

When two complex numbers are added, subtracted, multiplied, or divided, the resulting number is also complex. To add two complex numbers, you would use the following algorithm:

```
C = A + B


C.r = A.r + B.r
C.i = A.i + B.i
```

Where *A*, *B*, and *C* are complex numbers and r and i are real and imaginary, respectively. To subtract two complex numbers, you would say:

```
C = A - B


C.r = A.r - B.r
C.i = A.i - B.i
```

To multiply two complex numbers, you would

```
C = A * B


C.r = (A.r * B.r) - (A.i * B.i)
C.i = (A.i * B.r) + (A.r * B.i)
```

And finally, to divide two complex numbers:

```
C = A / B

C.r = ((A.r * B.r) + (A.i * B.i)) / ((B.r * B.r) + (B.i * B.i))
C.i = ((A.i * B.r) - (A.r * B.i)) / ((B.r * B.r) + (B.i * B.i))
```

If *A.r* = 3, *A.i* = 4, *B.r* = 4, *B.i* = 2 then the results of the above operation would be:

	C.r	C.i
Addition	7	2
Subtraction	-1	6
Multiplication	20	10
Division	0.2	1.1

Create a complex data type using either *struct* or *class*. Add to your complex class, function members to assign values to the complex numbers. In addition, build function members that add, subtract, multiply and divide. Write a program that test this complex class on all of the above operations. You can use the above numbers to check your results. Implement operator overloading into your function members. In this case, adding two complex numbers with operator overloading would look like:

```
Complex a, b, c;

a.r = 3; a.i = 4;
b.r = 4, b.i = 2;
c = a + b;
```

10 Linked List Example

This chapter consists of a walkthrough of a fairly simple example. We provide a lengthy description of a problem, from definition through modeling, then we design the classes and class diagrams and create the methods for the classes. When finished, we will have gone through the basic process of object-oriented design methodology.

Statement of the Problem

Our problem (or objective) is to create a simple linked list, something that is used for a variety of things. For example, we may want to create a list of clients, along with their addresses and phone numbers. Also, when you have an event (like a mouse-down event) on the Macintosh, you create an "event record" where things are put into a queue. That queue is actually a linked list of the kind shown in Figure 10.1.

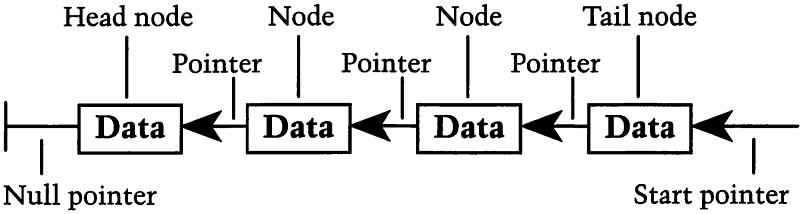


Figure 10.1 *A Linked List*

A linked list is made up of a number of nodes. The first node to be placed in the list is the head node, while the most recent node at any given moment is the tail node. Figure 10.2 shows the relationship of the pointers and nodes in a singly linked list.

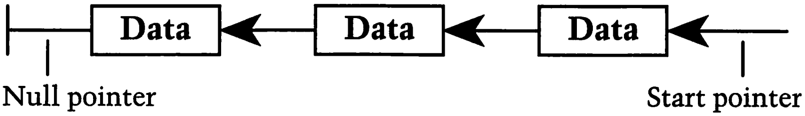


Figure 10.2 *Structure of a Singly Linked List*

Only the address of the tail is directly known to the program and is contained in the start pointer. All of the other nodes can be reached by "daisy-chaining" forward along the list in the direction of the head. When the end of the list is reached, the pointer of the head node is set to zero. Backward movement — from the head toward the tail — is not possible unless additional links are added.

Creating a Circular List

A variation on the singly linked list requires the head node to point back to the tail. This creates a circular list, and is shown in Figure 10.3.

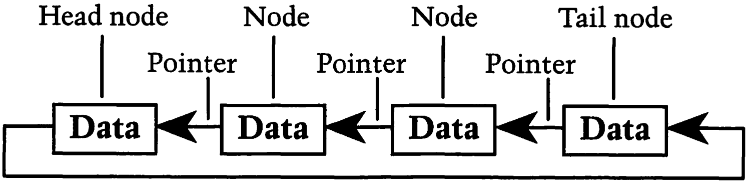


Figure 10.3 *A Circular List*

A singly linked list is useful when new nodes will be added only at the tail end of the list and when deletions also take place from the tail. Such a structure is perfectly adequate for a very large number of applications (the stack works in a way similar to this).

Adding and Deleting Nodes

Figure 10-4 shows a list comprising a single node; that is, when the head and the tail are the same. This list contains a null pointer and is pointed to by the start pointer.

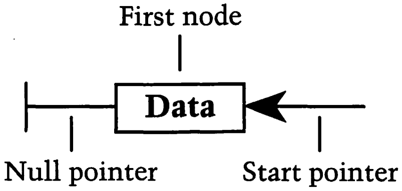


Figure 10.4 *A Single Node List*

Adding a new node at the tail end has the effect of "pushing" the head node farther up the list. Each time a node is added, the start pointer points to the new tail, which itself contains a new pointer to the head. The process, which is shown in Figure 10-5, continues as more new nodes are added at the tail of the list.

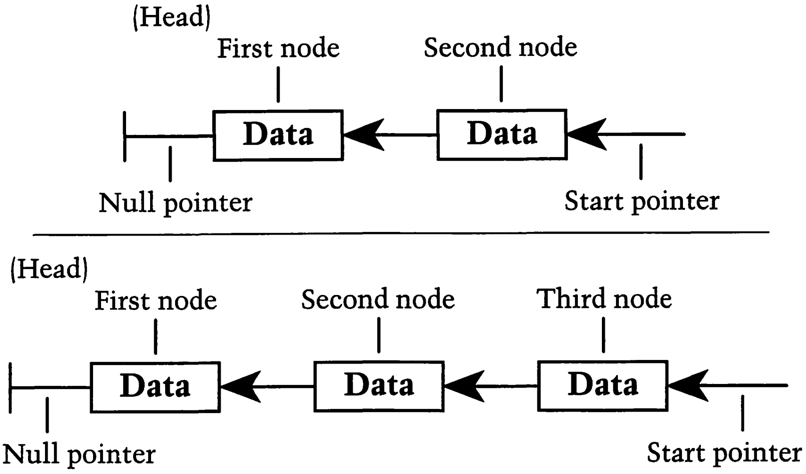


Figure 10.5 *Adding Nodes to the Tail of a Linked List*

To remove nodes from the tail end, set the start pointer to point to the next node. Next, delete the memory space occupied by the node using the *delete* operator, as shown in Figure 10.6.

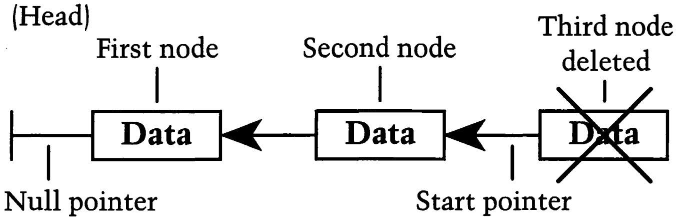


Figure 10.6 *Deleting a Node from a Linked List*

Building the Code

Now that we've determined what our objective is, it's time to build the code that will solve the problem. This actually takes quite a bit of insight. It is not as simple as saying, "Okay, now that I've described this, it's perfectly obvious how this code needs to be written." This can, in fact, take hours and numerous iterations. So, we start out with the knowledge that we have two things: a *list* and a *node*. Now we visualize how we're going to deal with them.

Creating a Friend Class

First, we have to create two classes: a *list* class and a *node* class. We begin with the *node class*, knowing that the *nodes* are a part of the *list* and must be accessed by the *list*. Therefore, we'll make this *node class* a friend to another class called *TList*. The *node class* has no public elements; the entire class is private. Objects of class *TNode* will only be accessible to member functions of class *TList*. The first part of the code will look like this:

```
class TNode
{
  friend class TList;

  TNode *next;
  char data[20];
};
```

Here, the node contains a pointer to *next*, which contains the address of the next node. In addition, the node contains the data that we are storing in the list. This data may be an address, phone number, or any other type of data. For now, we've allotted 20 characters. Now we're ready for the next chunk of code:

```
class TList
{
  public:
    TList(void) {start = 0;}
    ~TList(void);
    void Insert(void);
    void Extract(void);
    void InputData(void);
    void DisplayData(void);
  private:
    TNode *start;
}
```

In this part of the code, the *list* class contains a pointer to the start of the *list*. This pointer is private, so it can only by accessed by a member function.

We know that when we create this list we have to initialize the start pointer. When the list is first created, it contains no nodes. Therefore, we need to zero-out the start pointer. We do this by creating a *TList* constructor, inside of which we set the start point to *0*. (Notice that we have written an *inline* function here.) Once we have a constructor, we decide that it would be nice to have a destructor that goes through and deletes all the nodes when the list is deleted. So, we create the destructor *~TList*.

We know that we will want to insert things into the list, so we write the member function *void Insert(void);* . We'll also want to delete or extract things from the list, so we create *void Extract(void);*. In addition, we have written two other functions, *void InputData(void);* and *void DisplayData(void);*, which will allow us to input information into a data area and display it.

Insert Function

The *insert* function is called when a new node is to be added to the list:

```
void TList::Insert()
{
  TNode *temp;

  temp = new TNode;
  if (!temp) //no more space
  {
    cout << "Out of space\n";
    return;
  }
  if (!start) //head node
  {
    start = temp;
    temp->next = 0;
  }
  else    //not head node
  {
    temp->next = start;
```

```
      start = temp;
    }
  }
```

To insert a new node into the list, we declare a temporary pointer to a node. Then we call *new TNode,* which will give us the memory for the new node. We put the pointer to that memory into *temp.* Next we say, "If temp is not valid, we're out of memory space." We then write that out to the screen with *cout << "Out of space\n";*. If temp is not *0,* we're okay.

Next, we say, "If *start* is not valid (that is, *start* is pointing to 0), then this will be the first node to be added." We then set the start node equal to the one that we just created: *temp.* This has a pointer to the next node, and we set it back to *0.* If the start node is not pointing to *0,* there is already a node in the list. That means that we will be adding to the tail of that list. Therefore, we take the start node and put it into the pointer to the next: *temp->next = start.* Next we place the pointer to memory into *start.*

Extract Function

The *extract* function removes the current tail node from the list and reclaims the heap space:

```
void TList::Extract()
{
  TNode *p1, p2;

  if (!start)
  {
    cout << "\nEmpty list\n";
    return;
  }
  p1 = start;
  p2 = p1->next;
  delete p1;
  start = p2;
}
```

In the extract function, we create two pointers to *TNode (*p1, p2).* We then ask if the start pointer is valid using the expression *if (!start).* If *start* is pointing to *0,* we output to the screen : "Empty list." That ends the function because if there are no nodes, there is nothing to extract.

However, in case *start* is pointing to the next node, we make copies of the start node and the next node it is pointing to: *p1 = start;* and *p2 = p1->next;*. We then delete *p1*, which is the data of the node, and set the start pointer equal to *p2*.

The reason for using the local variables *p*1 and *p*2 is that had we simply said, "Delete start" (without making a copy of *next*), we would delete the node and the pointer to the next node, which we would not be able to retrieve. These temporary variables are really safety nets.

InputData Function

Our sequence is to first get the node and then to put the data into the node.

The *InputData* function reads the data from the user, then stores the data in the current node:

```
void TList::InputData()
{
  cout << "Data: ";
  cin >> start->data;
}
```

All this function does is to tell us that we need to input data with the expression cout << *"Data: ";*. It waits for the data to be input and then stores it in the start node: *cin* >> *start->data;*.

DisplayData Function

This function is a little more versatile than the *InputData* function. It will output to the screen all the data contained in all the nodes in the list:

```
void TList::DisplayData()
{
  TNode *i = start;
```

```
    while (i)
    {
      cout << i->data << "\n";
      i = i->next;
    }
  }
```

Here, we make *i* a pointer to a node, and we initialize *i* with *start* all in the same line. Next we say, "While *i* is not equal to *0*, execute the following loop." In the loop, we output the data of one node and then go to the next (*i = i->next*). When we reach the tail, *i* will be equal to *0*, and the function will end.

Destructor

The destructor reclaims the heap space occupied by the list:

```
  TList::~TList()
  {
    TNode *p1, *p2;

    p1 = start;
    if (!start)
    {
      return;
    }
    while (p1)
    {
      p2 = p1->next;
      delete p1;
      p1 = p2;
    }
  }
```

Once again we have the "delete" problem, so we must make copies of the start node and the next node that it is pointing to. We then set *p1* equal to *start*. We do a check to see if the list is empty. If it is, we do not bother with the destructor.

If there is a node, however, we say that while *p1* (the *start*) is valid, we assign *p2* to be a pointer to the next node and delete *p1*. We then set *p2* equal to *p1*. Then we delete each node until we reach the end of the list.

Writing the Main Function

In order to use all of this code, we must create the *main* function. The most outstanding feature of this example is that the implementation details of how the list is managed are hidden from the outside user. Let's look at the code:

```
#include <iostreams.h>

main()
{
  TList myList;
  myList.Insert();              //Input data
  myList.InputData();
  myList.Insert();
  myList.InputData();
  myList.Insert();
  myList.InputData();
  myList.Insert();
  myList.InputData();
  myList.DisplayData();         //Show data

  myList.Extract();             //Remove two nodes
  myList.Extract();

  myList.DisplayData();         //Show data
}
```

In this *main* function, we start by declaring a *#include*, in this case the iostreams, and then declaring a list: *TList myList*. As soon as this declaration is hit, the constructor for *myList* is called to initialize the start pointer that is contained in *TList*. (Note that this is not evident anywhere in the list. We do not have access to the start pointer because it is hidden from us.)

Next, we call *InsertData* and *InputData* to insert and put data into four nodes. After that, we call *DisplayData* to see what is in the nodes. We may not like what we see, so we remove the last two nodes and then display the data once more.

Once the function reaches the last curly brace, the destructor is called. It goes through and cleans out all of the memory.

Summary

In this linked list example, we covered:

- Defining data classes
- Data hiding and encapsulation
- Writing and calling methods
- Using dynamically allocated objects

In the next chapter, we'll show how to create subclasses and describe how inheritance works. We'll even use subclasses of *TNode* and *TList* as examples so that you can see more clearly the relationships between parent classes, subclasses, abstract classes, and inheritance.

Exercises

1) Rewrite the book example from Chapter 8 to incorporate the books in a linked list.

2) Modify the Linked list example to:

   a) a circular list

   b) a doubly linked list

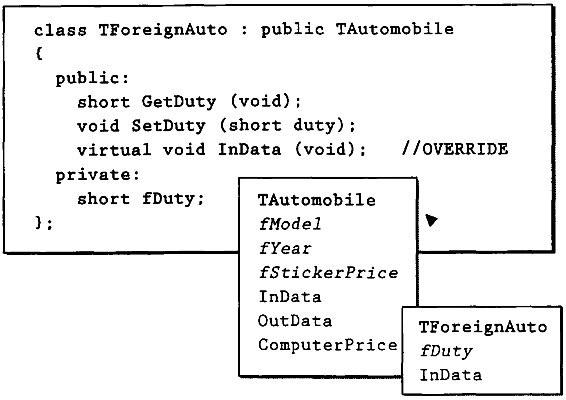
11 Subclassing and Inheritance

Object-oriented programming expands abstract data types by allowing a type and subtype association. In C++, the instrument for this association is subclassing. The subclass acquires shared characteristics—data members and member functions—from the parent class through *inheritance.* Because of their ability to reuse code and save storage space, subclassing and inheritance are the most useful and powerful aspects of object-oriented programming.

We touched on subclassing and inheritance briefly in Chapter 2. In this chapter, we examine in detail how to define derived classes, the value of virtual functions, and how and when to use protection keywords in base and derived classes. We also look at constructors, destructors, and *I* functions in derived classes; static members and static member functions; accessing member data; and the *this* pointer.

Defining Derived Classes

The existing class is called the *base class*, and the new class is called the *derived class*. There is no theoretical limit to how far the derivations can extend, but there are practical limits. Figure 11.1 shows the definition of a new class, complete with class diagrams for both the base and the derived classes.



```
class TForeignAuto : public TAutomobile
{
  public:
    short GetDuty (void);
    void SetDuty (short duty);
    virtual void InData (void);    //OVERRIDE
  private:
    short fDuty;
};
```

Figure 11.1 *Defining a derived class.*

In declaring a derived class, the name of the base class appears after the name of the derived class, separated by a colon. The keyword *public* makes the public section of the base accessible to the derived class and any further classes derived from that.

In the derived class, you must specify new data members and new member functions. Member functions may be overridden (as shown in Figure 11.1). The function will have the same name as the member function in the base class, but its purpose will differ from that of the base class. Member functions of the derived class can only access public and protected (not private) members of the base class.

For a derived class to override an inherited function, that function must have been prototyped as *virtual* in the base class (and not be *private*). The term *virtual* in Figure 11.1 means that the function *InData* may be overridden; *virtual void InData(void);* is the prototype. *InData* is then overridden in *TForeignAuto.* It appears that there is now an ambiguity because there are two *InData* functions. At compile time, the compiler does not know which function will be called, the *TAutomobile InData* or the *TForeignAuto InData*. This is determined at run time.

It is a good idea to make all public and protected functions virtual. Then, to optimize, go back and make functions that are not overridden nonvirtual. (If you have the "Optimize monomorphic methods" box checked, the compiler will optimize automatically.) You'll want to do this simply because nonvirtual functions run faster than virtual functions.

Virtual Functions

The word *virtual* tells the compiler to check the actual class of the object at run time. This ensures that the derived class's member function will be called. This happens even if the pointer used to access the member function is declared as a pointer to the base class. For instance, if we create a pointer to *TAutomobile* but assign that pointer the value of the address of a foreign automobile, at run time the program will determine that the pointer is actually pointing to a foreign automobile.

Pure Virtual Functions

Virtual functions initialized to zero are considered *pure virtual functions*. This process forces all the subclasses to override the routine. The format for a pure virtual function is shown in Figure 11.2.

```
class TAutomobile
{
  private:
     .
     .
     .
  public:
     virtual void InData(void) = 0;
     virtual void OutData(void) = 0;
};
```

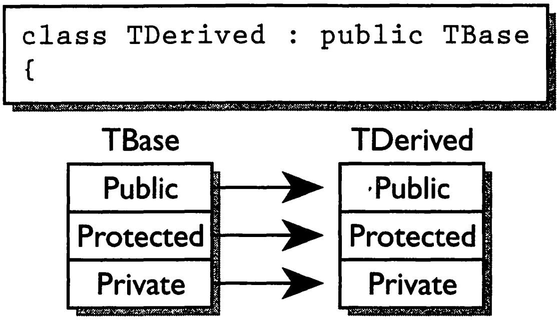
Figure 11.2 *Format for a pure virtual function.*

The syntax = *0* indicates that every concrete derived class must define its own version of the member function. Otherwise, there will be a compiler error. The concept is a bit obscure, but that is the syntax.

The definition is only necessary if we create an object. If we have not created an instance to an object, there will be no pointer. However, once we create an instance of an object, we need a pointer to a function. In this case, the pointer would be to zero—ERROR!—unless we define another version of the member function.

Protection Keyword

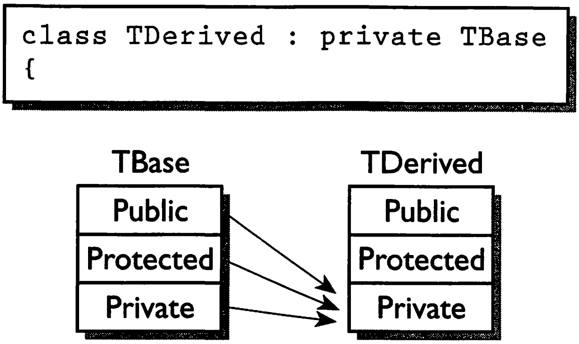
When you preface the base class name with the *public* keyword, all members inherited from the base class retain their original public, protected, or private status. Figure 11.3 illustrates this.



```
class TDerived : public TBase
{
```

Figure 11.3 *Protection keyword: public.*

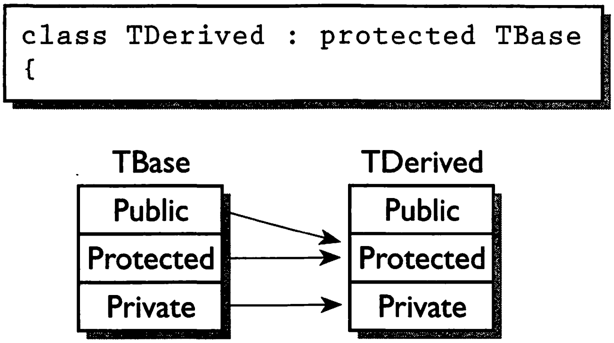
If you preface the base class name with the *private* keyword, all members inherited from the base class will become private to the derived class (see Figure 11.4). This means that the derived class may not have access to (cannot change) the private members of the base class. Any private members retain their original status.



```
class TDerived : private TBase
{
```

Figure 11.4 *Protection keyword: private.*

If you preface the base class name with the *protected* keyword (Figure 11.5), all public members become protected members in the derived class. Any private and protected members retain their original status.



```
class TDerived : protected TBase
{
```

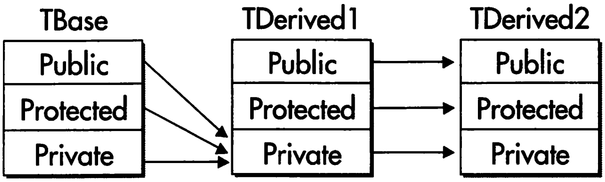
Figure 11.5 *Protection keyword: protected.*

If you do not specify a protection keyword, the default is *private*. This default feature is a holdover from early versions of C++ and should be avoided. Always preface the base class name with a protection keyword in the derived class declaration.

The significance of using the protection keyword in a derived class declaration becomes important only if another class inherits from the derived class. If the base class was converted to private, any further derivations of the derived class cannot access any of the base class members. This becomes apparent in Figure 11.6.

```
class TDerived : private TBase
```

```
class TDerived2 : public TDerived
```



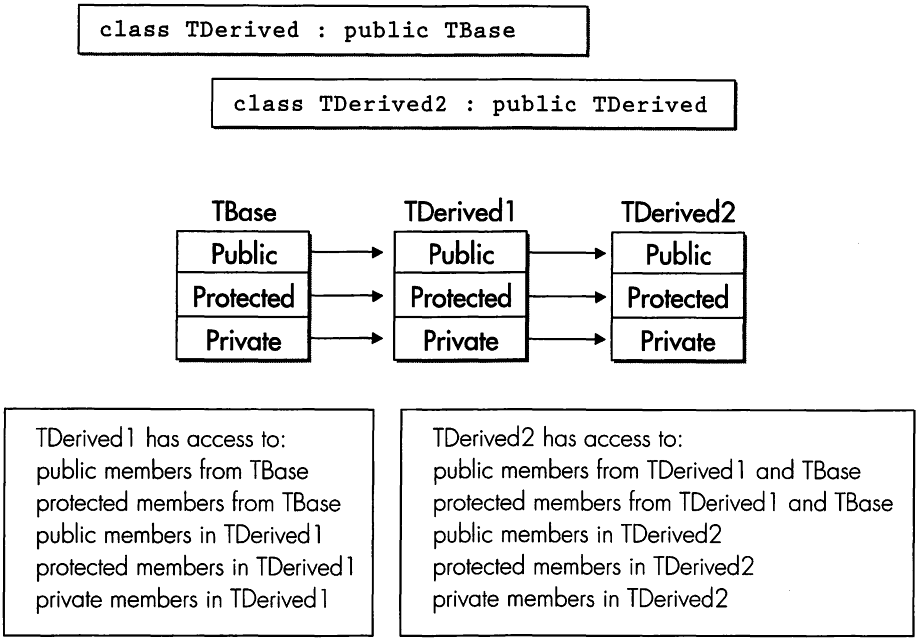
TDerived1 has access to:
public members from TBase
protected members from TBase
public members in TDerived1
protected members in TDerived1
private members in TDerived1

TDerived2 has access to:
public members from TDerived1
protected members from TDerived1
public members in TDerived2
protected members in TDerived2
private members in TDerived2

Figure 11.6 *Private inheritance of derived classes.*

In Figure 11.6, *TDerived1* creates its own copies of all of *TBase*'s public, protected, and private information (it creates its own version of everything). If *TBase* has a *short a* in its private section, *TDerived1* will have a copy of that *short a*, but it cannot access the *short a* in *TBase*. If the *short a* were in *TBase*'s protected section, *TDerived1* would have access to it.

In this same figure, *TDerived2* has no access to any members in *TBase* because *TDerived1* has made all of *TBase*'s information private. Figure 11.7 shows that if the information from the base class is declared public in the first derived class, the second derived class will still have access to the public and protected members of the base class.



```
class TDerived : public TBase
```

```
class TDerived2 : public TDerived
```

TDerived1 has access to:
public members from TBase
protected members from TBase
public members in TDerived1
protected members in TDerived1
private members in TDerived1

TDerived2 has access to:
public members from TDerived1 and TBase
protected members from TDerived1 and TBase
public members in TDerived2
protected members in TDerived2
private members in TDerived2

Figure 11.7 *Public inheritance of derived classes.*

Note that if the public information in *TBase* were to be put into the protected section of *TDerived1, TDerived2* would still have access to it. Also, remember that the only things that can access the private section of a class are the members of the class itself and friend classes.

Constructors in Derived Classes

When base and derived classes both have constructors, the constructor in the base class is called first. When that call is complete, the derived class is called. The headers for both the base and derived classes are shown in Figure 11.8.

Header of base class constructor:

```
TBase::TBase(short a, short b)
```

Header of derived class constructor:

```
TDerived::TDerived(short a, short b, float c)
```

Figure 11.8 *Headers for base and derived class constructors.*

The argument list in the prototype and header for a derived class constructor should include the arguments and argument types for both the base class and the derived class. In addition, the arguments (but not the types) for the base class should be enclosed in parentheses and appended to the end of the derived class header, preceded by a colon. Figure 11.9 shows this format.

```
class TBase
{
  public:
    TBase(short a, short b):
};
```

```
class TDerived : public TBase
{
  public:
    TDerived(short a, short b, float c):
};
```

```
TBase::TBase(short a, short b)
{
  ...
}
```

```
TDerived::TDerived(short a, short b, float c) : (a, b)
{
  ...
}
```

Figure 11.9 *Base and derived class constructor formats.*

If the derived class has any members that are not in the parent class, the derived class constructor should initialize them. For instance, in our example of

TForeignAuto, we have added a member—*fDuty*—that was not in the original *TAutomobile* class. So, the responsibility of the constructor in *TForeignAuto* is to initialize *fDuty*, not *fModel*, *fYear*, and *fStickerPrice*. The constructor of the base class will initialize those objects.

If the constructor for a class is not public, only *friends* of that class will be able to create objects of that class, since creating an object is just like calling its constructor.

Destructors in Derived Classes

Since destructors work in reverse order from constructors, the destructor in the derived class is called first, followed by the destructor for the base class. Since destructors do not pass arguments, the destructor of a derived class does not require any special syntax. Classes that will be used as base classes for derived classes should have virtual destructors.

WARNING

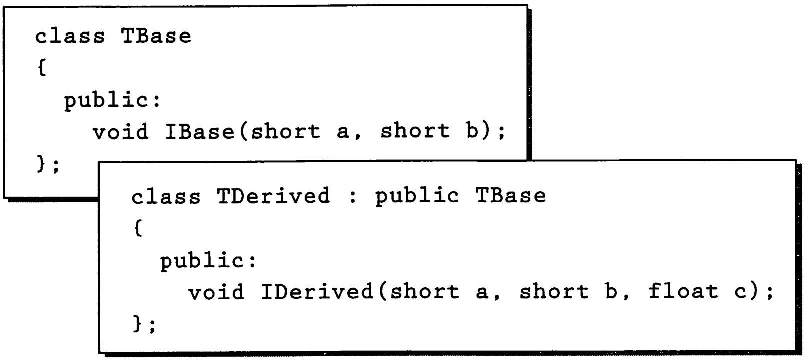
Watch out for the error of trying to reclaim the same heap space more than once.

To iterate:

- Constructors for parent classes execute *before* the constructors for derived classes.
- Destructors for parent classes execute *after* the destructors for derived classes.

"I" Functions in Derived Classes

If you are using *I* functions, then every class should have its own (as shown in Figure 11.10). The *I* function for the derived class should include the arguments from the *I* function in the base class.

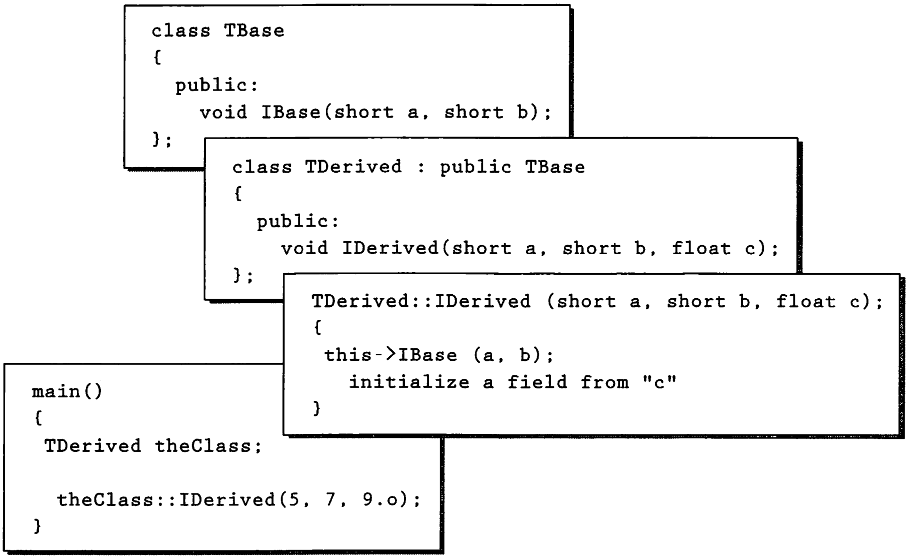


```
class TBase
{
  public:
    void IBase(short a, short b);
};
```

```
class TDerived : public TBase
{
  public:
    void IDerived(short a, short b, float c);
};
```

Figure 11.10 *"I" functions in derived classes.*

The *I* function for a subclass should gather initialization values for each of its data members, including inherited ones. (There is no mechanism for the *IDerived* class to call the *IBase* function unless you specifically write it inside of *IDerived*. Therefore, you need to pass in all the variables from *IBase* to *IDerived*.) The *I* function for the subclass should also initialize the data members that are defined by the class and call the inherited *I* function to initialize inherited data members. Figure 11.11 illustrates the initialization process.



```
class TBase
{
  public:
    void IBase(short a, short b);
};
```

```
class TDerived : public TBase
{
  public:
    void IDerived(short a, short b, float c);
};
```

```
TDerived::IDerived (short a, short b, float c);
{
 this->IBase (a, b);
   initialize a field from "c"
}
```

```
main()
{
 TDerived theClass;

  theClass::IDerived(5, 7, 9.o);
}
```

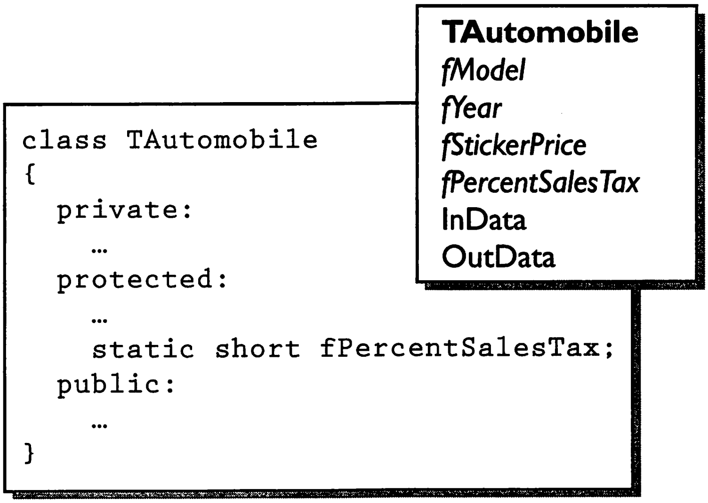
Figure 11.11 *Initializing data members in a derived class "I" function.*

In Figure 11.11, we have created *IBase* in the base class and *IDerived* in the subclass. In the *main* function, we have created *TDerivedtheClass*. Now we need to call the *I* function immediately, which we do, and we pass in the variables *5, 7,* and *9.0* for *a, b,* and *c*. We then pass in *a* and *b* to *IBase* and have it initialize them. Then we initialize *c*.

As you can also see from the figure, the creator of the object only has to call one initialization function.

Static Members

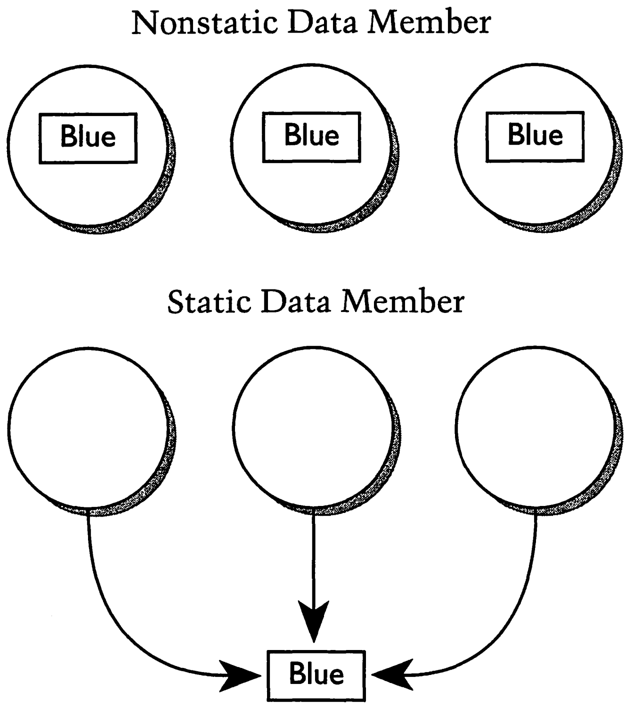
Static members are those that have the same address (and value) for all objects of the class. They are almost like global variables within the class. Figure 11.12 shows the format for the declaration of a static member.



```
class TAutomobile
{
  private:
    ...
  protected:
    ...
    static short fPercentSalesTax;
  public:
    ...
}
```

Figure 11.12 *Declaration of a static member.*

Static members save storage space, which is reserved only once, and all objects reference that storage. Figure 11.13 illustrates this process.



Nonstatic Data Member

Static Data Member

Figure 11.13 *Storage areas for nonstatic and static data members.*

In Figure 11.13, if we had three objects that all had an instance variable of *blue,* then each object would have its own copy of blue. However, if we make the instance variable in the three objects static, then all three share the same memory location for blue.

Static members do not need to have constant values, but the values are always the same for all instances of the class. In other words, you can change the value of the static member, but once you change it for one, you change it for all.

Static Member Functions

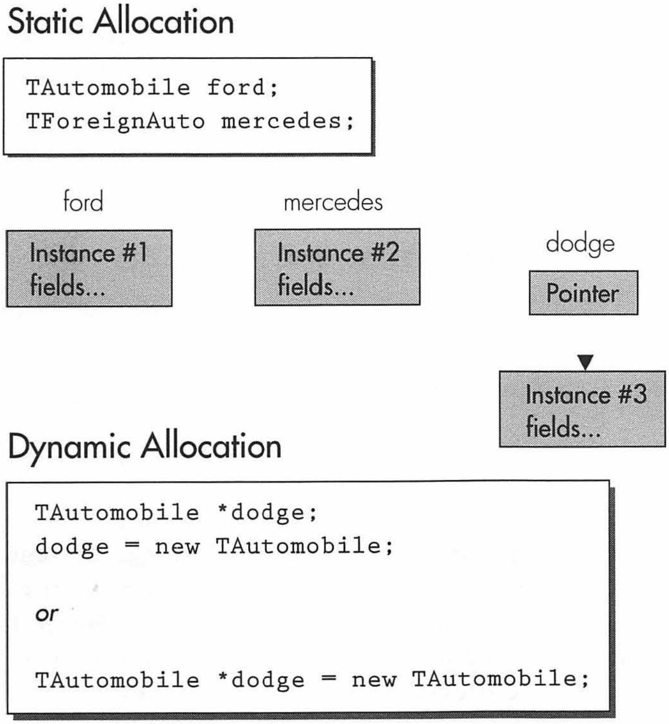
Static member functions are member functions declared with the keyword *static.* A static member function in a class can access only the static members of a class. They cannot access any of the nonstatic members of the class (although they can access them through *this->*).

When to Use Static Members

Use static members as a replacement for global variables. Static members and static member functions are often preferable to global variables because they can be protected from unauthorized modifications.

Creating Objects (Instances)

You can create objects or instances either by static allocation (on the stack) or dynamic allocation (on the heap). Figure 11.14 shows the process of creating objects with both types of allocation.



Static Allocation

```
TAutomobile ford;
TForeignAuto mercedes;
```

Dynamic Allocation

```
TAutomobile *dodge;
dodge = new TAutomobile;

or

TAutomobile *dodge = new TAutomobile;
```

Figure 11.14 *Example of creating objects using static and dynamic allocation.*

Dynamic Object

The class name that follows the *new* operator determines the actual class of a dynamically allocated object, no matter what type its pointer is defined to be. You might declare a dynamic object this way:

```
TAutomobile *dodge = new TAutomobile;
```

Initializing Instances

Initialize each newly created instance by calling its initialization member function immediately (if constructors are not used). For example:

```
TAutomobile ford;
TForeignAuto mercedes;
TAutomobile *dodge;

ford.IAutomobile("Ford Pinto", 257298);
mercedes.IForeignAuto("300E", 49995, 500);

dodge = new TAutomobile;
dodge->IAutomobile("Dodge Plymouth", 34796);
```

Using Instances

To get useful work from an instantiated object, send it a message. To send it a message, you must have a reference to it. You need to have a pointer to the object or the actual object itself to send a message to the object. Figure 11.15 shows an array of *AutoList* with four pointers to different objects (in this case, autos). The code is asking for the sticker price on the fourth auto.

Which Functions?

C++ must make all member functions available and keep track of which is to be called for each object. Figure 11.16 demonstrates the calling of an overridden function in a subclass.

```
fAutoList[4].ComputeStickerPrice;
dealerName = fDealer->GetName;
```

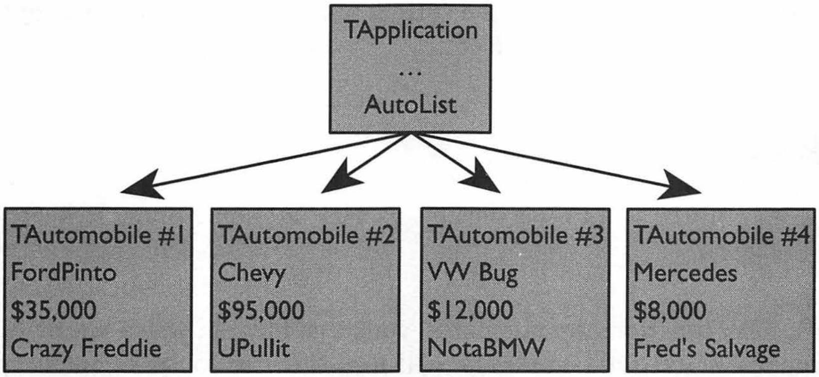


Figure 11.15 *Example of using instances.*

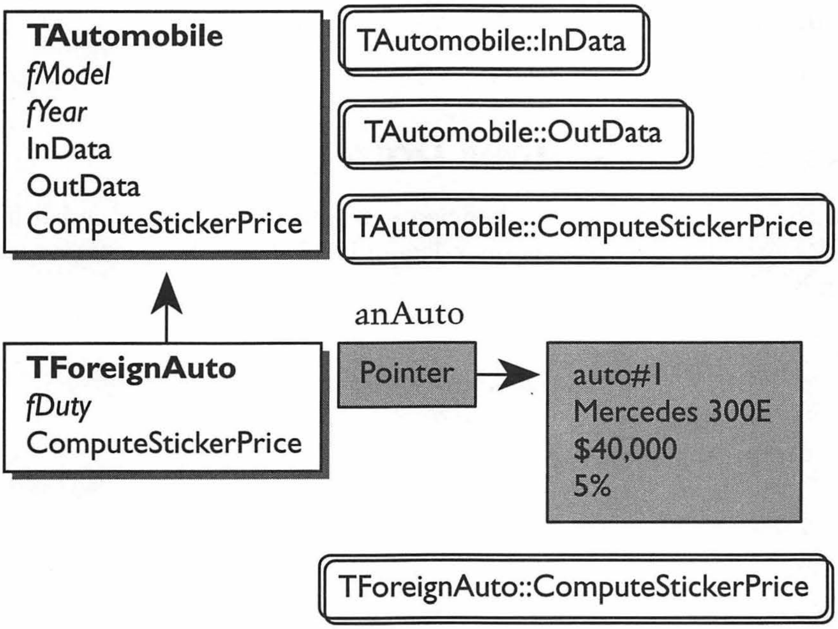


Figure 11.16 *Calling functions for each object.*

In Figure 11.16, we have *TAutomobile* and *TForeignAuto* and a pointer to *auto #1* (an instance), which is a foreign automobile. However, the pointer is actually pointing to *TAutomobile*. The question here is: "Which function will be called? *ComputeStickerPrice* for *TAutomobile* or *ComputeStickerPrice* for *TForeignAuto*?" C++ will call the correct function because it has kept track of what is to be called for each object.

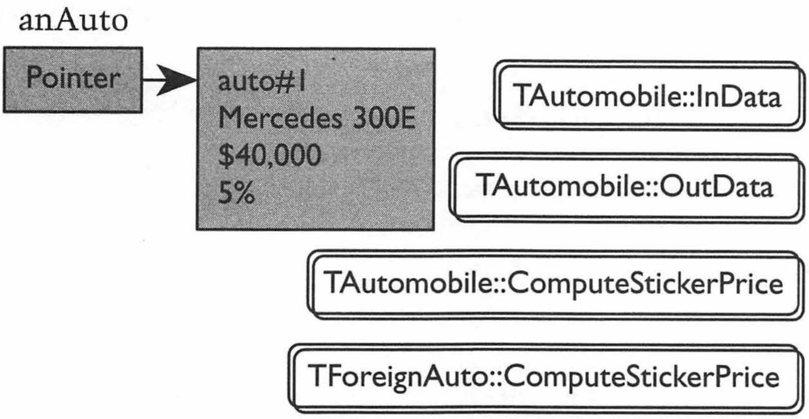
Which Data?

Messages must be sent to specific objects. Here are some things to keep in mind about objects:

- Member functions are not global; you must have an object reference to call them. For example, you must call *ford.InData*; you may not simply call *InData*. And, you may not call *ford.InData* from some place that does not have access to *ford*.
- Member functions use the same code for each instance. However, they usually reference the data members, and the data is unique for each instance.
- Effectively, every member function has a hidden parameter that points to the object data.

Accessing Member Data

Use the arrow operator (->) whenever you want to access a member of a particular class through a class pointer. Figure 11.17 shows the process for accessing member data.



```
// This call to a member function:
anAuto->ComputeStickerPrice (… )

// Really means something like this:
ComputeStickerPrice (anAuto, … )
```

Figure 11.17 *Accessing member data.*

In the code in Figure 11.17, we have a pointer (*anAuto*) to *mercedes*, and we are calling the function *ComputeStickerPrice*. Because we are calling a member function, we use the arrow from the pointer to the function.

The C++ "this" Pointer

The *this* pointer in C++ means "the address of the object (structure or class) that I am currently in." The *implicit* use of the *this* pointer allows a member function to know which object of a structure it is dealing with. Sometimes, however, it is necessary to use the *explicit* form of the pointer. From outside the member functions, for example, you would use an explicit reference, as shown in the following code:

```
anAuto->ComputeStickerPrice(…)
```

This means the same thing as:

```
ComputeStickerPrice(anAuto, …)
```

However, from inside the member function, we refer to the "current" object as *this,* and write the code this way:

```
this->ComputeStickerPrice(…)
```

That code means the same thing as:

```
ComputeStickerPrice(this, …)
```

or

```
ComputeStickerPrice(anAuto)
```

When calling an overridden member function, the call

```
this->ComputeStickerPrice(…)
```

gives us the same thing as:

```
TForeignAuto::ComputeStickerPrice(this, …)
```

If we want to call *TAutomobile::ComputeStickerPrice* instead, we can call it explicitly:

```
TAutomobile::ComputeStickerPrice(…)
```

This is the same as saying:

```
TAutomobile::ComputeStickerPrice(this, …)
```

Remember that the *this* call only works from a member function.

To iterate, the *this*-> argument to a member function can be used to show explicitly that a member or member function for the class is being accessed. Some programmers prefer the more explicit form, and there is no performance penalty for using it.

You are not required to use *this*-> when it is implicit, but by using it you can improve readability. Our recommendation is that you use *this*-> notation for all your messages.

Summary

In this chapter, we discussed subclassing and inheritance in detail. The topics covered included:

- Derived classes.
- Virtual functions.
- Constructors and destructors.
- Initialization functions.
- Using static members.
- The *this*-> pointer.

Exercises

1) Create a program called "Company" that implements the class design you created in Chapter 2. Create equipment and employee classes. Write member functions that enable each equipment object to output a description of itself (make, model, serial number, and, for computers only, amount of memory). Most of these functions are one-liners using *cout*. When your classes are ready, return to the main ("company") program and modify it to use your classes. Declare one variable for each of your class types, initialize each variable, and then send each one a message telling it to print a description of itself.

2) Add to the "company" tool by repeating the previous exercise for your employee classes. This is a somewhat bigger job than the last one because there are a few more methods and classes; but you know how to do it now, right? One new "wrinkle" for the employee objects is that they contain references to equipment objects, so some of your messages will be sent via pointers (rather than via static variable references).

3) Each of your employee objects should know how to print a paycheck that shows gross pay, tax, and net pay, and how to print a description of its piece of equipment (by sending a message to its collaborating equipment object). The bookkeeper object will have an array that holds a reference to each employee, plus methods to print paychecks and equipment descriptions for

each employee. (You could of course have the bookkeeper keep a variable-length list of employees rather than a fixed-length array, but an array will probably be easier to code.) For this exercise you may omit the reports of bosses (for secretaries) and language skills (for programmers) discussed in Exercise 1 of Chapter 2.

4) In your main program, test your classes by setting up a company that corresponds to the one described in Exercise 1 of Chapter 2 (i.e., one manager, one secretary, etc.) and printing the paychecks and equipment list for those employees.

Suggestion: Start by implementing just enough code to instantiate one type of employee object, such as *TProgrammer*, and test that single object via calls from the main program. Then implement additional objects one at a time until you have the entire company represented.

If you have additional time, and if the main program that you wrote instantiated all objects as static objects, change one or two of these objects to be allocated dynamically so that they must be accessed via pointers.

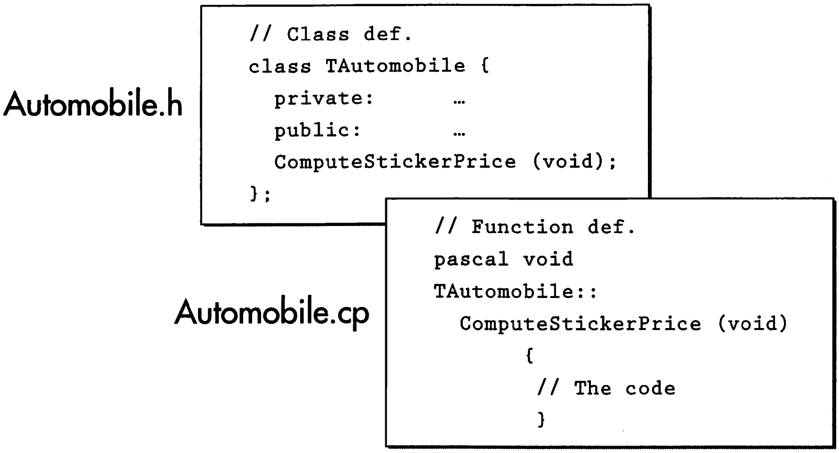
12 Phonebook Example

This PhoneBook example is a simple program that dials touch-tone phone numbers through the speakers of the Macintosh. The purpose of the example is to show you how to deal with some practical considerations. Before writing any C++ code, you must take into account:

- Source file organization
- Development environment

Source File Organization

When starting out, put class definitions in one file and function definitions in another. Put each class or family of classes in its own files, as shown in Figure 12.1.



```
// Class def.
class TAutomobile {
  private:      …
  public:       …
  ComputeStickerPrice (void);
};
```

```
// Function def.
pascal void
TAutomobile::
  ComputeStickerPrice (void)
       {
        // The code
        }
```

Figure 12.1. Example of Source File Organization

Naming Conventions

Here are the accepted C++ rules for naming among Macintosh programmers:

- Class names begin with "T"
- Data members begin with "f"
- Initialization functions are the class name preceded with the letter "I"
- Variables begin with a lowercase letter, while functions begin with an uppercase letter.
- Global variables begin with "g"
- Constant variables begin with "k"

The following are file naming conventions and definitions.

.cp	C++ source files
.cp.o	compiled object code from C++
.c	C source files
.c.o	compiled object code from C
.a	assembler code
.a.o	assembled code
.p	Pascal source files
.p.o	compiled object code from Pascal
.h	header files
.r	text file description of resources *(rez)*
.R	text file description of resources (*RMaker*)
.rsrc	compiled resources (*rez, RMaker, ResEdit*)

Class Diagrams

The first thing we need to show for the example are the class and subclasses, their members and functions that we'll be working with. Figure 12.2 shows the class diagrams for the PhoneBook example.

PhoneBook Project

The PhoneBook example is a project that contains a number of files, which are described in the following paragraphs. The project folder, which is titled PhoneBook.*f*, contains these files: *PhoneBook.π, UString.h, UString.cp, UTelephone.h, UTelephone.cp, UPhoneBook.h, UPhoneBook.cp,* and *PhoneBook.cp,* which contains the *main function.*

UString.h

After setting up the class diagrams, the next step is to create the header files. Header files (*.h files*) are used primarily to store forward declarations, including external variables, function prototypes, class definitions, and *inline* functions. Figure 12.3 shows the makeup of *UString.h.*

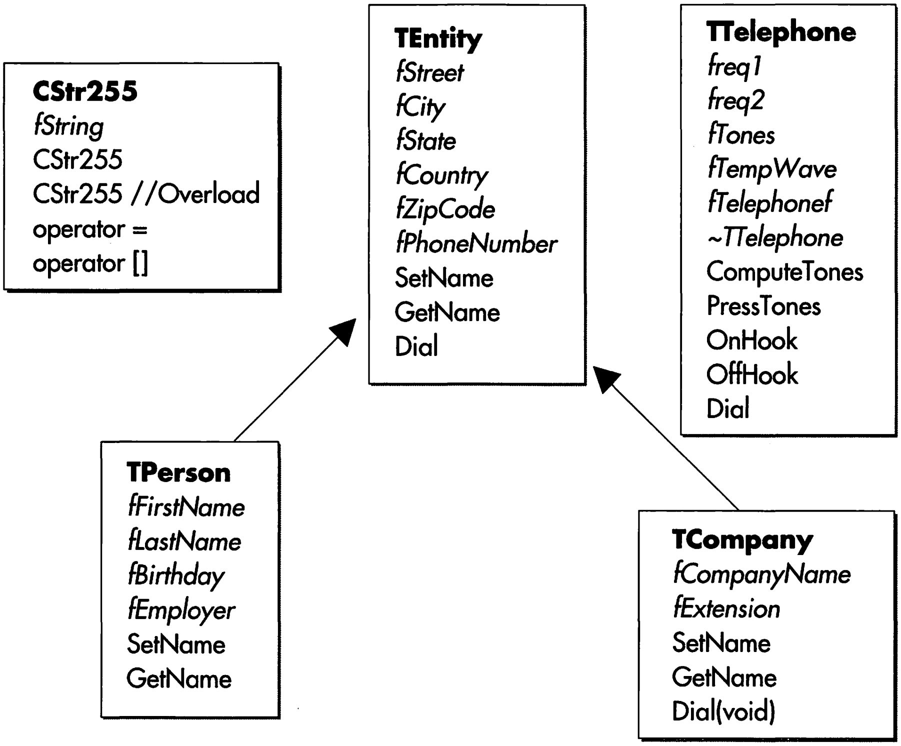
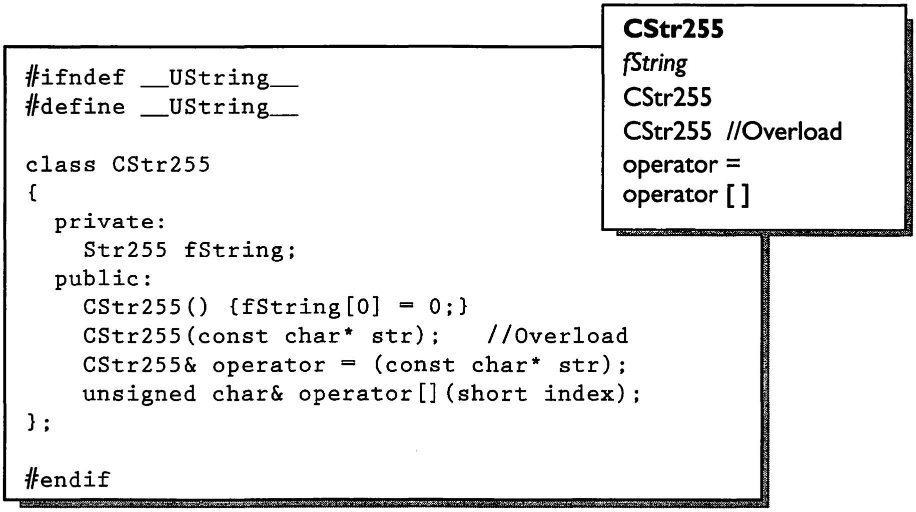


Figure 12.2. *Class Diagrams for PhoneBook Example*



```
#ifndef __UString__
#define __UString__

class CStr255
{
  private:
    Str255 fString;
  public:
    CStr255() {fString[0] = 0;}
    CStr255(const char* str);   //Overload
    CStr255& operator = (const char* str);
    unsigned char& operator[](short index);
};

#endif
```

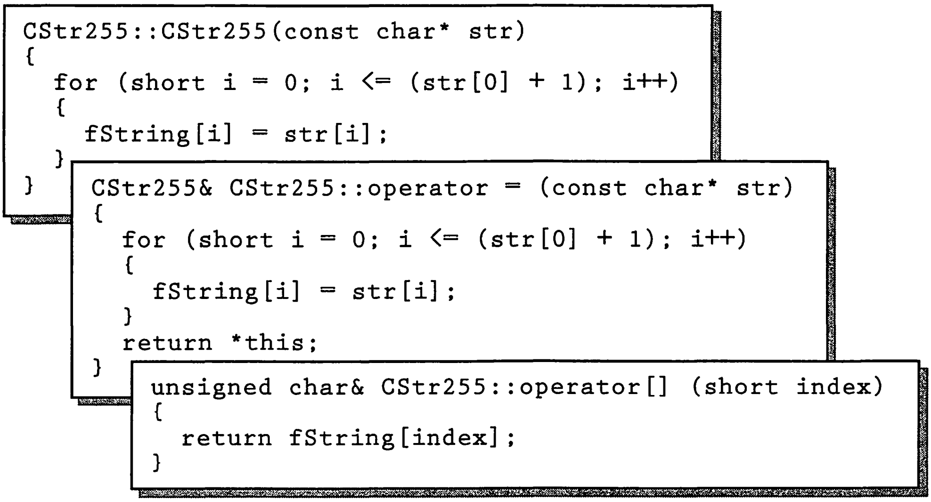
Figure 12.3. *UString.h Header File*

In Figure 12.3, we have used the preprocessor commands *#ifndef_UString_* and *#define_UString_* to mean, "If *UString* has not already been defined, then define it." If by some chance this file should be included again, and it has already been read in once, then the preprocessor commands will ensure that it is not read in twice.

We have created a *CStr255* class in order to avoid having to deal with arrays, which require each character in the string to be copied in individually. The only piece of data in the class that is private is the string itself, which is called fString. In the public area, there are two constructors, one (the first, which is inline) to create the length of the *fString* to be zero and one to be called if the string is set to a specific length. The third method overrides the = operator and the fourth overrides the [] operator.

UString.cp

Figure 12.4 shows the three non-inline methods for class *CStr255* contained in the file UString.cp.



```
CStr255::CStr255(const char* str)
{
  for (short i = 0; i <= (str[0] + 1); i++)
  {
    fString[i] = str[i];
  }
}
```

```
CStr255& CStr255::operator = (const char* str)
{
  for (short i = 0; i <= (str[0] + 1); i++)
  {
    fString[i] = str[i];
  }
  return *this;
}
```

```
unsigned char& CStr255::operator[] (short index)
{
  return fString[index];
}
```

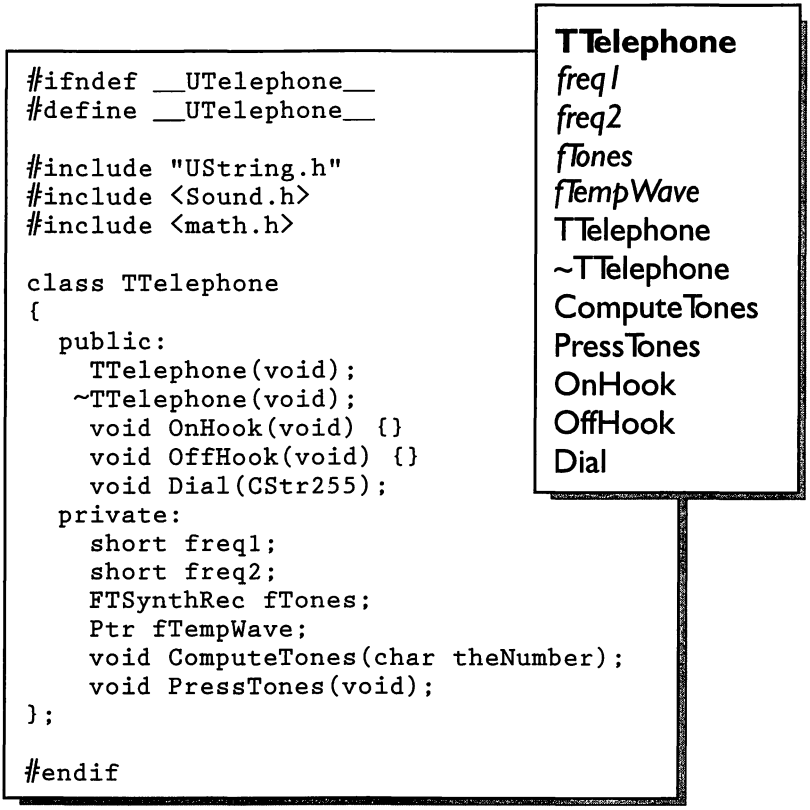
Figure 12.4. *Three member functions for the class CStr255, including its constructor*

The first method, which is actually the second constructor, takes each of the characters in the string that is passed in, hangs them in a loop, and copies each into *fString*. This is useful when you want to declare and initialize a *CStr255* in a statement, such as:

```
CStr255 theString = "Macintosh";
```

The second function overrides the = operator and does essentially the same thing as the first method. Wherever you have written in your code *x* = "some string", for example, the function will copy the string into *fString*.

The third method, which overrides the bracket operator, is passed in an index, and must return a character. To do this, we return fString indexed by the index to implement the bracket operator.



```
#ifndef __UTelephone__
#define __UTelephone__

#include "UString.h"
#include <Sound.h>
#include <math.h>

class TTelephone
{
  public:
    TTelephone(void);
   ~TTelephone(void);
    void OnHook(void) {}
    void OffHook(void) {}
    void Dial(CStr255);
  private:
    short freq1;
    short freq2;
    FTSynthRec fTones;
    Ptr fTempWave;
    void ComputeTones(char theNumber);
    void PressTones(void);
};

#endif
```

TTelephone
freq1
freq2
fTones
fTempWave
TTelephone
~TTelephone
ComputeTones
PressTones
OnHook
OffHook
Dial

Figure 12.5. *Defines, Includes, and Public and Private Members of Class TTelephone*

UTelephone.h

UTelephone.h, which is another header file, shows the forward declarations and *public* and *private* members of class *TTelephone* in Figure 12.5.

At the very top of the file, we include *UString.h, Sound.h* library from the Toolbox, and the *math.h* library (because we will be making computations). It is not really necessary to understand exactly what is going on here. In fact, that is one of the joys of object-oriented programming: things inside the black box (encapsulations) actually work.

In the *public* area, *TTelephone* is a constructor that creates a waveform in memory, and *~TTelephone* is a destructor that removes it from memory. The functions *OnHook* and *OffHook* are not actually used in this example (because the tones play through the Mac's speakers), but you could modify them to dial through the serial port connected to a modem. In that case, the modem would pick up the phone and dial the number, then replace the phone on-hook. The *Dial* routine is specific to this class.

In the *private* area, *freq1* and *freq2* are two variables that make up one tone on a touch-tone telephone. These must be calculated for each number. Playing the tones through the computer speaker requires two records: a sound record and a temporary waveform record. There are also two methods that no other class will use: *ComputeTones* and *PressTones..* The first function will compute the tone when two frequencies are passed in, and the second routine plays the tone. The *Dial* function actually takes a number and calls both *ComputeTones* and *PressTones.*

UTelephone.cp

The next step in the project development process is to write all of the methods for class *TTelephone.* These are shown in Figures 12.6 through 12.8.

The first method, shown in Figure 12.6, is the constructor *TTelephone,* which sets up a wave form that can be played through the speaker. The tone that will be playing is a sinusoidal tone, so we create a sampled wave form in memory that is a sinusoid. We then change the rate at which it is played back so that we can change the pitch of the sinusoid.

In order to create the sinusoid, we create a sound record (see Volume 6 of Inside Macintosh for details on the sound manager) that the sound synthesizer needs. We also need the temporary waveform with 256 bytes and a computation of *pi.* Next, we have a *for loop* from 0 to 255 that computes *theta* . We take the sine of that, multiply it by 127 (which is an amplitude), and add that to 128,

```
TTelephone::TTelephone()
{
  short i;
  double pi, theta;

  fTones.sndRec = (FTSndRecPtr)NewPtr(sizeof(FTSoundRec));
  fTempWave = NewPtr(256);
  pi = 4.0 * atan(1.0);
  for (i = 0; i <= 255; i++)
  {
    theta = double(i) * 2.0 * pi / 256.0;
    fTempWave[i] = 128 + char(127.0 * sin(theta));
  }
  fTones.mode = ftMode;
  fTones.sndRec->duration = 0;
  fTones.sndRec->sound1Wave = (WavePtr)fTempWave;
  fTones.sndRec->sound2Wave = (WavePtr)fTempWave;
  fTones.sndRec->sound3Wave = 0L;
  fTones.sndRec->sound4Wave = 0L;
  fTones.sndRec->sound1Rate = 0;
  fTones.sndRec->sound2Rate = 0;
  fTones.sndRec->sound3Rate = 0;
  fTones.sndRec->sound4Rate = 0;
  fTones.sndRec->sound1Phase = 0;
  fTones.sndRec->sound2Phase = 0;
  fTones.sndRec->sound3Phase = 0;
  fTones.sndRec->sound4Phase = 0;
}
```

```
TTelephone::~TTelephone()
{
    DisposePtr((Ptr)fTones.sndRec);
    DisposePtr(fTempWave);
}
```

Figure 12.6. *Methods for UTelephone.cp*

which is an offset. We then put that in the temporary wave form. If you plotted the numbers, they would go up (positive) and down (negative) to make one cycle of a sinusoidal waveform.

After computing the waveform, we set up the sound record so that the sound synthesizer knows what is going on. First, we set the tones in the Mac's four-tone mode and the duration of the tones (how long they will play in fractions of a second). Sound waves 1 and 2 are set to be pointers to the temporary waveform. Sound waves 3 and 4 are set to nothing, and the rest of the sound rates and phases are also set to zero.

The destructor deletes the wave form and the sound record from memory.

Figure 12.7 shows the *switch* statement for computing the tone from a single digit.

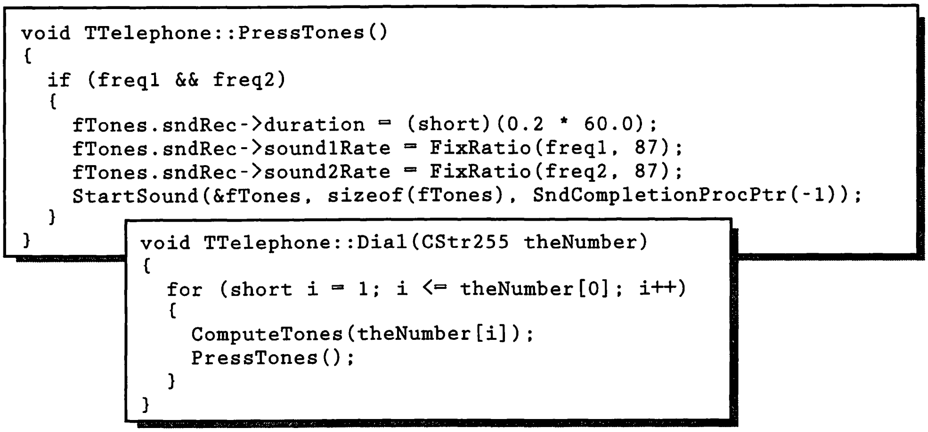
The first case is for digits *1, 2, 3,* and *A* (which is at the top of an additional column of tones put in by the phone company). The frequency for any of those digits is 697. The frequency for the second row *(4, 5, 6,* and *B)* is 770. For the third row (*7, 8, 9,* and *C*), it is 852, and for the fourth row (**, 0, #,* and *D*) it is 941.

```
void TTelephone::ComputeTones(char theNumber)
{
  switch (theNumber)
  {
    case ('1'):case ('2'):case ('3'):case ('A'):
      freq1 = 697;
      break;
    case ('4'):case ('5'):case ('6'):case ('B'):
      freq1 = 770;
      break;
    case ('7'):case ('8'):case ('9'):case ('C'):
      freq1 = 852;
      break;
    case ('*'):case ('0'):case ('#'):case ('D'):
      freq1 = 941;
      break;
    default:
      freq1 = 0;
      break;
  }
  switch (theNumber)
  {
    case ('1'):case ('4'):case ('7'):case ('*'):
      freq2 = 1209;
      break;
    case ('2'):case ('5'):case ('8'):case ('0'):
      freq2 = 1336;
      break;
    case ('3'):case ('6'):case ('9'):case ('#'):
      freq2 = 1477;
      break;
    case ('A'):case ('B'):case ('C'):case ('D'):
      freq2 = 1633;
      break;
    default:
      freq2 = 0;
      break;
  }
}
```

Figure 12.7. *Switch Statement for UTelephone.cp Methods*

The cases in the lower *switch* statement are for the columns, starting with the first column — *1, 4, 7,* and *. The frequency for those digits is 1209. For the second column (*2, 5, 8,* and *0*) it is 1336, for the third column (*3, 6, 9,* and #) it is 1477, and for the fourth (*A, B, C,* and *D*) it is 1633.

The number *2*, for example, would have the frequencies 697 and 1336. Now that we have computed the tones, the next step is to play them. This function is shown in Figure 12.8.



```
void TTelephone::PressTones()
{
  if (freq1 && freq2)
  {
    fTones.sndRec->duration = (short)(0.2 * 60.0);
    fTones.sndRec->sound1Rate = FixRatio(freq1, 87);
    fTones.sndRec->sound2Rate = FixRatio(freq2, 87);
    StartSound(&fTones, sizeof(fTones), SndCompletionProcPtr(-1));
  }
}
```

```
void TTelephone::Dial(CStr255 theNumber)
{
  for (short i = 1; i <= theNumber[0]; i++)
  {
    ComputeTones(theNumber[i]);
    PressTones();
  }
}
```

Figure 12.8. *Method for Creating Dial Tones in UTelephone.cp*

This function says, "If frequency *1* and frequency *2* do not equal *0*, then play, for two-tenths of a second, at a specific rate (via a Toolbox routine called *FixedRatio*) for each tone, each sound." (The speed at which the waveform is played back determines the frequency.) We call *StartSound* and pass it the records. It then plays the tone.

In the *Dial* routine, we have a for loop where i starts with 1 and loops through every character of the number, then computes the tones and plays them.

UPhoneBook.h

UPhoneBook.h is a header file that contains the *#includes* and *#defines*, as well as the structures of all three classes used in the example. Figure 12.9 shows the first part of the header file. Each class is described separately in the following paragraphs.

Again, the preprocessor commands *#ifndef_UPhoneBook_* and *#define_UPhoneBook_* ensure that if *UPhoneBook* has not been defined, it will be defined. And, if it has already been read in, it will not be read in a second time.

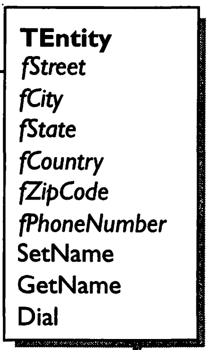
Note that we include both *UString.h* and *UTelephone.h* because we will be using them later on.

Class TEntity

Class *TEntity* , shown in Figure 12.10, is the parent class from which the other two classes inherit.

```
#ifndef __UPhoneBook__
#define __UPhoneBook__

#include "UString.h"
#include "UTelephone.h"

include TEntity class here
include TPerson class here
include TCompany class here

#endif
```

Figure 12.9. *Defines and Includes for UPhoneBook.h*



```
class TEntity
{
  protected:
    CStr255 fStreet;
    CStr255 fCity;
    CStr255 fState;
    CStr255 fCountry;
    CStr255 fZipCode;
    CStr255 fPhoneNumber;

  public:
    void SetName() {}
    void GetName() {}
    void SetStreet(CStr255 theStreet) {fStreet = theStreet;}
    void GetStreet(CStr255 theStreet) {theStreet = fStreet;}
    void SetCity(CStr255 theCity) {fCity = theCity;}
    void GetCity(CStr255 theCity) {theCity = fCity;}
    void SetState(CStr255 theState) {fState = theState;}
    void GetState(CStr255 theState) {theState = fState;}
    void SetCountry(CStr255 theCountry) {fCountry = theCountry;}
    void GetCountry(CStr255 theCountry) {theCountry = fCountry;}
    void SetZipCode(CStr255 theZipCode) {fZipCode = theZipCode;}
    void GetZipCode(CStr255 theZipCode) {theZipCode = fZipCode;}
    void SetPhoneNumber(CStr255 theNumber) {fPhoneNumber = theNumber;}
    void GetPhoneNumber(CStr255 theNumber) {theNumber = fPhoneNumber;}
    virtual void Dial(void);
};
```

Figure 12.10. *TEntity Class within UPhoneBook.h*

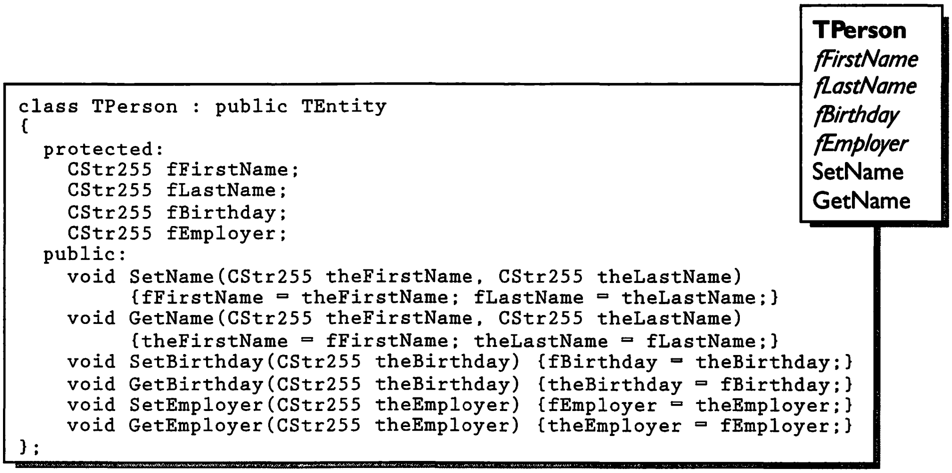
In the protected area we created six fields: *fStreet*, *fCity*, *fState*, *fCountry*, *fZipCodel*, and *fPhoneNumber.*

The public area has fourteen inline accessors — *SetName* through *GetPhoneNumber*. In *SetName* and *GetName* we do not pass in anything. In each of the other accessors, we pass in a string and make it equal to the name of the variable, such as *theStreet* or *theNumber* (for phone number). In addition, there is the prototype of the virtual function *Dial.*

Class TPerson

Class *TPerson* inherits everything from *TEntity* but adds a first name, last name, birthday and employer. It is shown in Figure 12.11.

Each of the fields in class *TPerson* has an accessor to set it and to get it (so that you could print out the whole set of fields if you wished).



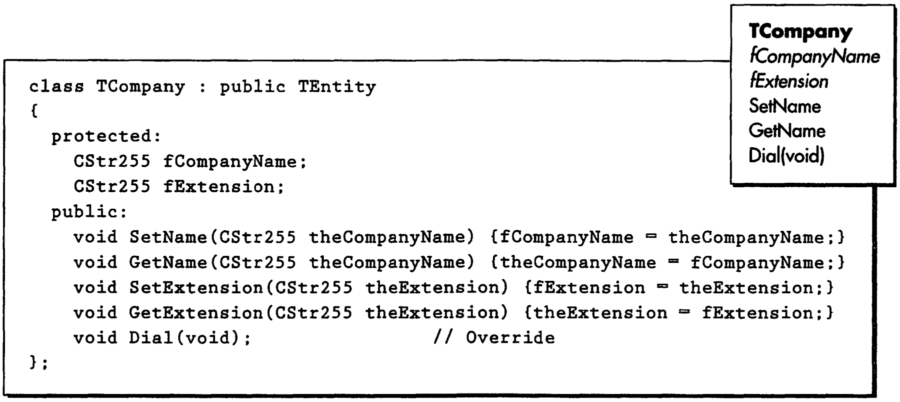
```
class TPerson : public TEntity
{
  protected:
    CStr255 fFirstName;
    CStr255 fLastName;
    CStr255 fBirthday;
    CStr255 fEmployer;
  public:
    void SetName(CStr255 theFirstName, CStr255 theLastName)
         {fFirstName = theFirstName; fLastName = theLastName;}
    void GetName(CStr255 theFirstName, CStr255 theLastName)
         {theFirstName = fFirstName; theLastName = fLastName;}
    void SetBirthday(CStr255 theBirthday) {fBirthday = theBirthday;}
    void GetBirthday(CStr255 theBirthday) {theBirthday = fBirthday;}
    void SetEmployer(CStr255 theEmployer) {fEmployer = theEmployer;}
    void GetEmployer(CStr255 theEmployer) {theEmployer = fEmployer;}
};
```

Figure 12.11. *TPerson Class within UPhoneBook.h*

Class TCompany

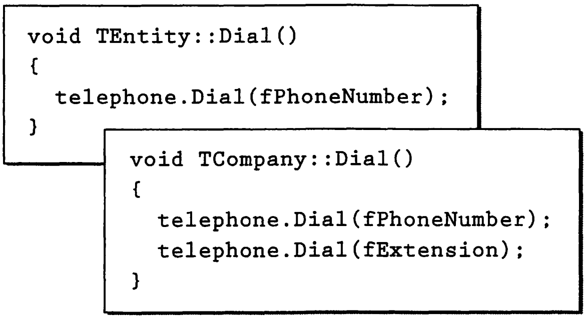
Class *TCompany* also inherits from *TEntity*. Class *TCompany* is shown in Figure 12.12.

In addition to the inherited fields, *TCompany* adds two strings: *fCompanyName* and *fExtension*. It also overrides the *Dial* function so that it not only dials the number but also the extension.



```
class TCompany : public TEntity
{
  protected:
    CStr255 fCompanyName;
    CStr255 fExtension;
  public:
    void SetName(CStr255 theCompanyName) {fCompanyName = theCompanyName;}
    void GetName(CStr255 theCompanyName) {theCompanyName = fCompanyName;}
    void SetExtension(CStr255 theExtension) {fExtension = theExtension;}
    void GetExtension(CStr255 theExtension) {theExtension = fExtension;}
    void Dial(void);                // Override
};
```

Figure 12.12. *TCompany Class within UPhoneBook.h*



```
void TEntity::Dial()
{
  telephone.Dial(fPhoneNumber);
}
```

```
void TCompany::Dial()
{
  telephone.Dial(fPhoneNumber);
  telephone.Dial(fExtension);
}
```

Figure 12.13. *Methods for UPhoneBook.h*

UPhoneBook.cp

UPhoneBook.cp shows the written method for dialing a phone number and the overridden method for dialing a company number and extension.

PhoneBook.cp

PhoneBook.cp contains the *main* function for the project. Figure 12.14 shows the code for *PhoneBook.cp.*

Inside *main,* we declare and create memory on the heap for Barney Rubble of class *TPerson* and for AcmeMerchandise of class *TCompany.* Next, we set the name and address for the personal number and the name, address and extension for the company number. We then dial the personal number and the companynumber and extension, with a pause in between.

```
main()
{
  TPerson *pBarneyRubble = TPerson;
  TCompany *pAcmeMerchandise = new TCompany;

  pBarneyRubble->SetName(*(CStr255*)"\pBarney", *(CStr255*)"\pRubble");
  pBarneyRubble->SetStreet(*(CStr255*)"\p15 Rockport Rd");
  pBarneyRubble->SetCity(*(CStr255*)"\pBedrock");
  pBarneyRubble->SetState(*(CStr255*)"\pCA");
  pBarneyRubble->SetZipCode(*(CStr255*)"\p94526");
  pBarneyRubble->SetPhoneNumber(*(CStr255*)"\p510-555-1212");

  pAcmeMerchandise->SetName(*(CStr255*)"\pAcme Merchandise Co.");
  pAcmeMerchandise->SetStreet(*(CStr255*)"\p128 Warner Brother's Road");
  pAcmeMerchandise->SetCity(*(CStr255*)"\pToon Town.");
  pAcmeMerchandise->SetState(*(CStr255*)"\pCA");
  pAcmeMerchandise->SetZipCode(*(CStr255*)"\p94583");
  pAcmeMerchandise->SetPhoneNumber(*(CStr255*)"\p415-555-1212");
  pAcmeMerchandise->SetExtension(*(CStr255*)"\p99");

  pBarneyRubble->Dial();
  Pause();
  pAcmeMerchandise->Dial();

  return 0L;
}
```

***Figure 12.14**. Main Function for PhoneBook.π*

Pause Routine

The code for the pause routine, which is used in *main,* is shown in Figure 12.15.

The pause routine pauses the program after dialing the personal number and waits for you to press the mouse button; it then dials the company number and extension.

```
void Pause()
{
  while(Button()){SystemTask();}
  while(!Button()){SystemTask();}
}
```

***Figure 12.15**. Pause Routine for PhoneBook.π*

Summary

This PhoneBook example emphasized the following features:

- Source file organization
- Naming conventions
- Examples using inheritance, methods, etc.

Exercises

1) Add a new person or company to the PhoneBook example

2) If you know how to use the Macintosh Toolbox, create snd resources for your phone book. Use the snds instead of the i/o stream data for each name.

13 Advanced Features of C++

Beyond the more obvious changes in C++ from C—classes, subclasses, inheritance, certain keywords, comments, and so on—there are several more advanced features. Although this book is a primer, and our objective is to teach the basics of C++ and Symantec C/C++, we feel that some of the advanced features are so useful that they need to be included. In this chapter, we take a look at *inline* functions, operator overloading, pointers and objects, polymorphism, templates, and multiple inheritance. We give some examples of when to use these features and, just as important, when not to use them.

Inline Functions in a Class

Inline functions can be placed inside a class definition. For example:

```
class TAutomobile
{
  private:
    char *fModel;
    long *fYear;
  protected:
    long *fStickerPrice;
  public:
    long GetYear (void) {return fYear;};
    void InData(void);
    void OutData(void);
};
```

Note:

***Inline* functions are an optimization that should be used sparingly, if at all. Note also that *inline* functions cannot be virtual.**

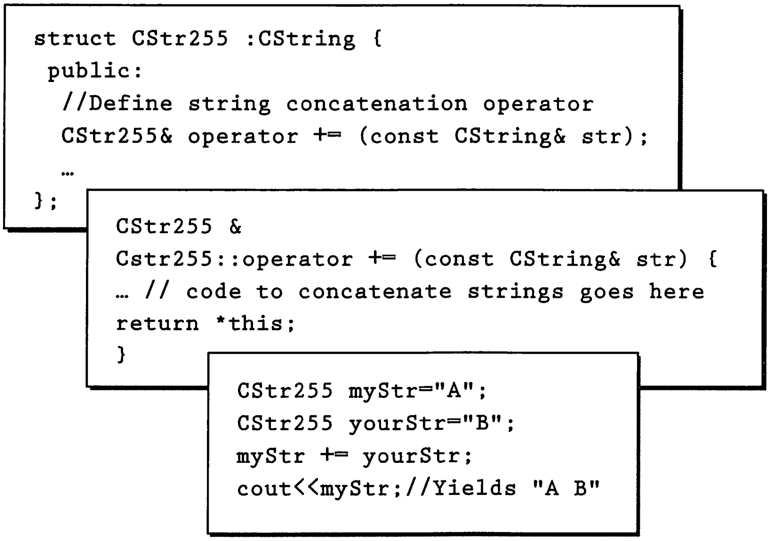
The reserved word *inline* is not required in the definition. The function's code is placed immediately after the function's declaration and is enclosed in curly braces. For inline member functions, the compiler will insert the inline code in each place that the member function is called. (It does not place it inline in the object but, rather, where it is called.)

Nonvirtual inline functions provide speed at the expense of code reusability. Use nonvirtual inline functions only when you are absolutely sure that you need every microsecond. In addition, you must be sure that no one will ever want to override the method in a derived class.

Operator Overloading in Classes

Operator overloading is a device that allows you to add new data types to those that C++ already handles. You can redefine standard operators for use with new classes. Redefinition consists of supplying a member function to be called when the operator is used. This is an advanced concept that can be useful for extending the language.

Figure 13.1 shows the code used in overloading operators.



```
struct CStr255 :CString {
 public:
  //Define string concatenation operator
  CStr255& operator += (const CString& str);
  …
};
```

```
CStr255 &
Cstr255::operator += (const CString& str) {
… // code to concatenate strings goes here
return *this;
}
```

```
CStr255 myStr="A";
CStr255 yourStr="B";
myStr += yourStr;
cout<<myStr;//Yields "A B"
```

Figure 13.1 *Operator Overloading in Classes*

Pointers and Objects

Whenever a base class is a public base of a derived class, you can:

- Convert a pointer to a derived object to a pointer to a base object.
- Convert a reference to a derived object to a reference to a base object.
- Initialize a base object address to refer to a derived class.

Figures 13.2 and 13.3 demonstrate these techniques.

```
class TBase {…};
class TDerived1 : public TBase {…};
class TDerived2 : public TDerived1 {…};
TBase b, *pb, &rd;
TDerived1 d1, *pd1, &rd1;
TDerived2 d2, *pd2, &rd2;

pb = &d1; // OK, base ptr set to a derived ptr.
pd1 = &d2;
pb = &d2;
```

Figure 13.2 *Setting Base Object Pointers to Derived Class Pointers*

In Figure 13.2, we declare the three classes — *TBase, TDerived1,* and *TDerived2.* In each class, we declare a class object, a pointer to the class object, and a reference to the class object. As you can see from the figure, we can set the pointer to the base object (*pb*) to an address of *TDerived1.* We can also say that a pointer to *TDerived1* contains an address of *TDerived2* and a pointer to the base also contains the address of *TDerived2.* The same thing is true with the references in Figure 13.3.

An object of a derived class is an object of its base class; the opposite is not true. You cannot convert a pointer (or reference) to a base object to a pointer (or reference) to a derived object unless you cast it. This could be dangerous! Figure 13.4 demonstrates this principle.

In Figure 13.4, we have made the same declarations as we made in the previous two figures. However, in this case, we show that you cannot set the pointer (or reference) to *TDerived1* equal to a pointer (or reference) to the

```
class TBase {…};
class TDerived1 : public TBase {…};
class TDerived2 : public TDerived1 {…};
TBase b, *pb, &rd;
TDerived1 d1, *pd1, &rd1;
TDerived2 d2, *pd2, &rd2;

pb = &d1; // OK, base ptr set to a derived ptr.
pd1 = &d2;
pb = &d2;
```

Figure 13.3 *Setting Base Object References to Derived Class References*

```
class TBase {…};
class TDerived1 : public TBase {…};
class TDerived2 : public TDerived1 {…};
TBase b, *pb, &rb;
TDerived1 d1, *pd1, &rd1;
TDerived2 d2, *pd2, &rd2;

pd1 = pb;          //error
pd1 = (TDerived1*)pb;//OK, but suspect
rd1 = b;           //error
```

Figure 13.4 *Attempting to Set Derived Class Pointers or References to Base Class Pointers or References*

base, nor can you set a reference (or pointer) to *TDerived2* equal to a reference (or pointer) to the base. In other words, you cannot go back up the chain.

We also have a pointer to *TDerived1* and we cast it to be a pointer to the base. This will compile, but it is suspect.

Figure 13.5 explains the principle with instance variables.

In Figure 13.5, we have two classes: *TBase* and *TDerived.* In the *public* section of *TBase* we have two functions—*foo* and *goo*—and in the *public* section

```
class TBase
{
  public:
    short foo(void);
    void goo(short);
  private:
    short i, j;
};
```

```
class TDerived : public TBase
{
  public:
    float hoo(float);
  private:
    float x;
};
```

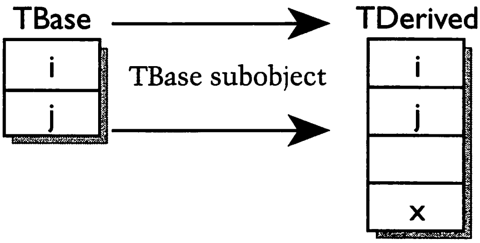


Figure 13.5 *Order of Memory Allocation for Inherited Objects*

of *TDerived,* we have the function *hoo.* In the *private* section of *TBase* are two *short* variables, *i* and *j.* In the *private* section of *TDerived,* we have a float variable, x. The diagram shows how *TDerived* makes room for the inherited variables (each box is 2 bytes), and it also shows that *TDerived* makes room for x (which is a *long* and takes up 4 bytes). *TDerived* makes room for the inherited variables first, and then makes room for its own. Therefore, there is no way that the base class can know anything about the derived class's objects.

Figure 13.6 shows even more graphically the areas that pointers for a base and two derived classes may access in each of the classes.

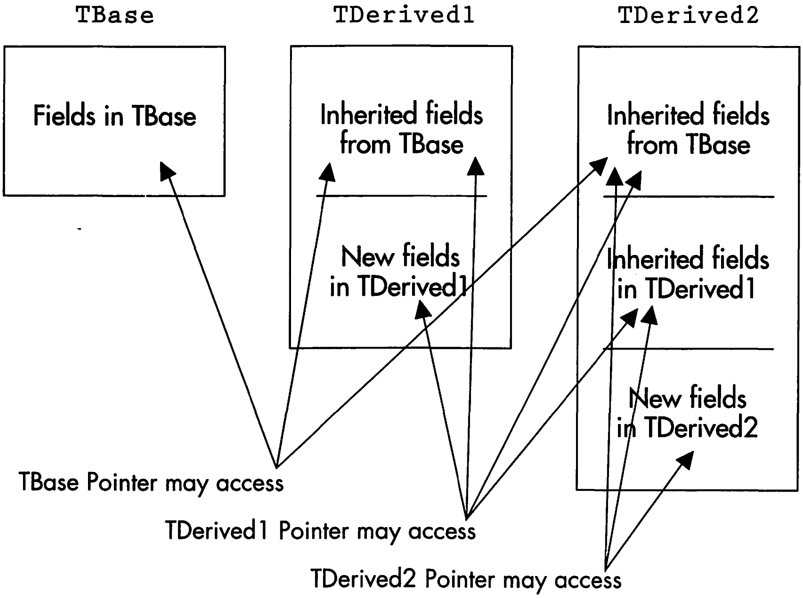

TBase
TDerived1
TDerived2
Fields in TBase
Inherited fields from TBase
New fields in TDerived1
Inherited fields from TBase
Inherited fields in TDerived1
New fields in TDerived2
TBase Pointer may access
TDerived1 Pointer may access
TDerived2 Pointer may access

Figure 13.6 *Fields That Pointers of a Base Class and Two Derived Classes May Access*

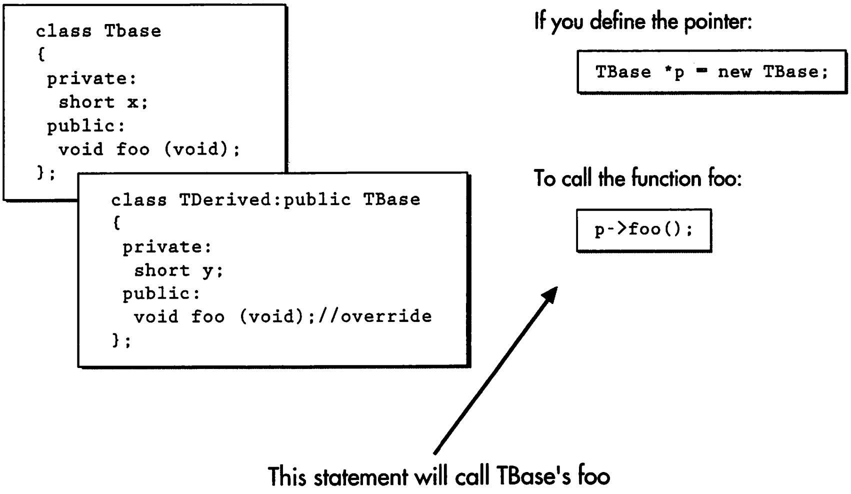
The following figures show what will happen in two given classes when you use pointers to call functions.

Given the following two classes:

```
class Tbase
{
 private:
  short x;
 public:
  void foo (void);
};
```

```
class TDerived:public TBase
{
 private:
  short y;
 public:
  void foo (void);//override
};
```

Figure 13.7 *Declaring a Base Class and a Derived Class*



```
class Tbase
{
 private:
  short x;
 public:
  void foo (void);
};
```

```
class TDerived:public TBase
{
 private:
  short y;
 public:
  void foo (void);//override
};
```

If you define the pointer:

```
TBase *p = new TBase;
```

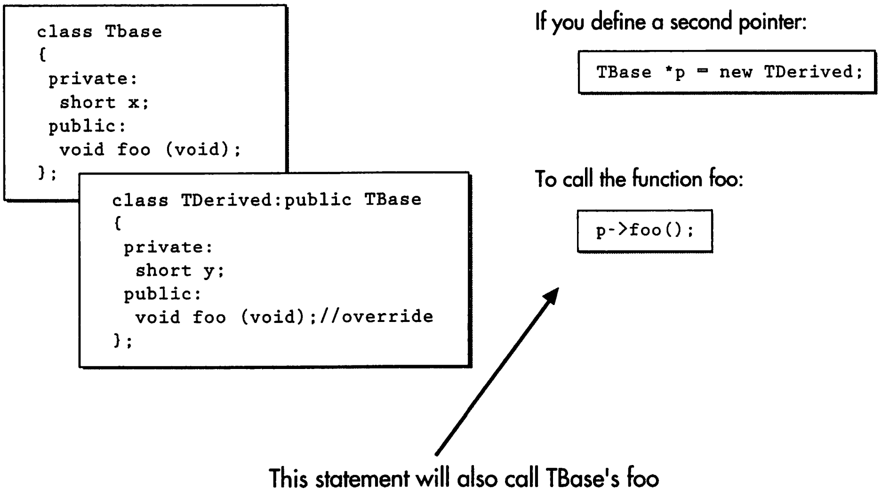
To call the function foo:

```
p->foo();
```

This statement will call TBase's foo

Figure 13.8 *Defining a Pointer to TBase*

In Figure 13.8, if we define *p* to be a pointer to *TBase* and set it equal to *new TBase, p->foo* will call the *foo* in *TBase*. This is not surprising.



```
class Tbase
{
 private:
  short x;
 public:
  void foo (void);
};
```

```
class TDerived:public TBase
{
 private:
  short y;
 public:
  void foo (void);//override
};
```

If you define a second pointer:

```
TBase *p = new TDerived;
```

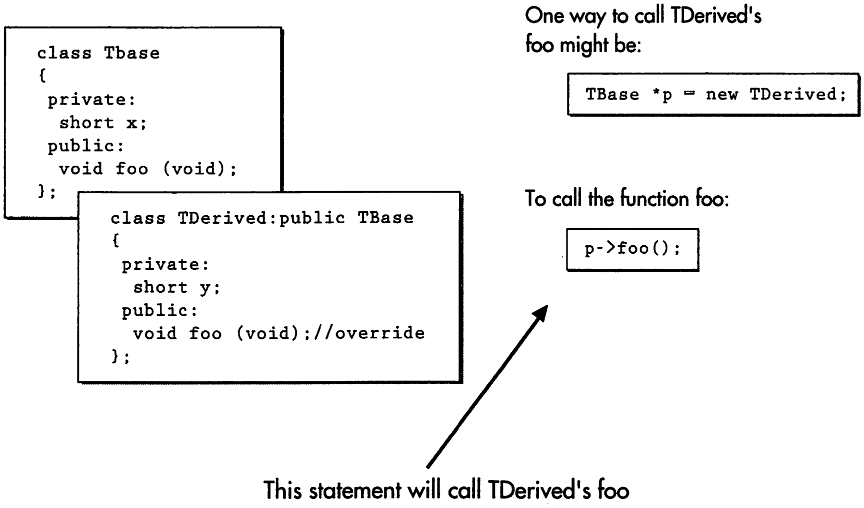
To call the function foo:

```
p->foo();
```

This statement will also call TBase's foo

Figure 13.9 *Defining a Second Pointer*

In Figure 13.9, if we define *p* to be a pointer to *TBase* and set it equal to *new TDerived, p->foo* will still call the *foo* in *TBase.*



```
class Tbase
{
 private:
  short x;
 public:
  void foo (void);
};
```

```
class TDerived:public TBase
{
 private:
  short y;
 public:
  void foo (void);//override
};
```

One way to call TDerived's foo might be:

```
TBase *p = new TDerived;
```

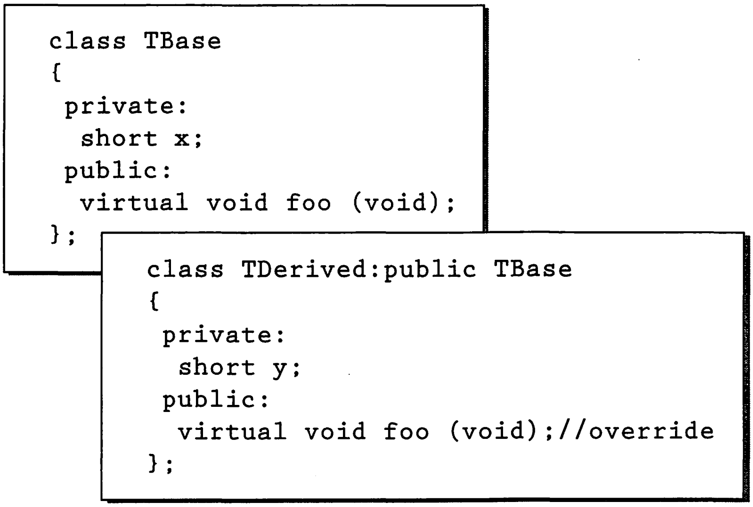
To call the function foo:

```
p->foo();
```

This statement will call TDerived's foo

Figure 13.10 *Calling TDerived's foo Function*

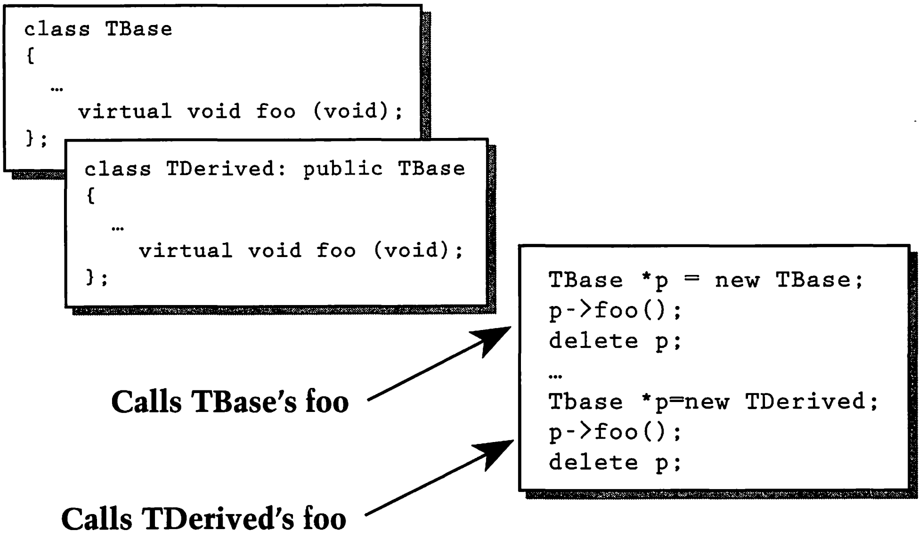
In Figure 13.10 we have to declare the type of an object's pointer in advance (compile time). However, suppose we do not know the type of an object pointer at compile time, or a single pointer addresses a list of objects, many of which are of derived classes? We just add the keyword *virtual* to the classes (Figure 13.11), and we have a new way to call *foo* in *TDerived.* This is polymorphism (Figure 13.12).



```
class TBase
{
 private:
  short x;
 public:
  virtual void foo (void);
};
```

```
class TDerived:public TBase
{
 private:
  short y;
 public:
  virtual void foo (void);//override
};
```

Figure 13.11 *Adding "virtual" to the Functions*



```
class TBase
{
  ...
   virtual void foo (void);
};
```

```
class TDerived: public TBase
{
  ...
   virtual void foo (void);
};
```

```
TBase *p = new TBase;
p->foo();
delete p;
...
Tbase *p=new TDerived;
p->foo();
delete p;
```

Figure 13.12 *Using Polymorphism*

In Figure 13.12, the type of the object pointed to by *p* is examined to select which function to invoke. Another example of calling *foo* in all three classes, this time with an array of pointers, is shown in Figure 13.13.

```
class TBase
{
 ...
  virtual void foo (void);
};
```

```
class TDerived1: public TBase
{
 ...
  virtual void foo (void);
};
```

```
class TDerived2: public TDerived1
{
 ...
  virtual void foo (void);
};
```

```
TBase *p[5];

p[0] = new TBase;
p[1] = new TDerived1;
p[2] = new TDerived2;
p[3] = new TDerived1;
p[4] = new TBase;

for (short i = 0; i<5; i++)
{
 i->foo();
}
```

Figure 13.13 *An Array of Pointers*

A container is a list or array normally used to contain objects.

In the example in Figure 13.13, *p* is an array of 5 pointers to *TBase.* We also have three different routines, because *TDerived1* overrides the *foo* in *TBase,* and *TDerived2* also overrides the *foo* in *TDerived1*. In the *for* loop, we set *i* to *0,* and while *i* is less than 5, we increment *i* each time through. When we then call *i->foo,* the program will automatically call the right one. This is really an iterator function.

Shape Example

The following Shape example, which is based on the List example in Chapter 10, illustrates polymorphism and the iterator function. The purpose of the example is simply to draw several shapes (rectangle, round rectangle, circle, triangle, and an X) on the screen with one call (iterator function).

We start with the *main* function, *Shapes.cp,* which follows:

```
#include "UList.h"
#include "UShapes.h"
#include "MyLib.h"

/*****************************************************/
main()
/*****************************************************/
{
  TShapeList theShapes;

  InitToolBox();
  OpenWindow();

  theShapes.Add(new TRectangle(10, 10, 60, 60));
  theShapes.Add(new TRoundRectangle(70, 70, 120, 120));
  theShapes.Add(new TOval(130, 130, 180, 180));
  theShapes.Add(new TTriangle(190, 190, 240, 240));
  theShapes.Add(new TXRect(250, 250, 300, 300));
  theShapes.DrawAll();

  Pause();
  return 0;
}
```

At the very beginning of the code, we declare the two *#include* files, which contain the functions that the *main* will call, and *MyLib.h*, which contains the *InitToolBox* (initializing the Toolbox), *OpenWindow* (this opens up a window on the screen titled *MyShapes*), and *Pause* (this keeps the window from going away immediately so that you can view what's in it). These are the *header* files. (Remember that in Symantec C++, you can view any header file connected with your source file by holding down the *option* key, clicking on the title of the source file, and clicking on the header file you wish to open.)

Inside the *main, TShapeList* is declared as the *Shapes*, and the calls are made to initialize the Toolbox and open the window in which to draw the shapes. Next, we add the various shapes and their size, which also includes the position where each will appear on the screen (the top left and bottom right points in pixels). Then we request the function to draw all the shapes.

After the shapes appear on the screen, they will remain there until the mouse is clicked inside the window area (*Pause()*). Note that in C++ we must return a *0* (nil) at the end of the routine; otherwise, we will get an error.

The next file to look at is *UList.h*, which is the header file for *UList.cp*, which in turn is derived from *TList* in Chapter 10.

```
#ifndef __UList__
#define __UList__

/*******************************************************/
class TNode                    // Node Definition
/*******************************************************/
{
  private:
    friend class TList;
    TNode *next;
    TNode *prev;
};
/*******************************************************/
class TList                    // List Definition
/*******************************************************/
{
  public:
    TList(void) {s1 = 0; s2 = 0;}
    virtual ~TList(void);
    void Add(TNode *n);
    void Remove(TNode *n);
    void* Next(TNode *n) {if (n == 0) return(s1);
        else return (n->next);}
    void* Prev(TNode *n) {if (n == 0) return(s2);
        else return (n->prev);}
  private:
    TNode *s1;
    TNode *s2;
};
#endif
```

Included in this header file is the notation *#ifndef __UList__* which tells the compiler to define the file if it is not already defined. Inside the file, we have *class TNode,* which is a friend of *class TList,* and the next and previous nodes (*TNode *next* and *TNode *prev*). Note that all attributes are *private.*

The second part of the file, *class TList,* has an inline constructor with two start pointers (*s1* and *s2*). One starts at the head and goes all the way through, and the other starts at the end and goes to the beginning. Those are both set to *0* because there is nothing in the list (so they are pointing to nothing). There is also a destructor, *virtual ~TList(void);* ,which clears out everything in memory.

TList also includes *Add* and *Remove* prototypes and an inline routine called *Next.* In this routine, if we pass a *0* to *Next,* it will return *s1,* which is a pointer to the first node. If we pass it any other number, it will return a pointer to the next node. The inline routine *Prev* does the same thing in reverse. If we pass it a 0, it will return *s2,* which is a pointer to the last node. *TList* also includes the *private* variables *TNode *s1* and *TNode *s2,* which are both pointers to *TNode.*

Note the source code for the add, remove, and destructor functions found in *UList.cp,* which follows:

```
#include "UList.h"

/*****************************************************/
void TList::Add(TNode *n) // Add node to list
/*****************************************************/
{
  if (!s1 || !s2)
  {
    n->next = 0;
    s1 = n;
    n->prev = 0;
    s2 = n;
  }
  else
  {
    n->next = s1;
    s1->prev = n;
    n->prev = 0;
    s1 = n;
  }
}
/******************************************************/
void TList::Remove(TNode *n)         // Remove node from
             // list and delete memory
/******************************************************/
{
  TNode *i;
  i = s1;
  while (i)
  {
    if (i == n)
    {
      if (!i->next)
      {
        i->prev->next = 0;
        s2 = i->prev;
      }
      else if (!i->prev)
      {
        s1 = i->next;
        i->next->prev = 0;
      }
      else
      {
        i->prev->next = i->next;
        i->next->prev = i->prev;
      }
    }
    i = i->next;
  }
  delete [] n;
}
/******************************************************/
TList::~TList()               // TList Destructor
/******************************************************/
{
  TNode *p1, *p2;
  p1 = s1;
  if (!s1)
  {
    return;
  }
  while (p1)
  {
    p2 = p1->next;
    delete p1;
    p1 = p2;
  }
}
```

Next, we look at the header file *UShapes.h*, which is also derived from *TList*.

```
#ifndef __UShapes__
#define __UShapes__
#include “UList.h”
```

```
/*****************************************************/
class TShapeList : public TList
/*****************************************************/
{
  public:
    void DrawAll(void);
};
/*****************************************************/
class TShape : public TNode
/*****************************************************/
{
  protected:
    Rect fRect;
  public:
    TShape(short top, short left, short bottom, short
                        right);
    virtual void Draw(void) = 0;
};
/*****************************************************/
class TRectangle : public TShape
/*****************************************************/
{
  public:
    TRectangle(short top, short left, short bottom,
        short right) : (top, left, bottom, right){};
    virtual void Draw(void);
};
/*****************************************************/
class TRoundRectangle : public TShape
/*****************************************************/
{
  public:
    TRoundRectangle(short top, short left, short bottom,
       short right) : (top, left, bottom, right){};
    virtual void Draw(void);
};
/******************************************************/
class TOval : public TShape
/******************************************************/
{
  public:
    TOval(short top, short left, short bottom, short
       right) : (top, left, bottom, right){};
    virtual void Draw(void);
};
/******************************************************/
class TTriangle : public TShape
/******************************************************/
{
  public:
    TTriangle(short top, short left, short bottom, short
       right) : (top, left, bottom, right){};
    virtual void Draw(void);
};
/******************************************************/
class TXRect : public TShape
/******************************************************/
{
  public:
    TXRect(short top, short left, short bottom, short
       right) : (top, left, bottom, right){};
    virtual void Draw(void);
};

#endif
```

In this header file, we include the usual *#ifndef* statement, and we also include *"Ulist.h"*. We then create a *TShapesList*, which is inherited from *TList* in Chapter 10. We added one new method which we call *DrawAll.*

We now create *TShape,* which is an abstract class (we will never instantiate it directly). This is inherited from *TNode,* but we add a new instance variable: *fRect.* It also has a constructor, which sets the top left and bottom right of a rectangle, and a pure virtual function *void Draw(void) = 0.* This requires each of the shapes inherited from *TShape* to draw its own individual shape; that is, this is not a generic draw function.

Next, we create a rectangle inherited from *TShape,* and we create a constructor with the top, left, bottom, and right into which we pass the top, left, bottom, and right measurements. We also declare a virtual *Draw* override. We then do exactly the same thing for a round rectangle, an oval, a triangle and an X class.

The next piece of code that we need to deal with is*UShapes.cp,* which follows

;

```
#include "UShapes.h"

/*********************************************************/
void TShapeList::DrawAll()
/*********************************************************/
{
  TShape *p;

  p = Next(0);
  while (p)
  {
    p->Draw();
    p = Next(p);
  }
};

/*********************************************************/
```

```
TShape::TShape(short top, short left, short bottom,
                    short right)
/******************************************************/
{
  fRect.top = top;
  fRect.left = left;
  fRect.bottom = bottom;
  fRect.right = right;
}
/******************************************************/
void TRectangle::Draw()
/******************************************************/
{
  FrameRect(&fRect);
}


/******************************************************/
void TRoundRectangle::Draw()
/******************************************************/
{
  FrameRoundRect(&fRect, (fRect.right - fRect.left) / 4,
       (fRect.bottom - fRect.top) / 4);
}
/******************************************************/
void TOval::Draw()
/******************************************************/
{
  FrameOval(&fRect);
}
/******************************************************/
void TTriangle::Draw()
/******************************************************/
{
    MoveTo(fRect.left + ((fRect.right - fRect.left) / 2),
                       fRect.top);
    LineTo(fRect.left, fRect.bottom);
    LineTo(fRect.right, fRect.bottom);
    LineTo(fRect.left + ((fRect.right - fRect.left) / 2),
                       fRect.top);
  }
  /******************************************************/
  void TXRect::Draw()
  /******************************************************/
  {
    MoveTo(fRect.left, fRect.top);
    LineTo(fRect.right, fRect.bottom);
    MoveTo(fRect.right, fRect.top);
    LineTo(fRect.left, fRect.bottom);
  }
```

The first function (*DrawAll()*) in *UShapes.cp* allows us to draw all of the shapes. We have a *ShapesList* where *p* is a pointer to *TShape*. We get the first pointer in the list by saying *Next* and passing it *0*. Then, while *p* is not equal to *0*, draw. And then *p* is equal to the next *p*. This continues through the list for as many shapes as are in it; that is, when *p* is equal to *0*.

The constructor for *TShape* sets *fRect.top*, and so on, to the values that are passed in. After that, we draw the rectangle with *FrameRect(&fRect);*, the round rectangle with the expression under *void TRoundRectangle::Draw()*, the circle with the function under *void TOval::Draw()*, the triangle with the function under *void TTriangle::Draw()* and the X with the function under *void TXRect::Draw()*. Notice the computations necessary to draw a round rectangle and a triangle. Note that all the draw routines are done through the Toolbox.

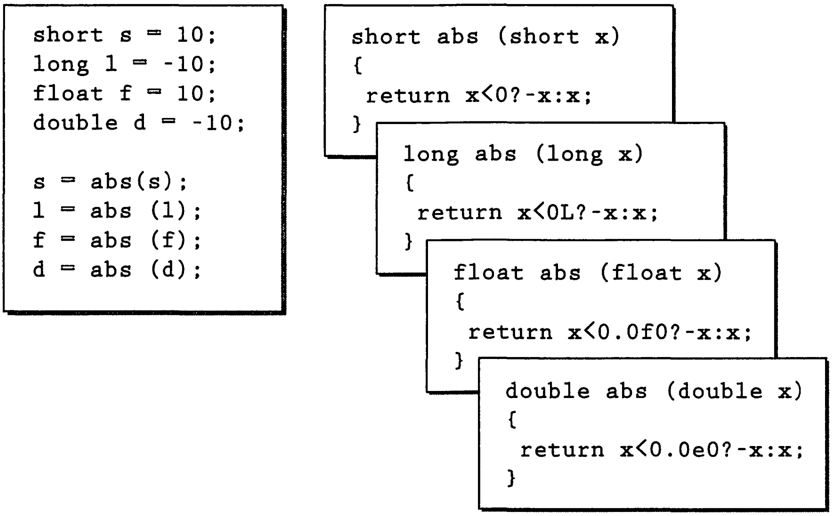
Adding another shape to the list is relatively simple. First we go into the *main*, where we need to change the source code. If you want to add another shape in this example, you would declare it (with its values) after *XRect*. Also, in *Shapes.cp*, you include a function to draw the shape that you are adding. The rest of the code remains valid.

Templates

Templates provide a way to parameterize types within a function or a class. To declare a template, you must place the *template* keyword and its associated parameter type list at the beginning of a function or class declaration. For example:

```
template <class Type> class MyClass
{
```

The types that will be parameterized are within the arrows.



```
short s = 10;
long l = -10;
float f = 10;
double d = -10;

s = abs(s);
l = abs (l);
f = abs (f);
d = abs (d);
```

```
short abs (short x)
{
 return x<0?-x:x;
}
```

```
long abs (long x)
{
 return x<0L?-x:x;
}
```

```
float abs (float x)
{
 return x<0.0f0?-x:x;
}
```

```
double abs (double x)
{
 return x<0.0e0?-x:x;
}
```

Figure 13.14 *Example of Absolute Value Routines*

In the example in Figure 13.14, we have created an absolute value routine for four variables using the ternary operator (?:). First we pass in a *short,* then overload it by passing in a *long,* then a *float,* and then a *double.* In each case we compare *x* to a *0* of the given type. (Note that we do not have to use the word *overload.* It is automatic.)

Template for a Routine

The problem with the routines in Figure 13.14 is that if we want to add anything else—say an extended *double* or an *int*—we will have to write additional routines. The way to solve that problem is to collapse all of those routines into a template, as shown in Figure 13.15.

```
short s = 10;
long l = -10;
float f = 10;
double d = -10;

s = abs(s);
l = abs (l);
f = abs (f);
d = abs (d);
```

```
template <class theType>

theType abs(theType x)
{
 return x <theType (0) ? -x :x;
}
```

Figure 13.15 *Collapsing the Code into a Template*

The code for the template in Figure 13.15 could all be on one line. Instead of naming each of the variables after we declare the template, we simply say *theType abs(theType x)*. We then cast *0* into *theType*. Now, when we call the absolute value routine, it will automatically call the correct one. It will look at the arguments, and when it finds the types that it needs, it will create an instance of this function in memory for the particular type. In the above example, it would create four instances of the routine.

Template for a Class

C++ also supports templates for classes. The keyword *class* indicates that the parameter may be a built-in or user-defined function. You can have more than one type in a template type list, but each type must be preceded by the word *class*, and the types are separated by commas. Figure 13.16 shows a template for the class *Array*.

The template class in Figure 13.16 could be very useful. In this example we have a class called *Array*. In the public section, we have a constructor into which we pass *short theSize*. We also set a protected variable *size* equal to the *theSize* that we are passing into the array, and we have *array* as some pointer to *theType*. We create an array of *theType* and whatever size we have on the heap. If *theType*

```
template <class theType>class Array
{
 public:
  Array (short theSize)
  {size=theSize;array=new theType [size]:}
 ~Array(){delete [] array;}
  theType & operator[] (short index)
  {
   if ((index<0) || (index > size))       error;
   else return array [index];
  }
 protected:
  short size;
  theType *array;
};
```

```
Array<short>s(10);
Array<double>d(132);

s[9] = 35535;
d[2] = 3.14;
```

Figure 13.16 *Template of the Class Array*

was a *short*, we would set the size by calling the constructor with *array(10)*. Then we would create an array of new shorts of 10 in size The destructor deletes all of the information in the array.

We then override the operator bracket ([]). The [] operator has one variable, an index, passed in. This will return the array that is indexed here.

To use this function, we declare *Array* (which is a class now) of shorts (s) that sets the size to be *10* and creates a pointer to an array of 10 shorts on the heap. We also create an array of 132 doubles. We set s[9] equal to a number (35535 here) and d[2] equal to a number (3.14). Next we say, "If the index is less than 0 or greater than size (10 in this case), then provide an error message of some sort. Otherwise, return the array." Since our index numbers are 2 and 9, the function would return the array.

What this function provides is a method of bounds checking that does not normally exist in C++. (Remember that you may declare an array of a number such as 10, but you can index it by any number because the compiler has no way of checking the bounds of the array.)

Note:

The parameterized type must appear at least once in the template function or class. Template functions can be overloaded, provided the arguments of each instance are distinguishable by type or number.

Multiple Inheritance

You may create classes in C++ that are derived from two or more base classes at the same time. This is known as *multiple inheritance*. The resulting class that is derived by multiple inheritance combines the properties of two or more ancestors. Figure 13.17 shows the class diagrams involved in multiple inheritance.

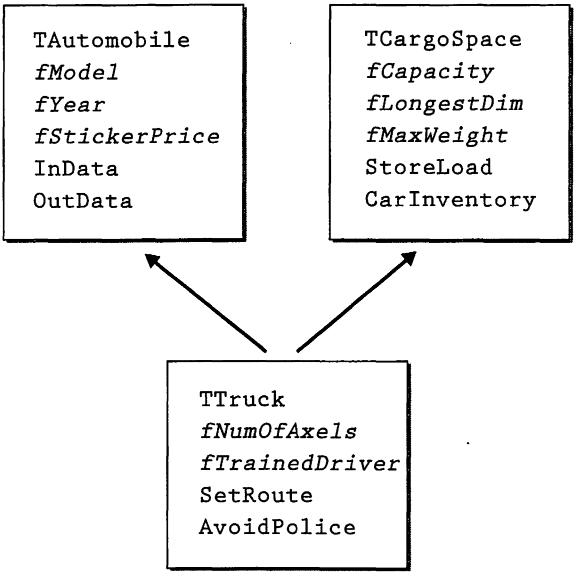


Figure 13.17 *Class Diagrams in Multiple Inheritance*

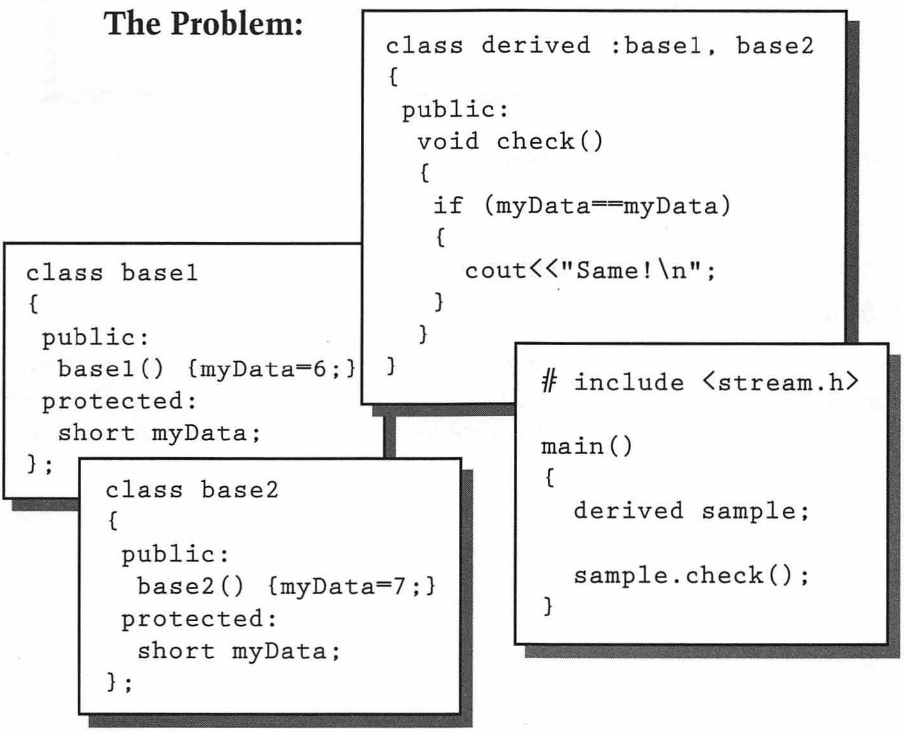
The following statement is used to define the class:

```
    class TDerived Class : TBaseClass1, TBaseClass2, …
```

Each base class may be given the identifier *public* or *private* (default).

Ambiguities of Data Members in Multiple Inheritance

A major problem with multiple inheritance is that data members or member functions in one base class might have the same name as a data member or function in another base class. If *base1* and *base2* both have a data member with the same name, then any attempt in the derived class to access this data member will result in a compiler error. Figure 13-18 illustrates the problem, and Figure 13.19 shows how to resolve it.



The Problem:

```
class base1
{
 public:
  base1() {myData=6;}
 protected:
  short myData;
};
```

```
class base2
{
 public:
  base2() {myData=7;}
 protected:
  short myData;
};
```

```
class derived :base1, base2
{
 public:
  void check()
  {
   if (myData==myData)
   {
     cout<<"Same!\n";
   }
  }
}
```

```
# include <stream.h>

main()
{
  derived sample;

  sample.check();
}
```

***Figure 13.18** Example of a Problem with Ambiguities in Data Members in Multiple Inheritance*

The Fix:

```
class derived :base1, base2
{
 public:
  void check()
  {
   if (base1::length==base2::length)
   {
     cout<<"Same!\n";
   }
  }
}
```

Figure 13.19 *The Solution to the Ambiguity of Data Members Problem in Multiple Inheritance*

Ambiguities of Member Functions in Multiple Inheritance

If *base1* and *base2* both have a member function with the same name, then any attempt in the derived class to call this member function will result in a compile error. Figure 13.20 depicts the problem ambiguities of member functions, and Figure 13.21 shows the solution.

The Problem:

```
class base1
{
 public:
   void test() {cout << "Base1\n";}
};
```

```
class base2
{
 public:
   void test() {cout << "Base2\n";}
};
```

```
class derived :base1, base2
{
 public:
   void test1() {test();}
   void test2() {test();}
};
```

```
# include <stream.h>

main()
{
  derived sample;
  sample.test1();
  sample.test2();
}
```

Figure 13.20. *An Example of a Problem with Ambiguities in Member Functions in Multiple Inheritance*

The Fix:

```
class derived :base1, base2
{
 public:
   void test1() {base1::test();}
   void test2() {base2::test();}
};
```

Figure 13.21. *Solution to the Problem of Ambiguities in Member Functions in Multiple Inheritance*

Summary

The main items covered in this chapter on advanced C++ features were :

- Inline functions
- Overloading operators
- Pointers and objects
- Polymorphism
- Templates
- Multiple inheritance

Exercises

1) Modify the shapes example to support a new shape class.

2) Rewrite the shapes example without using (OOP). You will need to use switch statements and structures to implement the code. Compare the traditional version to the OOP version. How do they differ?

14 ToolBox, Memory and Symantec C++

Symantec C++ has an easy way to call the toolbox and also contains some nuances for handling memory.

Using the ToolBox from C++

THINK C++ provides data structures and glue code for accessing the Macintosh ToolBox, as shown in Figure 14.1.

```
PROCEDURE FrameRoundRect
   (r: Rect; ovalWidth, ovalHeight:INTEGER);
Pascal void FrameRoundRect(const CRect& r,
   short ovalWidth, short ovalHeight);

FUNCTION StringWidth(s: Str255):INTEGER;
pascal short StringWidth(const CStr255& s );
```

Figure 14.1. *Data Structures and Glue Code for Accessing the ToolBox*

In Figure 14.1, it appears that the procedure is calling the ToolBox directly. It declares a *Pascal* function, so it looks like a Pascal routine. This is an Apple extension. THINK C/C++ will not use the *Pascal* declaration.

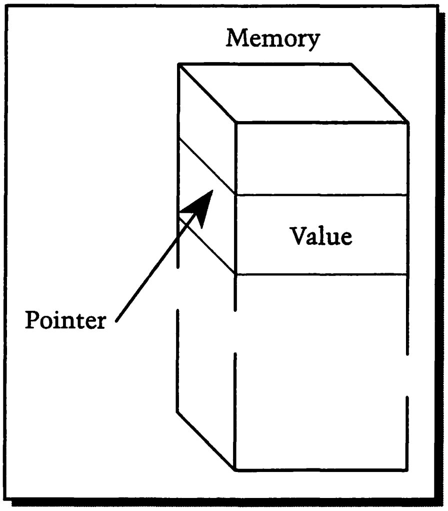
Pointers and Dynamic Memory

Figure 14.2 shows you how to request a chunk of memory. That memory is non-relocatable. The call to NewPtr is almost the same as a call to "new"; it sets memory on the heap but, it does not execute the constructor.

Master Pointers

Master pointers are pointers to dynamic memory that are created and maintained by the Macintosh ToolBox Memory Manager. These master pointers enable us to use relocatable memory via *handles*. Figure 14.3 shows how handles work.

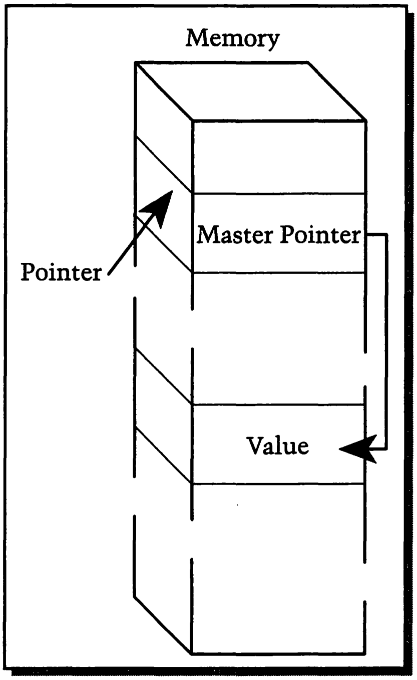
A handle is a pointer to a Mac master pointer. It is also known as *double indirection*. You use handles to point to *relocatable* areas in memory, as shown in Figure 14.4.



```
long *p;

p = NewPtr(400);

p[49] = 10;

DisposPtr(p);
```

The location pointed to by "p" had better not move!

Figure 14.2. *Pointers and Dynamic Memory*



```
long **h, *p;

h = NewHandle(400);
p = *h;

p[49] = 10;
(*h)[49] = 10; //Same

DisposHandle(p);
```

The data block pointed to by "h" might move!

Figure 14.3. *Example of a Handle in Memory*

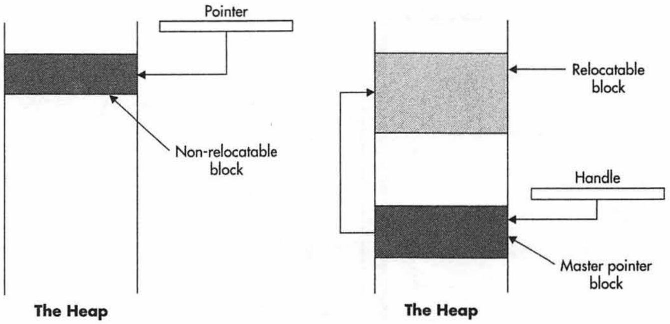
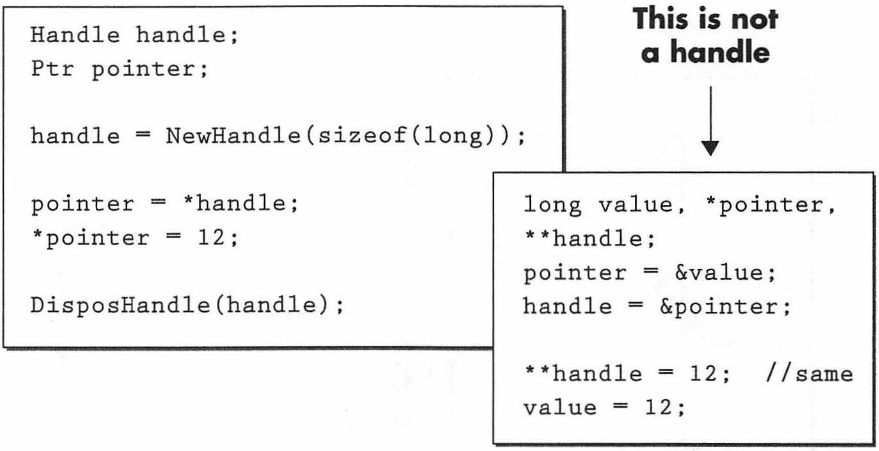


Figure 14.4. *Relocatable Memory*

Figure 14.5 points out what is a handle and, more important, what is *not* a handle.



```
Handle handle;
Ptr pointer;

handle = NewHandle(sizeof(long));

pointer = *handle;
*pointer = 12;

DisposHandle(handle);
```

This is not a handle

```
long value, *pointer,
**handle;
pointer = &value;
handle = &pointer;

**handle = 12;  //same
value = 12;
```

Figure 14.5. *Code Depicting a Handle*

A handle *must* point to a master pointer that has been created and is managed by the Mac ToolBox..

Dereferencing a Handle

Figure 14.6 depicts the dereferencing of a handle.

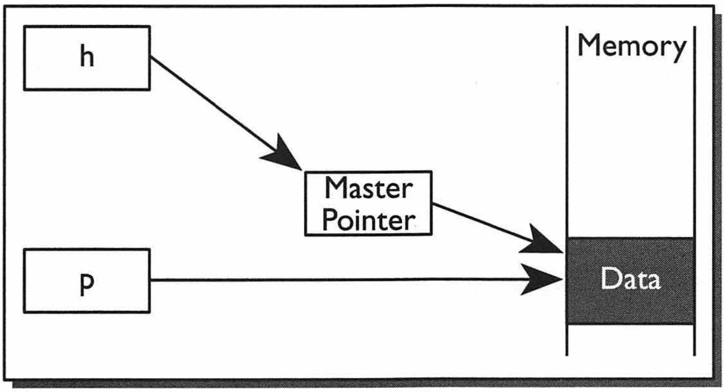


***Figure 14.6**. Dereferencing a Handle*

In Figure 14.6, *p* is a copy of the master pointer. If the data moves in memory, *p* points to garbage. When this happens, *p* is called a dangling pointer. You can avoid a dangling pointer by using *hLock* and *hUnlock*. Look at the following code:

```
Handle h;

ptr p;


hLock(h);

p = *h;
```

...use the dereferenced handle here...

```
hUnlock(h);
```

The Keyword "inherited"

The keyword *inherited* can be used unambiguously with descendants of *PascalObject* to refer to superclass methods. You might write the code this way:

```
class TMyClass : public PascalObject
public:
  virtual pascal void Free(…)    //OVERRIDE
  …
};

pascal TMyClass::Free(…)
{
  inherited::Free(…);
  //  … means the same as PascalObject::Free(…);

}
```

This is an Apple extension, but it will probably be in THINK C/C++ as well.

This example contains the class *TMyClass* (which is inherited from *PascalObject*). We have a *virtual* function, *Free*, which we have overridden. However, we want to call the original *Free*. Both functions have the same name, and we cannot call the original with *Free(...)*. That is the same as calling *this->Free*, which in this case would put us in an infinite loop because the function would continue to call itself; there is no way out of it.

By placing the keyword *inherited* in front of the two colons, the program will automatically call the original *Free*. This feature comes from Pascal, which is evident from the line *pascal TMyClass::Free()*.

Summary

In this chapter, we've covered THINK Extensions to C++:

- Calling the ToolBox from C++.
- Using handles and master pointers with C++.
- The perils of relocatable memory.
- The keyword *inherited.*

15 Using Symantec C++

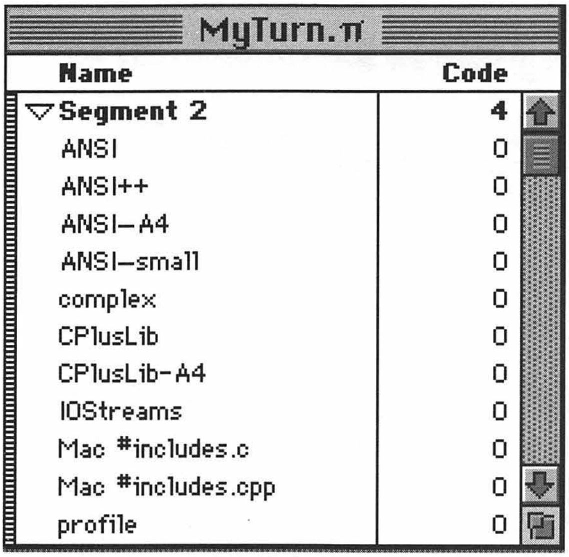
In Chapter 3, you created a simple project and built it into an application. The process probably went smoothly because the THINK Project Manager is easy to work with and understand. However, most projects that you develop will not be that simple and will require from you a much greater knowledge of the Symantec development environment than was necessary for the project in Chapter 3.

The Users Manual is well written and complete—it covers every menu, dialog, warning, and checkbox that you will see on your screen. Furthermore, the on-screen help and explanations give you most of the information you'll need as you use the varied and numerous features of the program. Our purpose in this chapter is to introduce you to those features, show you what the menus and dialogs look like, and add some information that is not available in the manual or on the screen. We leave the detailed descriptions up to Symantec C++ itself.

THINK Project Manager

The THINK Project Manager is the heart and soul of Symantec C++. It is the application that does everything, including accessing the debugger, libraries, SourceServer, and translators (compilers).

When you open the THINK Project Manager, the first thing it asks you to do is open a project, either new or existing. Figure 15.1 shows the Project Window, which is the mainstay of the THINK Project Manager.



MyTurn.π

Name	Code
▽ Segment 2	4
ANSI	0
ANSI++	0
ANSI–A4	0
ANSI–small	0
complex	0
CPlusLib	0
CPlusLib-A4	0
IOStreams	0
Mac #includes.c	0
Mac #includes.cpp	0
profile	0

Figure 15.1. *THINK Project Manager Project Window*

The Project window lists every file and library that you have included in your project, divides the project into 32-Kbyte segments (discussed in Chapter 3), and lists the number of bytes of complied code that each file contains. You can access any of the files listed in the Project Window by double-clicking on the file directly in the window.

Source Menu

The Source menu, which is divided into four main sections, allows you to do and view a number of things at various times during the creation of a project. The Source menu is closely related to the Project menu. Figure 15.2 displays the Source menu.

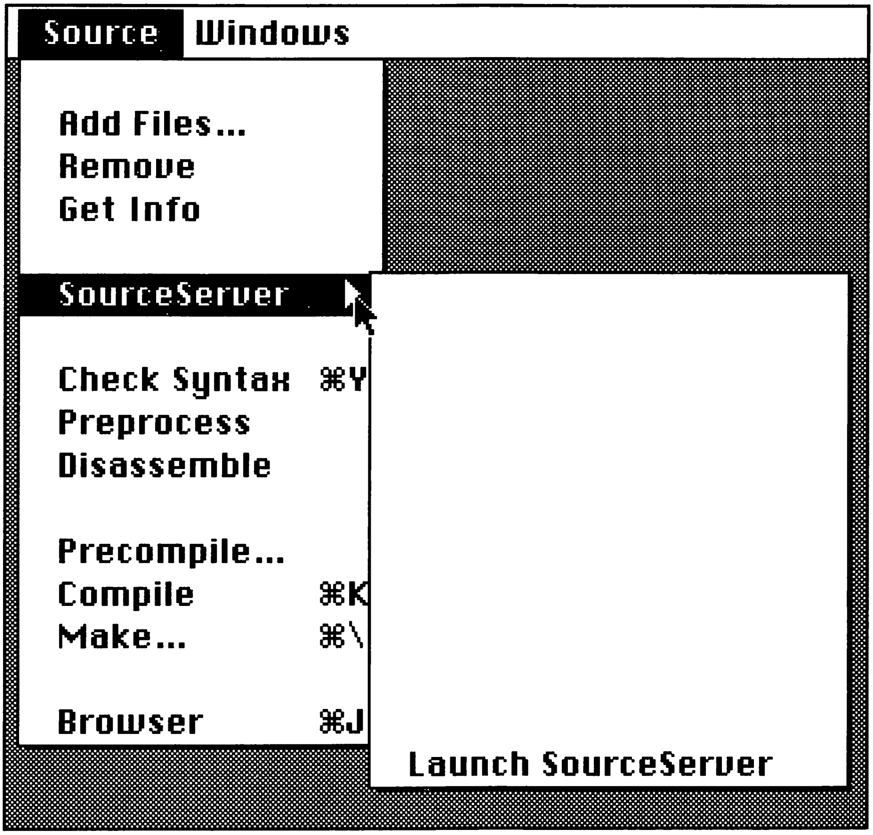


Figure 15.2. Source Menu

Adding, Removing, and Getting Information on Files

When you need to add files and libraries to your project, you do it through the Source menu. Selecting the **Add Files** option will bring up a dialog box (shown in Fig. 3.6 in Chapter 3) that asks you which files you wish to add. You can move around the various folders on your hard disk and add whatever you want; you can add selected files or all the files in a folder with the **Add All** button, or you can remove files with the **Remove** button.

Also included in the Source menu is the option Remove. This can be a dangerous option: if you happen to have a file selected in the Project Window and you accidentally click on **Remove**, THINK Project Manager will automatically remove the selected file. The problem is that you may not be aware of it because you may have another window open in front of the Project Window.

The **Get Info** option brings up a dialog box that gives you information on the components of your project; that is, it tells you the size of your files, segments, and entire project. The Users Manual gives a detailed description of what each of the elements in the dialog box means.

Debug

You'll find that the **Debug** option is grayed out unless you are running your program with the debugger on and select a file to edit through the debugger's Source menu. Once the window with the file to edit comes up, the THINK Project Manager menu bar reappears. At this point, you can choose **Debug** from the Source menu. This will take you back to the Source and Data windows of the debugger.

SourceServer

Source Server acts as a sort of librarian. If you have a large number of people working on the same project, you can fix the code so that people have to check routines in and out. The routines cannot be checked back in until they have been compiled and run with the rest of the program. Furthermore, two people cannot check out the same routine to work on at the same time.

Checking the Syntax, Preprocessing, and Disassembling

Check Syntax does only that; it doesn't compile. It does, however, tell you if your program will not compile by giving you error messages at the bottom of the screen.

Click on **Preprocess** if you think you might have any bugs in your macros, *#include* files, or *#ifdef* statements. **Preprocess** creates a new file that expands the macros, and so on, and allows you to see the contents of all these files.

Disassemble allows you to look at your code in assembly language, not a pretty sight. However, it can help you on occasion to debug your code and find out how efficient it is.

Precompiling, Compiling, and Making a Project

Precompile pertains only to header files, which are the *#include files*. Precompiling these files allows them to load faster during the compiling process.

When you select **Compile**, you compile or recompile only the file that you have open or selected. (When you choose **Run** from the Project menu and click the button to bring everything up to date, Symantec C++ recompiles any file that has been changed since it was last compiled before it runs it.)

Make is a facility that changes the out-of-date flag. You can force certain files that you want to recompile to be out of date by checking them in the dialog box that appears when you click on **Make**. The dialog box displays all of the files and libraries in your project. You can check on those items that you want to recompile or reload and THINK Project Manager will recompile them when you bring your project up to date, compile it, or run it.

Browser

Browser brings up your class diagram. If you have several classes—as in the Shape example in Chapter 13, for instance—you will see by the diagram that each of the subclasses was derived directly from *TList* or *TShape*. This browser can be extremely helpful, not only in keeping track of classes and subclasses, but in planning out your project.

Project Menu

The Project menu is the most used and probably most important of the menus in the THINK Project Manager. It is shown in Figure 15.3.

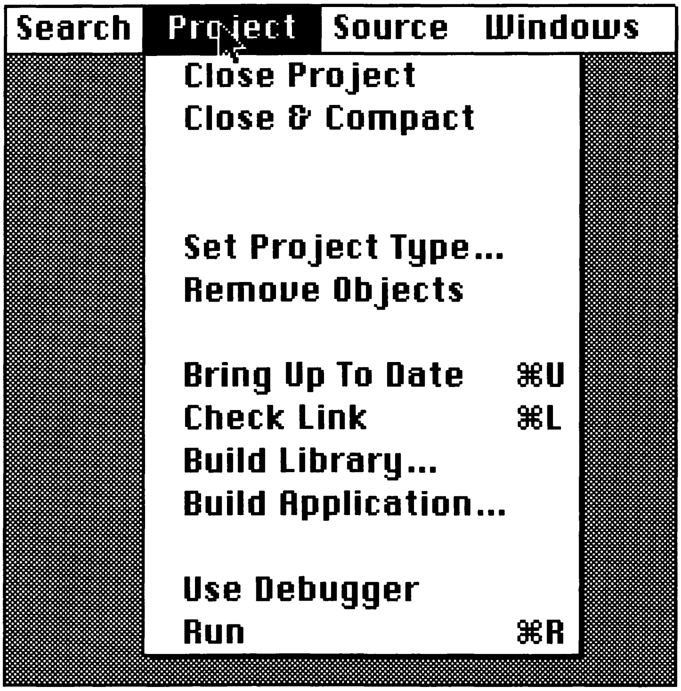


Figure 15.3. *Project Menu*

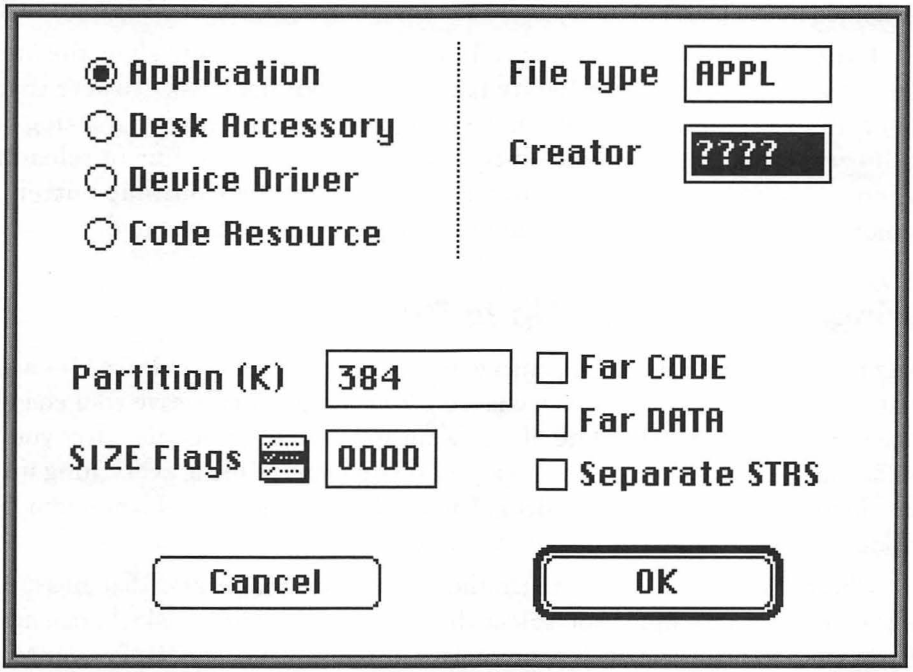
Closing the Project

When you click on **Close Project**, Symantec C++ will close the project that you currently have open and bring up a dialog box for you to open up another project. You can also choose **Close & Compact**, which stores the project in a compacted mode to save disk space. However, it takes more time to reopen that project from its compacted state, so if you are working on the project off and on, you will probably not want to compact it.

If you have other projects that you've opened up recently, the menu attached to **Switch To Project** will give you a list of those, and you can select one. This process of switching projects is very fast and saves you the time and trouble of going through the **Open** option under the File menu.

Setting the Project Type

Set Project Type allows you to choose among four project types listed in a dialog box, as shown in Figure 15.4.



● Application
○ Desk Accessory
○ Device Driver
○ Code Resource

File Type APPL

Creator ????

Partition (K) 384

SIZE Flags 0000

□ Far CODE
□ Far DATA
□ Separate STRS

Figure 15.4. *Dialog Box for Set Project Type*

As you can see in Figure 15.4, Symantec allows you to create applications, desk accessories, device drivers, and code resources. If you are creating an application, you will also create code resources. When you develop a desk accessory, you create a driver resource (DRVR) instead of a code resource (CODE). You would select **Device Driver** if you wanted to write a program for a laser printer, for example. Code resources are things like CDEVs, CDEFs, MDEFs, WDEFs, and so on. The default selection is **Application**, which brings up *APPL*.

Writing desk accessories (which are also drivers) and device drivers and building code resources is a complex process that is not well explained in the Users Manual. The Symantec C++ Users Manual gives you the volume and chapter references for each type.

If you choose **Application** (certainly the most common project type), the letters *APPL* will appear in the box next to **File Type**. You can set the **Creator** to what you want your application's signature to be. If you leave the box empty, your application icon will be the standard diamond shape with a hand over it.

The **Partition** size in Kbytes is 384 by default. The **SIZE Flags** box has to do with whether or not your application is 32-bit compatible, multifinder aware, and so forth. You can see the things available on the pop-up menu by clicking on the little box with the checkmarks in front of the **SIZE Flag** *box*. This is well explained in the Users Manual, as are **Far CODE**, **Far DATA**, and **Separate STRS**.

If the project that you have open has been compiled, it will show the number of bytes that each file or library takes up on your hard disk. To save space, you can select **Remove Objects** and Symantec C++ will bring up a dialog box telling you that removing the objects necessitates recompiling or reloading when you want to rerun the program. If you click the **Continue** button, it removes the objects in each file and turns the number of bytes to *0*.

Bringing the Project Up to Date

Any time you make an editing change to one of your files, Symantec C++ automatically marks it as having been changed. You can essentially save your changes by selecting **Bring Up To Date**. If you want to run the application after you've made changes, a dialog box will ask you if you want to bring everything up to date before it runs. The option **Bring Up To Date** does not run the program, but it does save and make permanent all of your changes.

Check Link simply goes through the program and makes sure that all aspects of the program link up. If you select this option, you will be asked once again if you wish to bring the project up to date. If you say **no**, THINK Project Manager will check the links in the old version of your program.

Build Library permits you to build your own library file and include it in your project. It is best to add the library to your project as a project file (that is, a *.π* file) rather than as a *.lib* file. If you add the library as a project file instead of building a library, Symantec C++ will include the code only for the items that you use in your program; it ignores the other files, libraries, and useless routines in the project file if your program does not need them. (This is another example of code optimization in Symantec C++.)

When you select **Build Application**, THINK Project Manager will bring the project up to date, link it, and also copy in the resource file if you have one. After you name the application and save it, it is then copied to your hard disk as a stand-alone application.

Using the Debugger and Running the Program

The debugger is a subordinate application in Symantec C++. You cannot launch it from the Finder; you have to have a project open to run it. If you check **Use Debugger** in the Project menu, the debugger will come up each time you run a program. (You can also launch the debugger by checking the box **Use Debugger** in the **Debugging** option inside the Edit menu of the THINK Project Manager.) See the more detailed discussion of the debugger in the section on the THINK Project Manager options menu later in this chapter.

Search Menu

While the Source and Project menus pertain to the THINK Project Manager, the Search and Edit menus deal strictly with the Editor. The Search menu acts much like the Find option in a good word-processing program, allowing you to

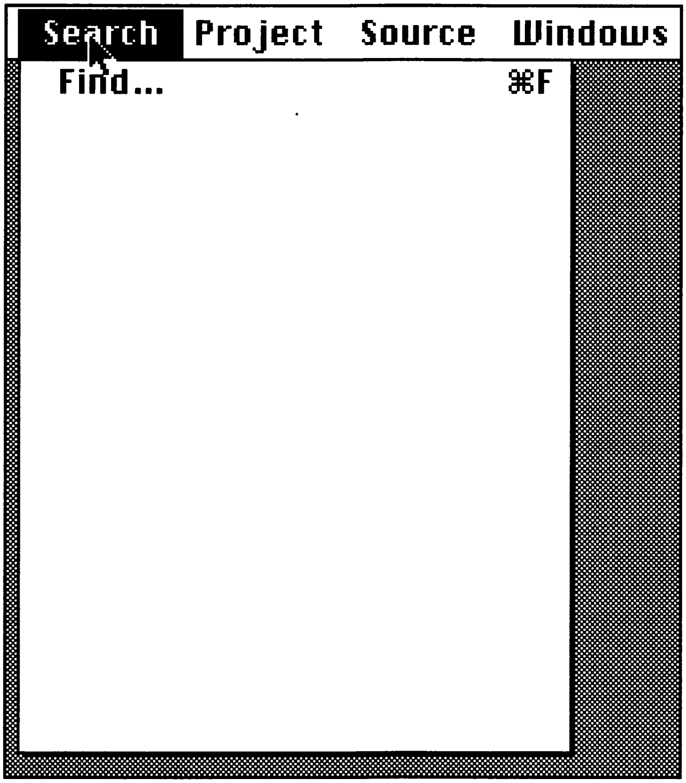


Figure 15.5. *Search Menu*

search through the files of your project for strings and symbols and replace items as well. The Grep option allows you to find strings that match a pattern. Figure 15.5 displays the Search menu.

The first section of the Search menu contains most of the items that you would normally find in the Find and Replace dialog box of a word processing program. Symantec C++, however, allows you to find and replace directly from the menu or through keystrokes without bringing up a dialog box.

Find in Next File will find whatever string you are looking for in any of the files associated with your project. To use the command for **Find** *in THINK* Reference, you must have installed THINK Reference in the same folder as Symantec C++.

Go To and Marking

The command **Go To** brings up a dialog box with a text edit box in which you enter the number of the line that you want to go to. The problem with this is that the lines are not automatically numbered and shown, so any number you enter in the box may be a stab in the dark.

You can set a mark in your code to help you get to a certain point in your program quickly. When you select **Mark** from the menu, a dialog box asks you to name the mark and press the **Okay** button. Later on, when you want to get to a marker (jump to it), you do so by holding down the **command** key and clicking on the menu bar. A menu pops up with a list of all the markers you've made in the program. Then you just click on the marker you want and it takes you to that part of the code.

When you click on the **Remove Marker** option in the menu, a dialog box containing a list of all your markers appears. As soon as you select one or more markers in the group, the **Remove** button becomes active and allows you to remove all of the selected markers.

Go To Next Error and **Go To Previous Error** are active only when you are attempting to compile or run your code and the Error Window comes up. Using those two options allows you to move around the errors in your code more quickly.

Edit Menu

The Edit menu contains the normal Macintosh editing tools: **Undo**, **Cut**, **Copy**, **Paste**, **Clear**, and ***Select All***, as seen in Figure 15.6.

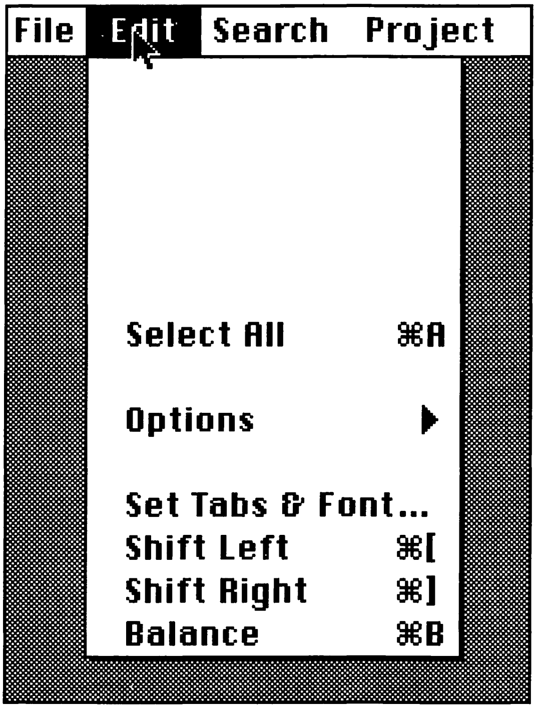


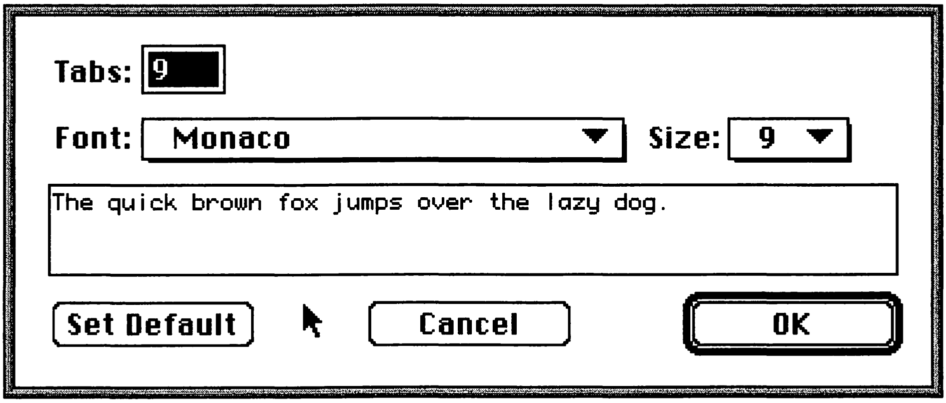
Figure 15.6 *Edit Menu*

Specific Edit Items

The Edit menu also contains some items not normally found in the edit menus of most applications for the Mac. These items, aside from the options, are discussed in the following paragraphs.

Tabs and Fonts

If you select **Set Tabs& Font** from the menu, the following dialog box (shown in Fig. 15.7) will appear.



Tabs: 9

Font: Monaco

Size: 9

The quick brown fox jumps over the lazy dog.

Figure 15.7. *Tabs and Fonts Dialog Box*

Since you can only have one font and size for the whole file, this dialog box allows you to choose the one you want. You set tab stops only if you want to use the **Shift Left** and **Shift Right** options. The default option in the dialog sets the tabs to 9, the font to **Monaco**, and the size to 9. The text box shows you what your font and size will look like on the screen.

Shift Left and Shift Right

If you select a block (or line) of code and either choose **Shift Left** or hold down the **Command** key and the open bracket, it will shift the block to the left in tabs of the number of spaces you have chosen. If you choose **Shift Right** or hold down the **Command** key and close bracket, it will shift the block to the right.

Balances

The **Balance** command helps you balance parentheses, brackets, and braces. If you set the cursor at the first open curly brace, for example, **Balance** will extend the selection to the corresponding closing curly brace. If you set the cursor to another open curly brace further along in the program, it will find that curly brace's corresponding closing brace. **Balance** works in both directions to enclose the smallest block of text enclosed in parentheses, brackets, or braces.

Options Menu

The **Options** dialog is also in the Edit menu. It contains choices for the THINK Project Manager; THINK C and Symantec C++, which are both translators; THINK Rez, which is the resource compiler; and .o Converter, which converts *.o* (Macintosh Programmers Workshop) files to Symantec C++.

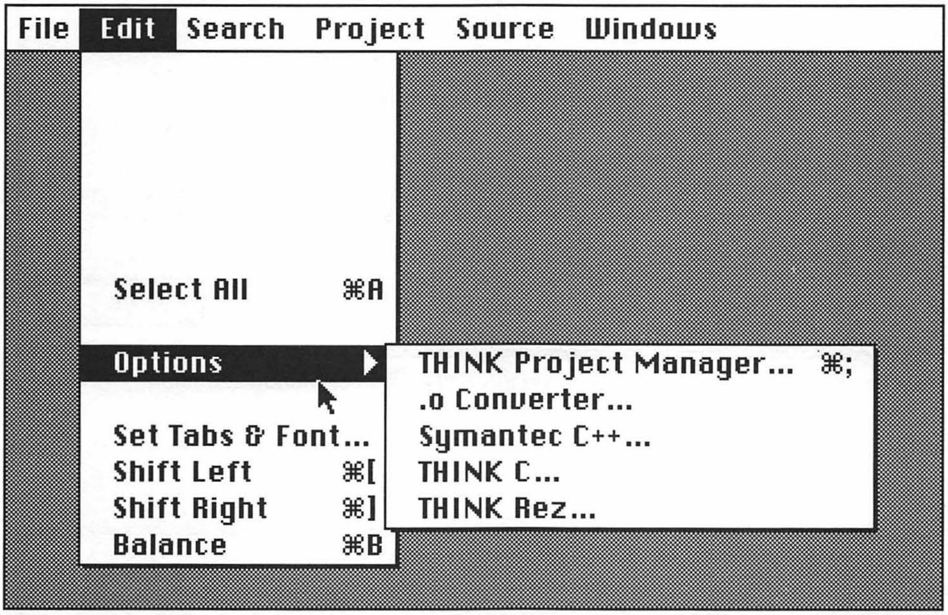


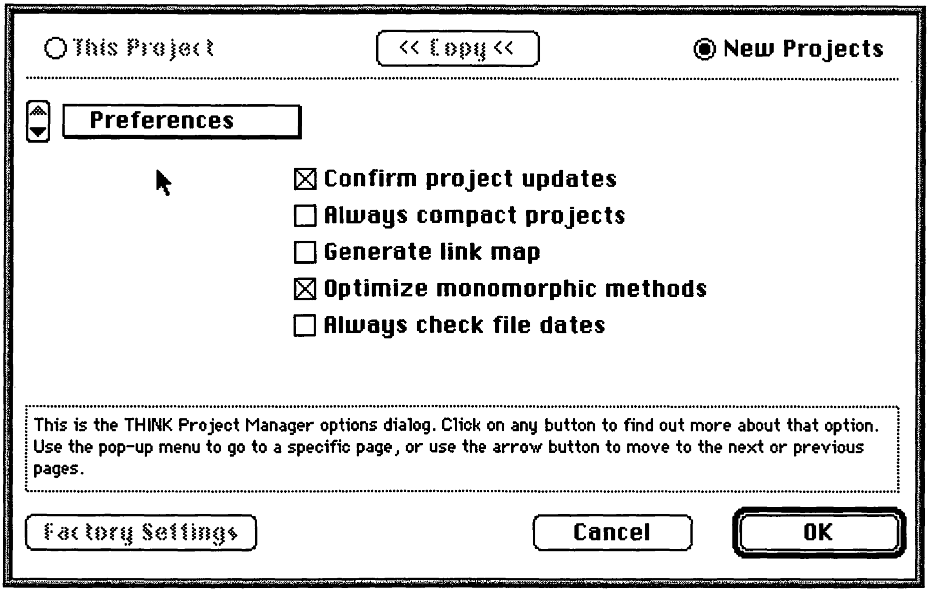
Figure 15.8. *Options Menu*

THINK Project Manager Options

When you choose the **THINK Project Manager** option, a dialog box corresponding to that in Figure 15.9 appears. The first dialog box shows items under the **Preferences** option. Other menus in the THINK Project Manager option are **Editor**, **Debugging**, **Extensions**, and **Project Window**.

Preferences

Most of the options in this menu are self-explanatory. However, if you need to get more information, click the mouse on any of the buttons or boxes; a complete explanation of the function will appear in the text box (above the **Factory Settings**, **Cancel**, and **OK** buttons).

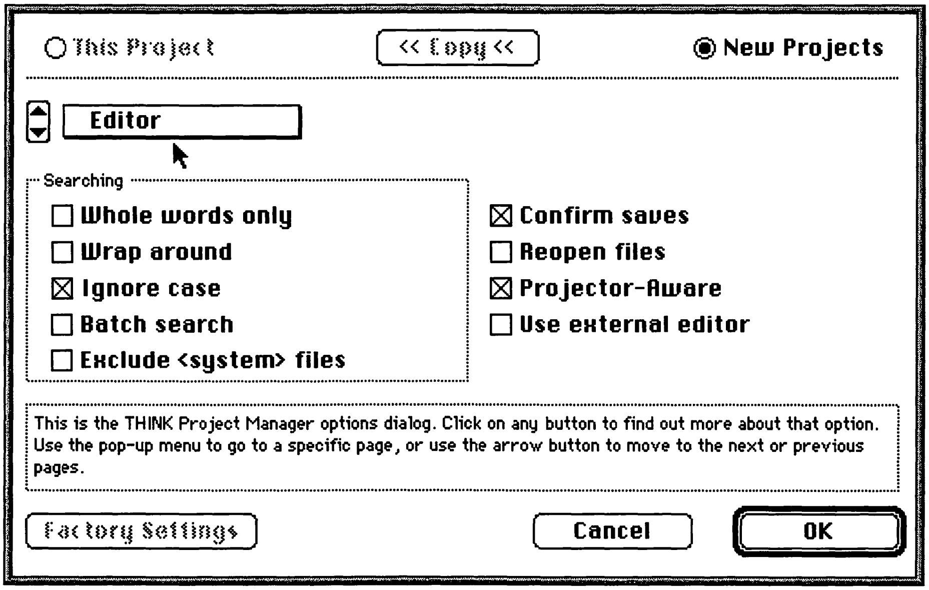


○ This Project ● New Projects

☒ Confirm project updates
□ Always compact projects
□ Generate link map
☒ Optimize monomorphic methods
□ Always check file dates

This is the THINK Project Manager options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.9. *Preferences Menu within the THINK Project Manager Option*



○ This Project ● New Projects

Searching

□ Whole words only
□ Wrap around
☒ Ignore case
□ Batch search
□ Exclude <system> files

☒ Confirm saves
□ Reopen files
☒ Projector-Aware
□ Use external editor

This is the THINK Project Manager options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.10. *Editor Menu within the THINK Project Manager Option*

Note that even if the **Confirm project updates** box is not checked, THINK C++ will update the projects anyway; it just will not require you to confirm the update.

Editor

Figure 15.10 shows the Editor menu within the *THINK Project Manager* option.

The *Editor* menu is divided into two sections: the items in the first section within the **Searching** box deal entirely with search and replace functions, and the items in the second section to the right of the box, are more general. You can get an explanation of each of the functions by clicking on any of the boxes within the lists. The only item that is not self-explanatory is the **Projector-Aware** option, which, when checked, forces THINK C++ to honor Macintosh Programmers Workshop Projector resources. This allows you to work on a large project with people who are using MPW Projector.

The **Use external editor** item is a new feature for Symantec C++. In the old versions of THINK C, if you wanted to use an editor other than the THINK C editor (e.g., QUED/M™ or Microsoft Word™), you had to get out of THINK C, work on the program in the outside editor, then go back into THINK C to compile the program. With this new optional feature in the Editor menu, you can now work on your program in an external editor without leaving the Symantec C++ environment. However, your external editor must be compatible with THINK. Check the User Manual carefully for instructions on using an outside editor.

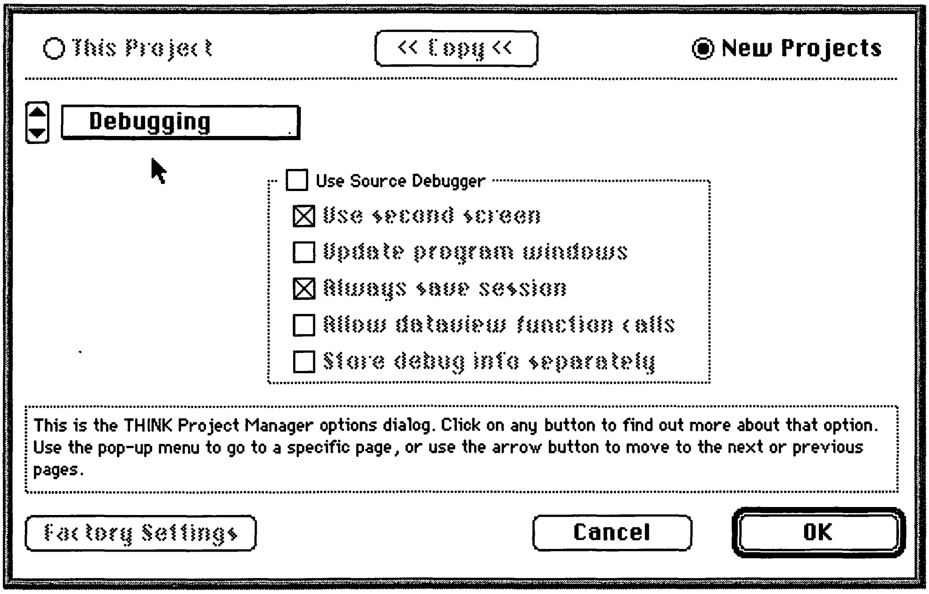
Debugging

Figure 15.11 shows the Debugging menu within the THINK Project Manager option.

To use any of the options in the Debugging menu, you must first check the box beside **Use Source Debugger**. Otherwise, all of the options within the box will be grayed out and unavailable. Checking the **Use Source Debugger** is the same as checking **Use Debugger** in the Project menu. Generally, you will have this turned on, which also means that you will be using the source debugger instead of an external debugger like Tmon or MacsBug.

A nice option in this menu is **Use Second Screen**. If you have more than one screen and you check this option, the debugger will automatically come up on the second screen.

If you do not have a second screen, you may find turning off the **Update program windows** useful. If you have the option turned on and the debugger menu is over the front of a window, for example, the source code window behind it will be constantly updated to the point where you may not be able to see what the actual problem in the code is. If you turn this option off, the window behind stays in a steady state.



□ Use Source Debugger

☒ Use second screen

□ Update program windows

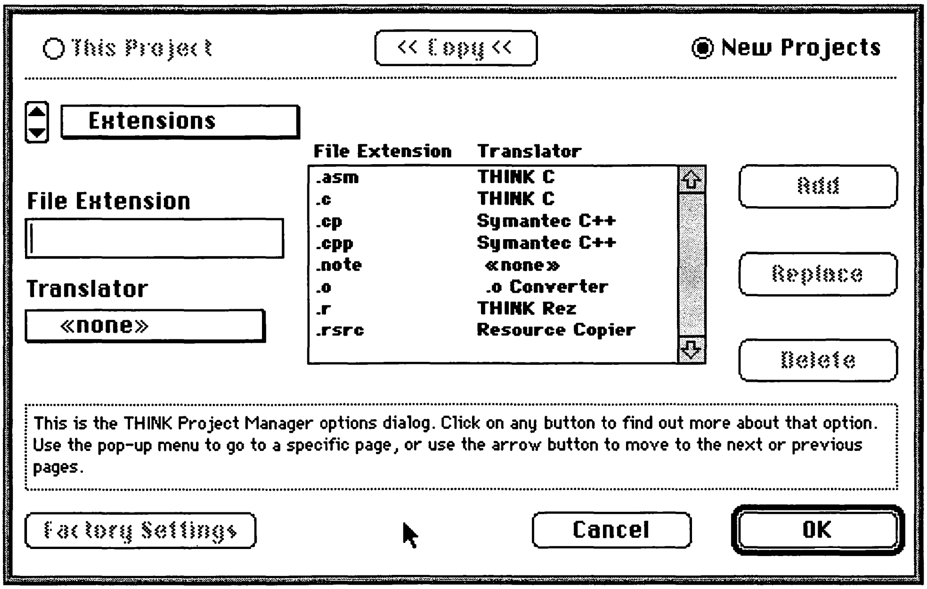
☒ Always save session

□ Allow dataview function calls

□ Store debug info separately

This is the THINK Project Manager options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.11. *Debugging Menu within the THINK Project Manager Option*



File Extension:

Translator: «none»

File Extension	Translator
.asm	THINK C
.c	THINK C
.cp	Symantec C++
.cpp	Symantec C++
.note	«none»
.o	.o Converter
.r	THINK Rez
.rsrc	Resource Copier

This is the THINK Project Manager options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.12. *Extensions Menu within the THINK Project Manager Option*

Always save session does just that, but it also marks the place where you left off in the debugger session. When you return, it takes you directly to that place.

Extensions

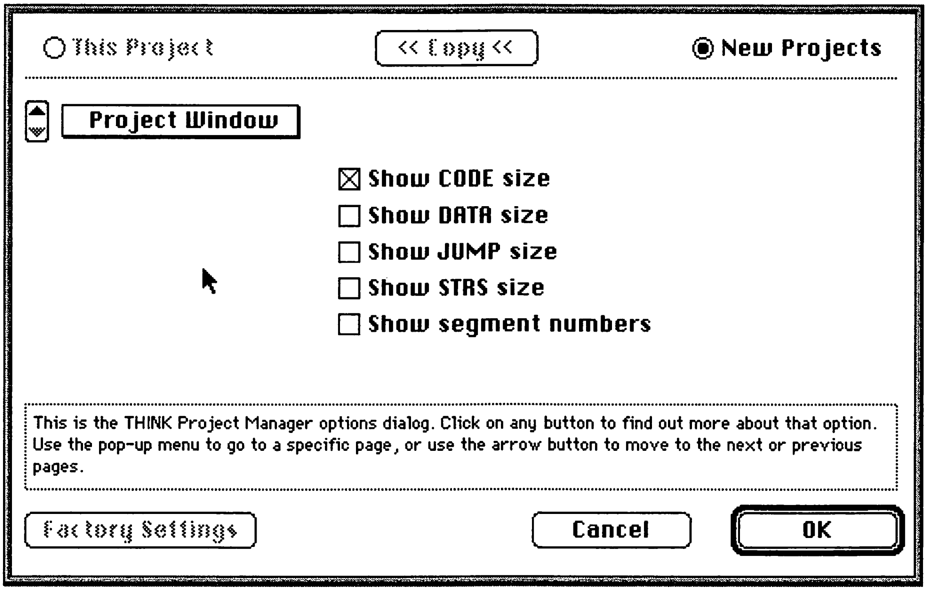
Figure 15.12 shows the Extensions menu within the **THINK Project Manager** option.

The Extensions menu gives you a way to set old files to have new extensions or translators and to add extensions. For example, if you created a file using a file extension and a compiler that are not on the installed extension and translator list in the Extensions menu, you can change the extension of your file, choose a translator, and click replace to turn the file into one that can be used in Symantec C++. Or, you can enter a new extension into the **File Extension** box, choose whichever translator you want, and click **Add** to add the extension to the list.

Project Window

Figure 15.13 shows the Project Window menu within the **THINK Project Manager** option.

This menu allows you to choose the number of the elements you want to see in the project window: **size of code**, **data**, **jump tables**, and **strings**, and the



☒ Show CODE size
□ Show DATA size
□ Show JUMP size
□ Show STRS size
□ Show segment numbers

This is the THINK Project Manager options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.13. *Project Window Menu within the THINK Project Manager Option*

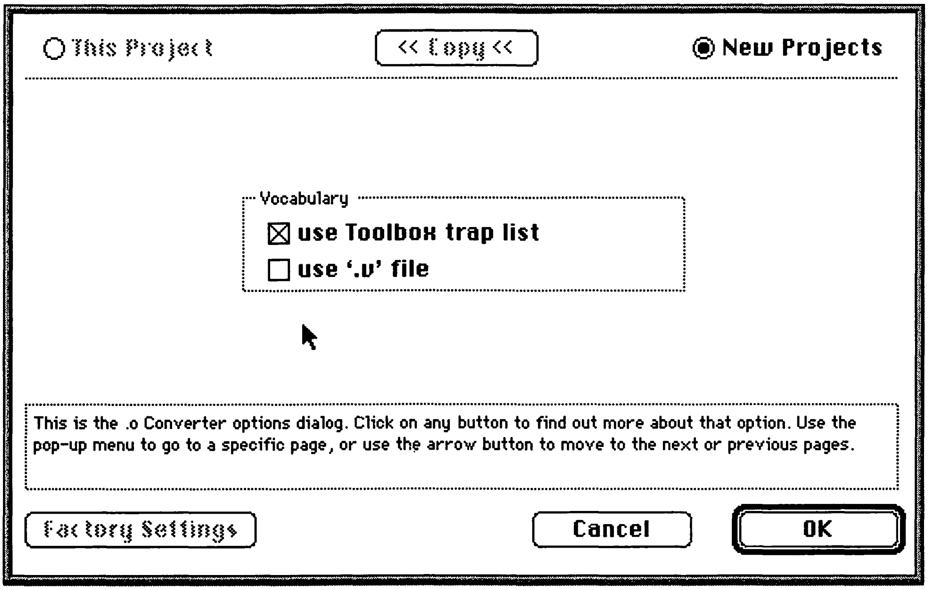
segment numbers. The default setting is to show code size only. However, the segment numbers appear even if you do not have that option selected. The project window expands in columns to the right by the number of options you have selected.

.o Converter

The ***.o*** **Converter** option allows you to use a Macintosh Programmers Workshop (MPW) .o file in a Symantec C++ project. This automatic conversion is new for Symantec C++. In previous THINK C versions, you were required to go through a two-pass, eight-step conversion process. Now, you simply include the .o files in your project, and Symantec C++ converts them. Figure 15.14 shows the dialog box for the **.o Converter**.

If the **use Toolbox trap list** box is checked, Symantec C++ converts the spelling of any all-uppercase MPW file name that has the same name as a Toolbox trap to the Inside Macintosh spelling of the trap.

If the *use* **.v file box** is checked, Symantec C++ converts any all-uppercase name that is the same as a name in the supplied vocabulary file to the mixed-case spelling in the vocabulary file. If the vocabulary file does not exist, Symantec C++ creates a new one.



Vocabulary

☒ use Toolbox trap list

□ use '.v' file

This is the .o Converter options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.14. *.o Converter Menu within the .o Converter Option*

Symantec C++

The Symantec C++ menu (under the Options menu) contains five options: **Language Settings**, **Compiler Settings**, **Code Optimization**, **Debugging**, and Prefix.

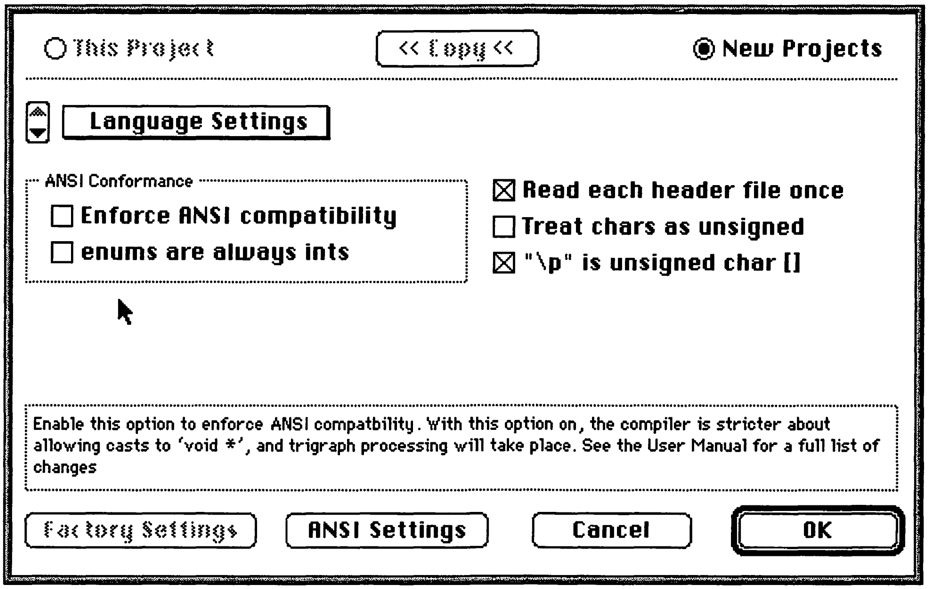
Language Settings

Figure 15.15 displays the language settings available in Symantec C++.

The first choices in the menu involve ANSI Conformance. You do not want to check **Enforce ANSI compatibility** if you are using Toolbox calls: they are not ANSI compatible. Also, Symantec C++ lets you set up enumeration constants to be something other than *ints*. However, if you check **enums are always ints**, that takes away your option to declare them something else.

Treat chars as unsigned means that anytime you declare something to be a *char* it will be read as unsigned no matter what you do.

Read each header file once is new to Symantec C++. If you have this box checked, you no longer have to use the *#ifndef*, *#define*, and *#endif* sequence in your header files to make sure the file is only read once. This is a time saver.



○ This Project   << Copy <<   ● New Projects

Language Settings

ANSI Conformance

□ Enforce ANSI compatibility

□ enums are always ints

☒ Read each header file once

□ Treat chars as unsigned

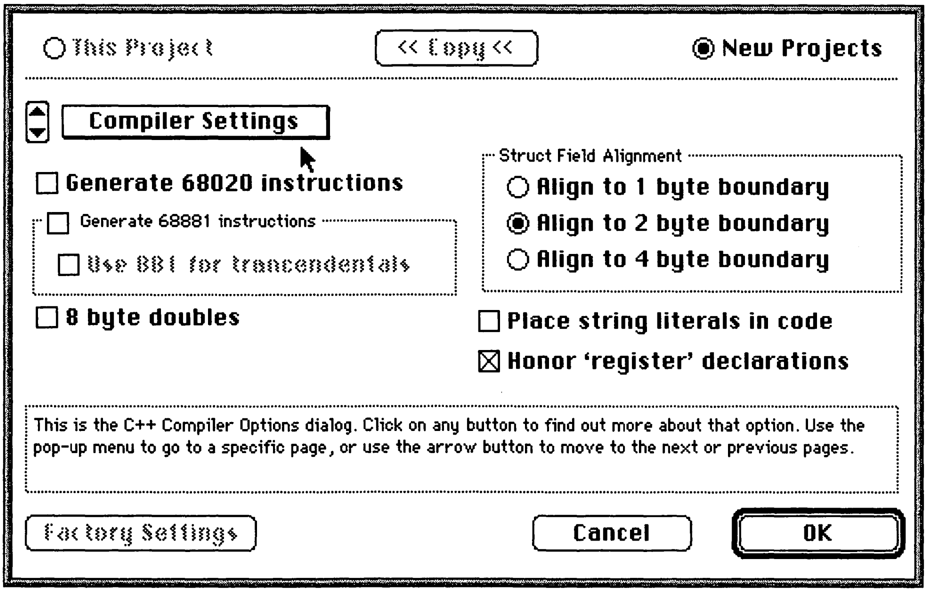
☒ "\p" is unsigned char []

Enable this option to enforce ANSI compatbility. With this option on, the compiler is stricter about allowing casts to 'void *', and trigraph processing will take place. See the User Manual for a full list of changes

Figure 15.15. *Language Settings Menu within the Symantec C++ Option*

Compiler Settings

Figure 15.16 shows the compiler settings available to Symantec C++.



○ This Project ◉ New Projects

□ Generate 68020 instructions

□ Generate 68881 instructions

□ Use 881 for trancendentals

□ 8 byte doubles

Struct Field Alignment

○ Align to 1 byte boundary

◉ Align to 2 byte boundary

○ Align to 4 byte boundary

□ Place string literals in code

☒ Honor 'register' declarations

This is the C++ Compiler Options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.16. *Compiler Settings Menu within the Symantec C++ Option*

The option **Generate 68020** instructions (as opposed to 68000) gives some additional instructions for the 68020 and 68030 that are enhanced over those for the 68000. Checking this option also allows you to check the **Generate 68881 instructions** for the math coprocessor.

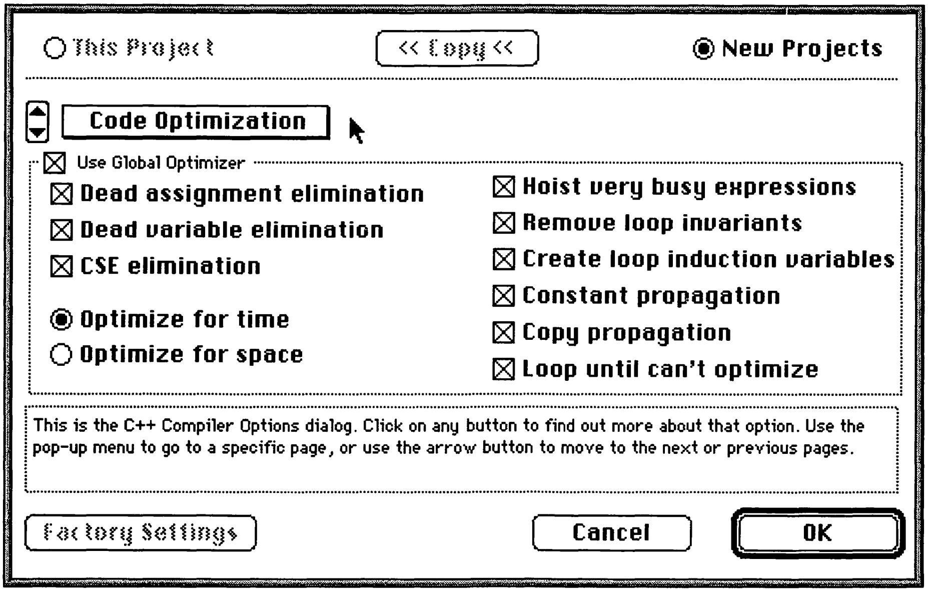
Struct Field Alignment allows you to change the byte boundaries to avoid the padding of bytes to even-byte boundaries (see Chapter 8, Data Structures). The problem with aligning to a 1-byte boundary is that you may still get an address error eventually.

Place string literals in code is an interesting and helpful option. These string literals and constants to into an area that Symantec C++ creates called a **data resource** instead of a **code resource**. The problem with the data area is that it is restricted to 32 Kbytes. If you have a lot of strings, you might not want to fill up all that data area with hundreds or thousands of strings, because you may overrun the data area. If you check this box, it will force the strings into the code area where you can segment them.

Honor 'register' declarations is most useful if it's turned off, since the Mac has a hard time honoring register declarations. If this box is not checked, then the compiler can ignore any of those declarations (especially in old code that might have been written for the VAX or Cray) and save some time.

Code Optimization

Figure 15.17 shows the Code Optimization menu for Symantec C++.



○ This Project ● New Projects

☒ Use Global Optimizer

☒ Dead assignment elimination

☒ Dead variable elimination

☒ CSE elimination

● Optimize for time

○ Optimize for space

☒ Hoist very busy expressions

☒ Remove loop invariants

☒ Create loop induction variables

☒ Constant propagation

☒ Copy propagation

☒ Loop until can't optimize

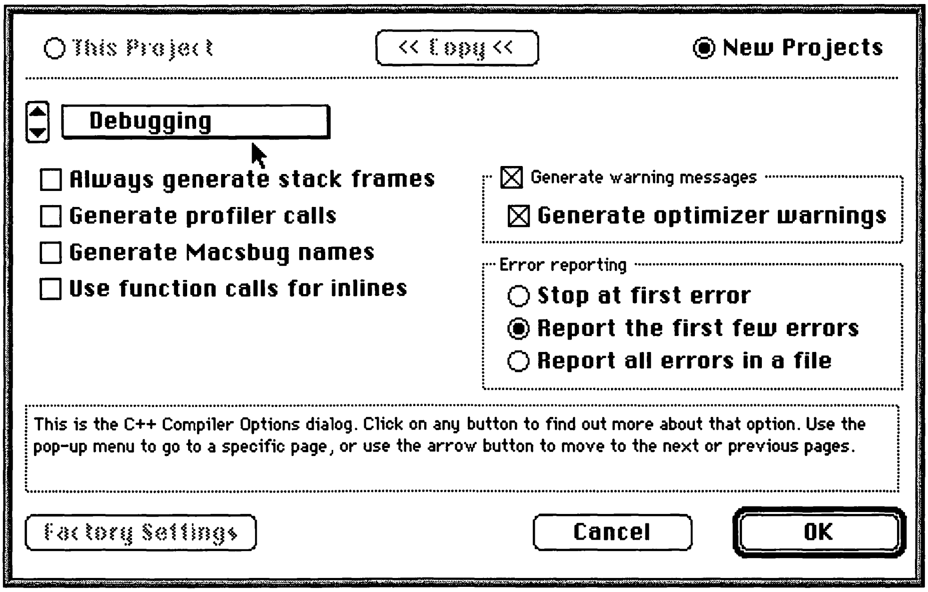
This is the C++ Compiler Options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.17. *Code Optimization Menu within the Symantec C++ Option*

One could write an entire book on these code optimizations. However, Symantec C++ does a good job of describing each of the options in the text box on screen as well as in the *Users Manual.* The only box that is not checked as a default (**Factory Settings**) is **Optimize for space**. (We discussed this "optimization" in Chapter 2, Object-Oriented Development.) Remember that you have to check the **Use Global Optimizer** box for any of the other options to be active.

Debugging

Figure 15.18 displays the Debugging menu for Symantec C++.



○ This Project

<< Copy <<

● New Projects

Debugging

□ Always generate stack frames

□ Generate profiler calls

□ Generate Macsbug names

□ Use function calls for inlines

☒ Generate warning messages

☒ Generate optimizer warnings

Error reporting

○ Stop at first error

● Report the first few errors

○ Report all errors in a file

This is the C++ Compiler Options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Factory Settings

Cancel

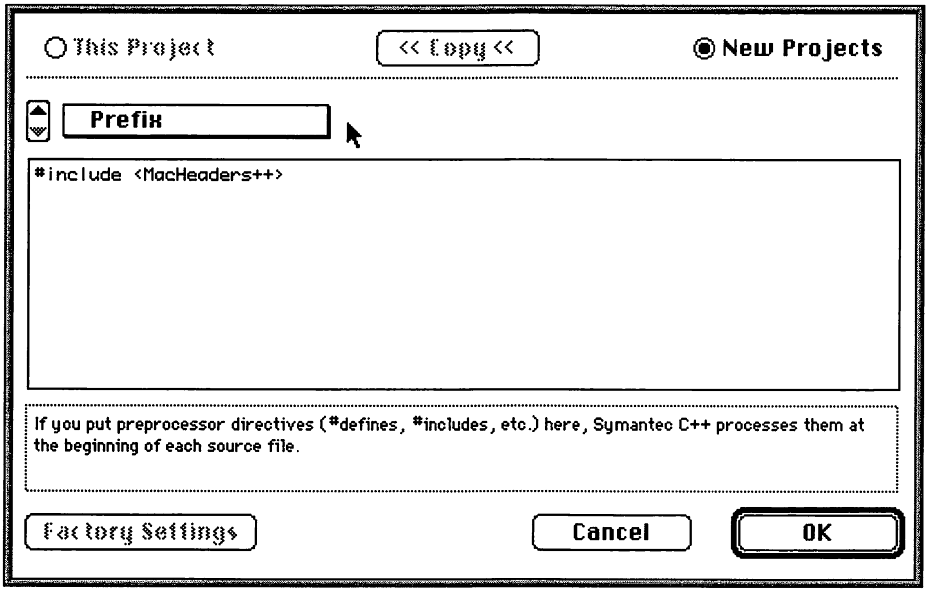
OK

Figure 15.18. *Debugging Menu within the Symantec C++ Option*

One interesting feature on this menu is **Generate Macsbug names**. When you compile code, the names of all your variables are lost (there are no names, only registers). The THINK Debugger can create a symbol table that gives the names that you were using and their memory addresses (or registers). It cross-relates them, so that if you request what a certain rectangle has in it, it tells you or it gives the memory location. If you check this box in the Debugging menu, it will generate the Macsbug™ symbol table with the names in it.

Prefix

Figure 15.19 shows the Prefix menu in Symantec C++.



```
#include <MacHeaders++>
```

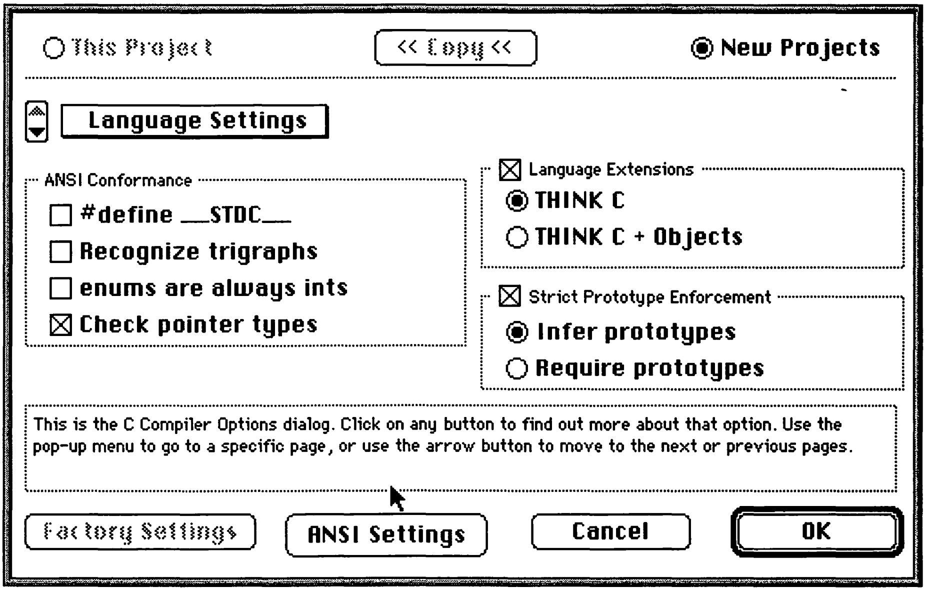
If you put preprocessor directives (#defines, #includes, etc.) here, Symantec C++ processes them at the beginning of each source file.

Figure 15.19. *Prefix Menu within the Symantec C++ Option*

Before your code is compiled, the items that you include in the Prefix edit text box will be processed at the beginning of each file. In Figure 15.19, the *#include* file is *MacHeaders++*. This is a list of precompiled commonly used headers for Macintosh Toolbox routines, such as *Quickdraw.h*, *Palettes.h*, *Icons.h*, and so forth. However, you may want to add a header file that has been commented out of the *MacHeaders++*. One way to do this is to go into *MacIncludes.cpp*, remove the comment slashes, recompile the list, and name it *MacHeaders++* to replace the old one. However, there is danger here because if someone else is compiling your code and does not have the same *MacHeaders++* file as you have, that person will get an error in the program. The best thing to do is simply to add a *#include* for the particular header file you want in your own code.

THINK C

Figure 15.20 shows the Language Settings menu for the **THINK C** options.



ANSI Conformance

□ #define __STDC__

□ Recognize trigraphs

□ enums are always ints

☒ Check pointer types

☒ Language Extensions

● THINK C

○ THINK C + Objects

☒ Strict Prototype Enforcement

● Infer prototypes

○ Require prototypes

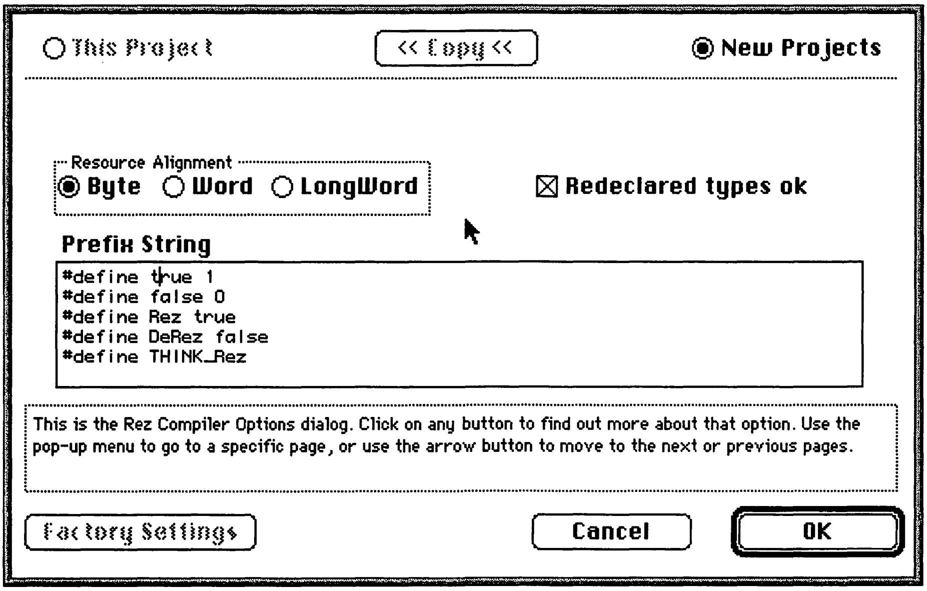
This is the C Compiler Options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.20. *The THINK C Compiler Options Dialog Box*

THINK C is included in the Symantec C++ environment so that earlier programs written in one of the THINK C versions can be included in your Symantec C++ programs. THINK C is not within the scope of this book.

THINK Rez

Symantec C++ has built-in Rez, which allows you to write textual resources and compile them, just as you compile code. This is a new and important feature. In prior versions of THINK C, you could create a Rez program (a text file) inside of THINK C, but you were then required to get out of THINK C and run Rez separately to compile it. You would then produce your .rsrc file either with **Rez** or **ResEdit**. Now, Symantec C++ has the ability to create a Rez file and, once you click **Build Application**, compile the resources automatically with Rez. Figure 15.21 shows the menu for the **THINK Rez** option.



```
#define true 1
#define false 0
#define Rez true
#define DeRez false
#define THINK_Rez
```

This is the Rez Compiler Options dialog. Click on any button to find out more about that option. Use the pop-up menu to go to a specific page, or use the arrow button to move to the next or previous pages.

Figure 15.21. *THINK Rez Compiler Options Dialog Box*

File Menu

The File menu of the THINK Project Manager contains the standard file items that most Mac applications contain. Figure 15.22 displays the File menu.

One puzzling thing about the File menu is that you can select **Save A Copy** As from the File menu, but it brings up exactly the same dialog box as the **Save As** option. If you want to save a copy of your source code and not replace the existing file, you must append the word copy after the name or change the name of the file.

The **Modify Read-Only** option pertains to projects using MPW Macintosh Programmers Workshop Projector. You cannot edit a file marked read-only when the **Projector-Aware** option is checked in the Editor menu (in the THINK **Project Manager** option under the Edit menu). You can select or copy it , but not its Projector resources. However, you can edit it by selecting the **Modify Read-Only** option. The icon changes to a pencil with a dotted cross-out for the editing process.

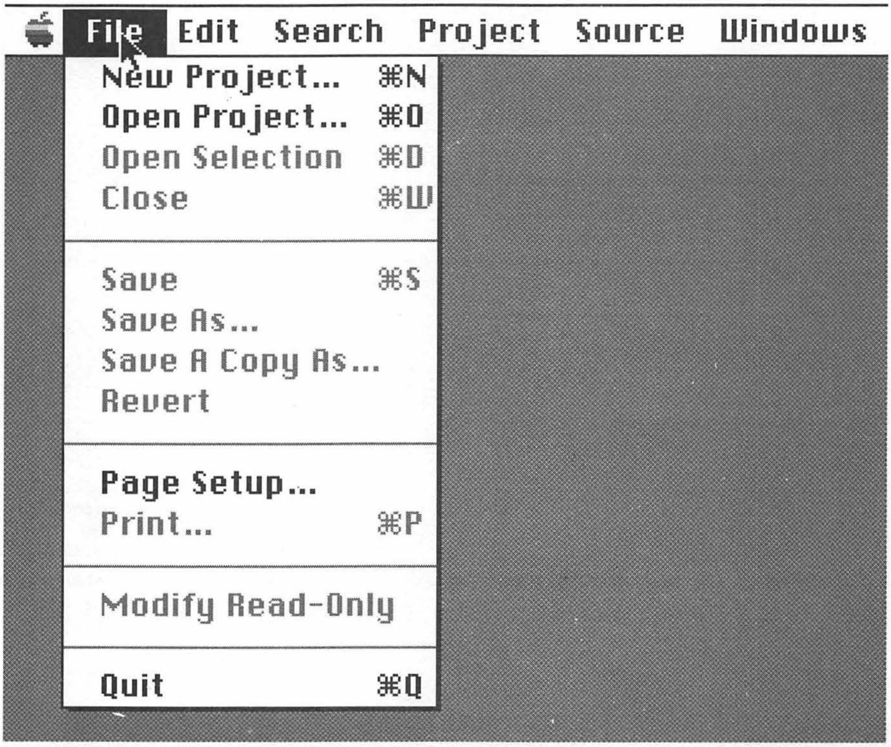


Figure 15.22. *File Menu*

Windows Menu

Figure 15.23 displays the Windows menu of the THINK Project Manager.

The Windows menu items are only active (with the exception of the **Full Titles** option, which is always active) when you have an editing window open. Most of the options here are self-evident. When you select **Full Titles**, the title in the open window will expand to show you to which tree in the **project tree** your open file is assigned.

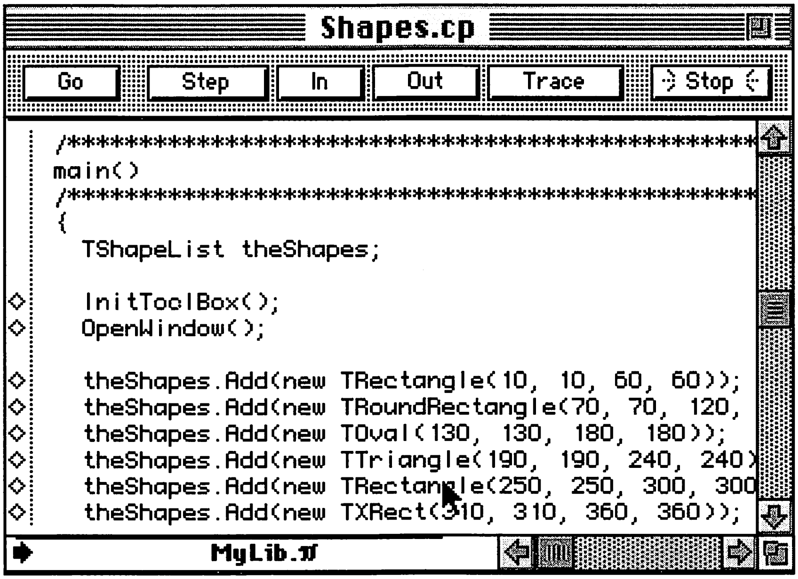
THINK Debugger

The THINK Debugger is a source-level debugger and is a subordinate program to the THINK Project Manager; it cannot run on its own. You launch the debugger by setting the option to use the debugger when you run a program from the Project menu. Once the debugger is launched, a Source window and a Data window appear on the screen and the menu bar changes from the Project Manager menu bar to the debugger menu bar. All of the debugger functions are explained fully in the Users Manual, but we will take the opportunity in this section on the debugger to point out things of interest and also exceptions.

Once you select **Use Debugger**, a little bug appears next to **Name** in the Project Window. Also, every file that you can debug will have a little diamond in front of it.

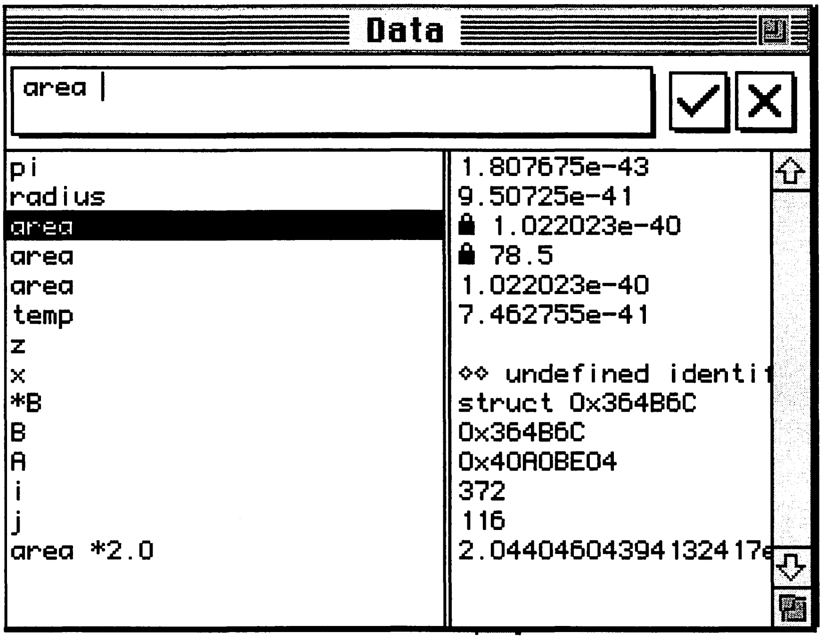
Source Window

The Source Window, which contains the source code of your program, is shown in Figure 15.24.



```
   /**************************************************
   main()
   /**************************************************
   {
     TShapeList theShapes;

◇    InitToolBox();
◇    OpenWindow();

◇    theShapes.Add(new TRectangle(10, 10, 60, 60));
◇    theShapes.Add(new TRoundRectangle(70, 70, 120,
◇    theShapes.Add(new TOval(130, 130, 180, 180));
◇    theShapes.Add(new TTriangle(190, 190, 240, 240)
◇    theShapes.Add(new TRectangle(250, 250, 300, 30
◇    theShapes.Add(new TXRect(310, 310, 360, 360));
```

Figure 15.24. *Source Window in THINK Debugger*



```
pi           1.807675e-43
radius       9.50725e-41
area         🔒 1.022023e-40
area         🔒 78.5
area         1.022023e-40
temp         7.462755e-41
z
x            ◇◇ undefined identi
*B           struct 0x364B6C
B            0x364B6C
A            0x40A0BE04
i            372
j            116
area *2.0    2.04404604394132417e
```

Figure 15.25. *Data Window in the THINK Debugger*

We've used *Shapes.cp*, which is part of the Shapes example from Chapter 13, as the example for the Source Window in Figure 15.24. The file that contains the main program is the file that comes up first in the debug window, but you can bring up any of the other files that can be debugged by selecting one and then selecting **Debug** from the Source menu.

Data Window

The Data Window allows you to examine and edit the values of your variables while you debug your programs. Figure 15.25 shows the Data Window.

The top box in the Data Window is the text entry box. You can copy a variable into the box by selecting it in the Source Window and then selecting Copy To Data from the debugger Edit menu. You can then enter the expression by clicking on the checkmark. It will appear in the lefthand column below the box (the expression column) and its value will appear in the righthand column (value column).

Inside the Data Window you can edit expressions, remove expressions, format values, display and change contexts, evaluate expressions, and set values.

File Menu

The File menu inside the debugger is very simple: you can save the file you are working on, and you can close. It does not allow you to open any other window or print. This menu is shown in Figure 15.26.

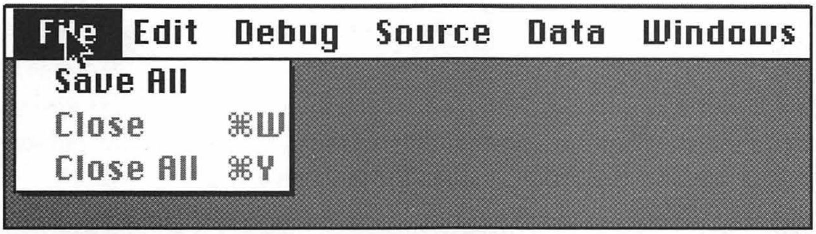


Figure 15.26. *Debugger File Menu*

Edit Menu

The Edit menu contains the standard options that most Edit menus contain plus the **Copy To Data** command mentioned in the discussion on the Data Window. This is an extremely useful feature when you want to work with variable expressions and values. The Edit menu is shown in Figure 15.27.

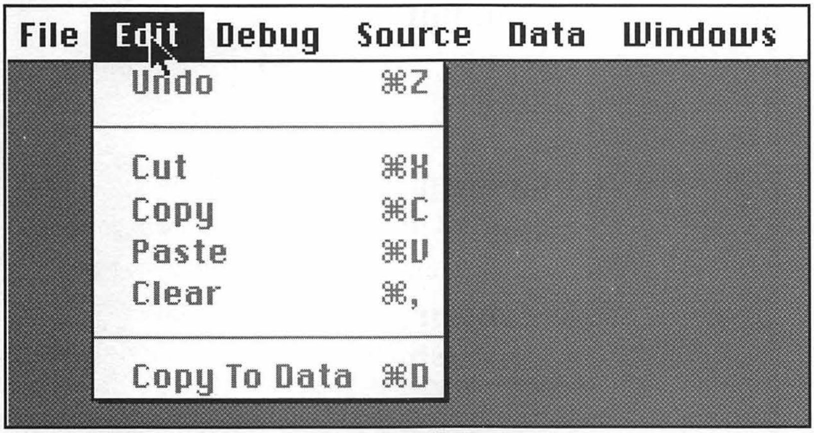


Figure 15.27. *Debugger Edit Menu*

Debug Menu

Much of what is in the Debug menu is redundant. The commands for **Go**, **Step**,

Step In, **Step Out**, **Trace**, and **Stop** appear in the Source Window of the debugger. However, if you do not want to use the mouse, you can use the equivalent keystrokes to do the same thing quickly. The Debug menu is shown in Figure 15.28.

The commands for **Go Until Here** and **Skip To Here** act as though you have set a temporary breakpoint in the Source Window. All you have to do is select a line and use the command **Go Until Here** and the program starts execution and stops at the selected line. The **Skip To Here** command skips to the selected line without executing any code in between.

The **Monitor** command works only if you have a low level debugger installed. When you use the **Monitor** command, it assures that all registers and low-memory globals contain the proper values for your program.

The only practical way to get out of the debugger is to use the **ExitToShell** command in the Debug menu. This exits you to the current file window where you can close the window, switch to another file or project, or quit the THINK Project Manager.

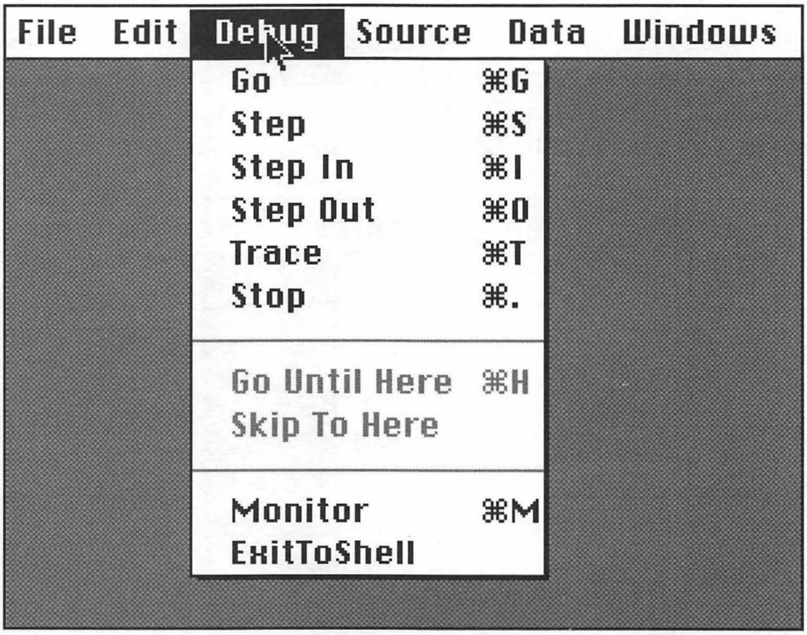


Figure 15.28. *Debugger Debug Menu*

Source Menu

The Debugger Source menu allows you to set and clear breakpoints, which we discussed above in the section on the Source Window. The **Attach Condition** command allows you to turn a simple breakpoint into a conditional breakpoint (something that has a condition attached to it). Since you may forget the conditions that you attached to certain breakpoints as you debug your program, the **Show Condition** command will show the associated condition in the Data Window. Finally, the **Edit <fileName>** command takes you out of the Source and Data windows and back to the current file window. The Source menu is shown in Figure 15.29.

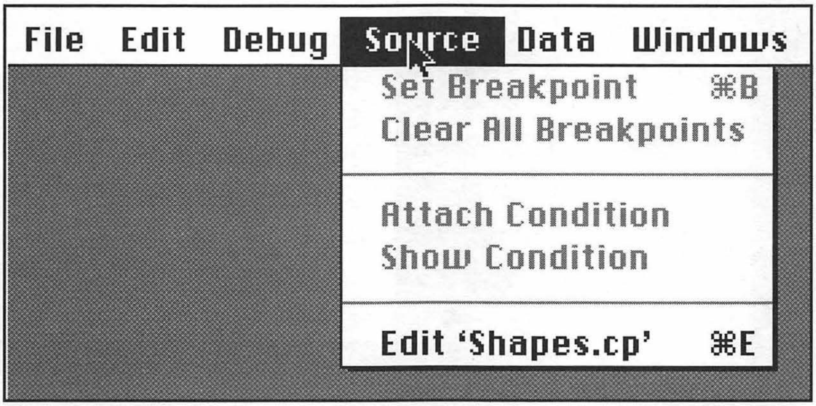


Figure 15.29. *Debugger Source Menu*

Data Menu

The Data menu works only with the Data Window. Some of the commands are complex but are well described in the Users Manual. Figure 15.30 shows the Data menu.

When you choose **Set Context** for a selected expression in the Data Window, the expression will copy to the next line and the cursor will blink in the data entry box. Enter what you want the context to be and press the **Return** key. You can also show the context of an expression by selecting **Show Context**. The context will appear in the values column.

The list of display formats from **Signed Decimal** to **Floating Point** allows you to change the format of an expression in the Data Window. Check the Users Manual for the types and formats available.

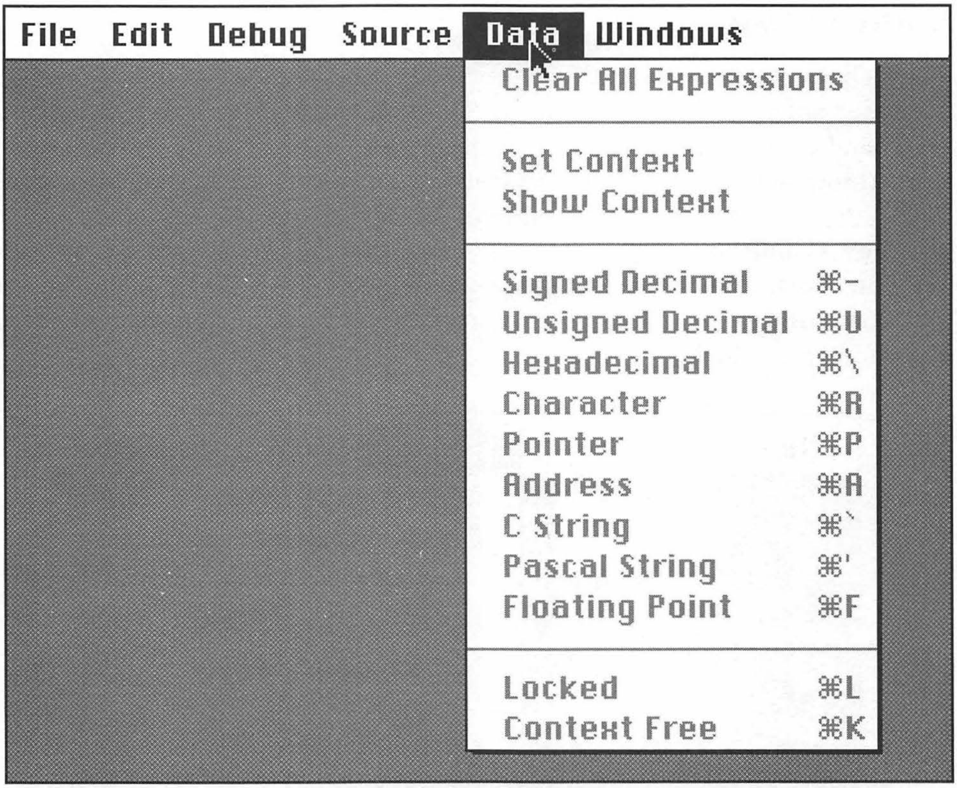


Figure 15.30. *Debugger Data Menu*

The **Locked** command is useful if you don't want an expression to be reevaluated, especially if you want to compare the values of the same expressions at different times. To lock the expression, select it in the Data Window and click on **Locked** or hold down the **command** key and press **L**. Similarly, if you want an expression to be context free (not have a value listed in the value column), you can select an expression in the Data window, click on **Context Free** or hold down the **command** key and press **K**.

Windows Menu

The Windows menu offers you an alternative way to switch around the different windows as you debug. The current source file is in the topmost position in the menu. After the segment line, the first window mentioned is the Source Window of the file being debugged and the second is the Data Window for that file. The Windows menu is shown in Figure 15.31.

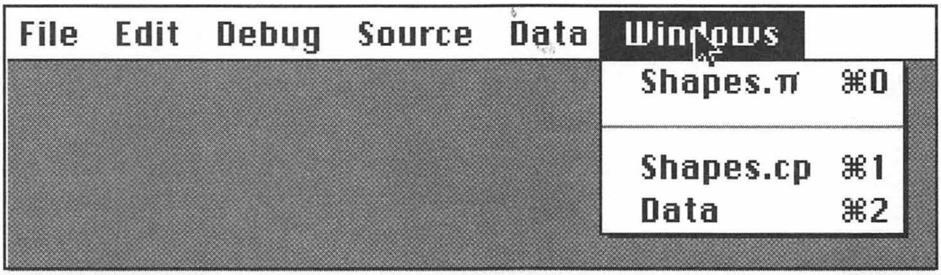


Figure 15.31. *Debugger Windows Menu*

Summary

This chapter has covered the main features of the Symantec C++ 6.0 development environment, with an emphasis on how to use those features. Although this is a reference chapter, it does not contain anywhere near the information you need to really understand the program. We strongly suggest that you carefully read your Users Manual for those items that are not clear to you or on which you want more information.

A Appendix A Glossary

abstract class An abstract class acts as a template for other classes. It is usually used as the root of a class hierarchy.

actor A model of concurrent computation in distributed systems. Computations are carried out in response to communications sent to the actor system.

argument A variable that is passed into a function.

brackets [ ] Used for subscripting an array.

C++ An object-oriented language based on C.

call Instruction that passes control to a different part of the program or function. A call executes other programs or parts of programs as though they were written in at the point where the call occurs.

class A data type from which objects can be created. It is used to specify the behavior and attributes common to all objects of the class.

compiler Utility that translates the source code from a high-level programming language (C++) into the object code used in running the machine.

curly braces { }
Also called braces, used in C++ to enclose executable statements.

data abstraction
Viewing data objects in terms of the operations with which they can be manipulated rather than as elements of a set. The representation of the data object is irrelevant.

declaration
Statement of values and data types having a global influence on a program.

delegation
Each object is considered an instance without a class, and new objects can be defined in terms of other objects. Attributes are delegated from base objects to the new objects.

encapsulation
The facility by which access to data is restricted to legal access. Illegal access is prohibited in an object by encapsulating the data and providing the member functions as the only means of obtaining access to the stored data.

function
Equivalent to a subroutine or function in FORTRAN or a procedure in Pascal, a function is a basic operational entity of any C program. A function encapsulates a series of computations in a black box, which you can then use without worrying about what is inside. With properly designed functions, you can ignore how a job is done and concentrate on what is done.

genericity
Technique for defining software components that have more than one interpretation depending on the parameters representing types.

handle
Pointer to a master pointer.

heap
Area of memory where space is allocated and released on demand via the Memory Manager.

hierarchy
The set of superclasses and subclasses derived from the superclasses can be arranged in a treelike structure, with the superclasses on top of classes derived from them. Such an arrangement is called a "hierarchy of classes."

inheritance The mechanism by which new classes are defined from existing classes. Subclasses of a class inherit operations of their parent class. Inheritance is the mechanism by which reusability is facilitated.

instance variables Variables representing the internal state of an object.

integer Whole number; that is, containing no fractions or decimal points.

linker Utility that links individual object-coded modules produced by a compiler into a complete machine language program ready for execution.

long Variable with a data length of 4 bytes.

master pointer Pointer to a pointer. Master pointers enable the Memory Manager to keep track of memory locations that it has relocated.

member functions Functions that are used to implement different operations on the object. They are part of the specification of a class.

message The process of invoking an operation on an object. In response to a message, the corresponding method is executed in the object.

methods Implementation of the operations relevant to a class of objects. Methods are invoked in response to messages.

multiple inheritance A subclass inherits from more than one superclass. Instances of classes with multiple inheritance have instance variables for each of the inherited superclasses.

object A combination of data and the collection of operations that are implemented on the data. Can also be described as a collection of operations that share a state.

object code Machine language produced by compilation of the source code.

parameter Sometimes used as a substitute word for argument.

parentheses ( ) In C++, a pair of parentheses encloses a variable in a statement. If no variable is to be passed in, nothing goes between the parentheses, but they are still required by C++ syntax.

persistence The phenomenon where data outlives the program execution time and exists between executions of a programs. All databases support persistence.

pointer Memory address containing a data item used in running a program.

polymorphism The same operation can be applied to different classes of objects. The operation on the object can be invoked without knowing its actual class.

prototype A prototype represents the default set of documents of a function.

Resource Data unit representing a dialog box, menu, alert, icon or other element of the Macintosh graphical interface.

returned value A value that is returned by a function.

reusability The ability to use well-designed software modules that have been tested, in several places, in different applications, so as to minimize development of new code. Object-oriented languages employ inheritance as a mechanism for reusability.

short Variable with the default length of 2 bytes in C++.

Smalltalk One of the first object-oriented languages. It provides an integrated software development environment, including the facility to display multiple windows and browse through classes.

source code Statements in a programming language.

structure Logical arrangement of a program.

structured programming Software development methodology that employs functional decomposition and a top-down design approach for developing modular software; traditional programming techniques of breaking a task into modular subtasks.

subclass A class that inherits behavior and attributes from another class. The subclass exploits reusability of design and reusability of code from its superclass.

superclass A class that serves as a base class to another class. A superclass provides behavior and attributes to classes derived from it by the inheritance mechanism.

this pointer A pointer to the current object in C++. Serves as a pointer to self.

variable Value that changes with program dynamics and is written to or read from memory as required.

void function Indicates to the Mac that there is no value to return for a particular function.

B Appendix B Bibliography

Anderson, Paul and Anderson, Gail. *Advanced C Tips and Techniques*, Indianapolis, IN: Hayden Books, 1988.

Andrews, Mark. *Programmer's Guide to MPW*, Volumes I and II, Reading, MA: Addison-Wesley, 1991.

Bar-David, Tsvi. *Object-Oriented Design for C++*, Englewood Cliffs, NJ: PTR Prentice-Hall, 1993.

Brown, T.D., Jr. *C for FORTRAN Programmers*, Summit, NJ: Silicon Press, 1990.

Cargill, Tom. *C++ Programming Style*, Reading, MA: Addison-Wesley, 1992.

Cargill, Tom. "Using Multiple Inheritance in C++," *Dr. Dobb's Macintosh Journal*, Volume 17, Issue 12, December 1992, pp. 48-51.

Coad, Peter and Jill Nicola. *Object-Oriented Programming*, Englewood Cliffs, NJ: Prentice Hall, 1993.

Computer Language, San Francisco, CA: Miller Freeman Publications, August 1991.

The C Users Journal ™, Volume 11, Number 5,. Lawrence, KS: R&D Publications, Inc., May 1993.

Davis, Stephen R. *C++ Programmer's Companion,*. Reading, MA: Addison-Wesley, 1993.

Develop, The Apple Technical Journal, Issue 8, Mt. Morris, IL: Apple Computer, Inc., Autumn 1991.

Dr. Dobb's Journal, San Mateo, CA: M&T Publishing, August 1991.

Ellis, Margaret A. and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Reading, MA: Addison-Wesley, 1990.

Hancock, Les and Krieger, Morris. *The C Primer,* New York, NY: McGraw-Hill, Inc. 1982.

Hansen, Tony L. *The C++ Answer Book,*. Reading, MA: Addison-Wesley, 1990.

Hughes, John M. *Programming in Zortech C++ with Version 2,*. Wilmslow, Cheshire, England: Sigma Press, 1991.

Journal of Object-Oriented Programming, New York, NY: SIGS Publications, Inc., February 1993.

Keffer, Thomas. "Why C++ Will Replace Fortran," *Dr. Dobb's Journal,* Volume 17, Issue 12, December 1992, pp. 39-46.

Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language,* Englewood Cliffs, NJ: Prentice-Hall, 1978.

Knuth, Donald E. *The Art of Computer Programming,* Volume 1, *Fundamental Algorithms,* Reading, MA: Addison-Wesley, 1973.

Koenig, Andrew. *C Traps and Pitfalls,* Reading, MA: Addison-Wesley, 1989.

Ladd, Scott Robert. *Applying C++,*. San Mateo, CA: M & T Books, 1992.

Lippman, Stanley B. *C++ Primer,*. Reading, MA: Addison-Wesley, 1991.

Maher, Tim. "A C++ Beautifier," *Dr. Dobb's Journal,* Volume 17, Issue 12, December 1992, pp. 23–26.

Mark, Dave. *Learn C on the Macintosh,* Reading, MA: Addison-Wesley, 1991.

Matthies, Kurt W.G. and Hogan, Thom. *Macintosh C Programming by Example,* Redmond, WA: Microsoft Press, 1991.

McDonald, Tom. "C for Numerical Computing, "*Journal of Supercomputing,* Volume 5, 1991, pp. 31–48.

Mead, Carver and L. Conway. *Introduction to VLSI Systems*, Reading, MA: Addison-Wesley, 1980.

Meyers, Scott. *Effective C++, 50 Specific Ways to Improve Your Programs and Designs*, Reading, MA: Addison-Wesley, 1992.

Moving from C to C++, Supplement to SIGS Publications, Cupertino, CA: Symantec Corporation, 1992.

Murray, Robert B. *C++ Strategies and Tactics*, Reading, MA: Addison-Wesley, 1993.

Object Magazine, New York, NY: SIGS Publications, Inc., Mar-Apr 1993.

Parker, Richard O. *Easy Object Programming for the Macintosh Using AppMaker™ and Think C™*, Englewood Cliffs, NJ: Prentice-Hall, 1993.

Pipelines, Volume 7, Number 3, Des Moines, IA: Microware, 1992.

Plum, Thomas. *Learning to Program in C*, Cardiff, NJ: Plum Hall, Inc., 1983.

Plum, Thomas and Dan Sacks. *C++ Programming Guidelines*, Plum Hall, 1991.

Pohl, Ira. *C++ for Programmers*, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1989.

"Programming in C++ on the Macintosh", A Programming Course Customized for HBO Time-Warner, Berkeley, CA: Bear River Institute, 1993.

Rao, Bindu R. *C++ and the OOP Paradigm*, New York, NY: McGraw-Hill, Inc., 1993.

Saks, Dan. "Standard C++: A Status Report," *Dr. Dobb's Journal*, Volume 17, Issue 12, December 1992, pp. 15–20.

Shapiro, Jonathan S. *A C++ Toolkit*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Shiffman, Harris. "Toward a Less Object-oriented View of C++," *Dr. Dobb's Journal*, Volume 17, Issue 12, December 1992, pp. 35–38.

Sprowls, R. Clay. *Computers: A Programming Problem Approach*, New York, NY: Harper & Row, Publishers, 1966.

Stevens, Al. "A Conversation with Bjarne Stroustrup," *Dr. Dobb's Journal*, Volume 17, Issue 12, December 1992, pp. 7–12.

Stevens, Roger T. *Fractal Programming and Ray Tracing with C++*, San Mateo, CA: M & T Books, 1990.

Straker, David. *C Style, Standards and Guidelines*. New York, NY: Prentice-Hall, 1992.

Swan, Tom. *C++ Primer*, Carmel, IN: SAMS Publishing, 1992.

Swan, Tom. *C++ Code Secrets*, Carmel, IN: SAMS Publishing, 1993.

THINK C, Object-Oriented Programming Manual, Cupertino, CA: Symantec Corporation, 1989.

THINK C, Users Manual, Cupertino, CA: Symantec Corporation, 1989.

Traister, Robert J. *Mastering C Pointers*, San Diego, CA: Academic Press, Inc., 1990.

Weston, Dan. *Elements of C++ Macintosh Programming*, Reading, MA: Addison-Wesley, 1990.

Wiener, Richard S. and Pinson, Lewis J. *An Introduction to Object-Oriented Programming and C++*, Reading, MA: Addison-Wesley, 1988.

Wilson, David A., Rosenstein, Larry S. and Shafer, Dan .*C++ Programming with MacApp*, Reading, MA: Addison-Wesley, 1990.

Index

A

abstract class, 39, 278, 331

abstract data type (ADT), 199, 225

accessing member data, 240

accessor, 36, 206-207, 208, 256

actor, 331

Add Files command
    (Source menu), 52, 53, 299

addresses of, 72, 74, 75, 79

ANSI (American National Standards
    Institute), 53-54, 59, 138
    conformance, 315

arguments, 28, 80, 136, 137, 158, 161,
    196, 283, 331
    default, 203-204
    functions, 35, 80, 138-146
    in derived classes, 232-233

arithmetic operators, 70, 85

arrays, 69, 72, 156, 173, 177-183, 203-
    204, 238, 270, 282-283, 331
    assignment, 179
    creating an array of structures, 190-191
    dynamic, 82
    indexing, 179
    initializing, 178
    multidimensional, 180
    subscript, 80, 81-82

ASCII characters, 12

assignment operators, 70, 73, 91-94

automatic variables, 150, 159

B

backslash (\), 162, 166

base classes, 31, 226-235, 263-270, 335

Bedrock framework, 33

binary numbering system, 8-12

binary operators, 70, 85-97, 125

bits, 13

bitwise AND operator, 73, 85, 87

bitwise exclusive OR operator, 73, 89

bitwise inclusive OR operator, 73, 88-89
brackets, 331
breakpoints, 327
break statements, 112-114, 118-119, 128-129
Bring Up To Date command (Project menu), 304
Browser, 301
Build Application command (Project menu), 55, 304, 320

C

.c files, 247
C++, 331
  comment syntax, 60
  fundamentals, 59-98
  improvements over C, 4, 31-32
  right-left rule, 156-157
  statements, 61-62
  style, 121-129
  variables, 62-64
  white space, 62
call, 331
C and Pascal for the Macintosh, 146
characters (chars), 63, 68
class diagrams, 35, 207
class libraries, 33
classes, 199-209, 331
  abstract, 39
  and structures, 201
  defining, 200
  member functions, 202
  objects, 33-35, 202
Code Optimization panel, 317
code resources, 51, 153, 303, 316
collaborators, 36, 208
comment syntax, 4-2
comments, 60-61
Compile command, 54-55, 301
compiler, 331
Compiler Settings panel, 316
  Generate 68020 instructions option, 316
  honor registers, 316
  Place String Literals, 316
  Struct Field Alignment, 316
comparison operators, 94-96
const, 154-155, 162
constants, 68
constructors, 203-204
  in derived classes, 231-233
curly braces, 332

D

data abstraction, 332
data hiding, 201, 209
data members, 31, 194, 200-208, 228-231, 235-236, 246, 285
data structures, 173
  see Chapter 8, Advanced Data Structures, 173
Debugger, 323-328
Debugging panel, 318
  Generate MacsBug names option, 318
decimal numbering system, 8
declarations, 66, 127, 136, 332
  and definitions, 66
  reference, 155
decrement operators, 72, 74, 77-78
definitions, 66, 129, 136, 176
delegation, 332
delete operators, 72, 72, 79, 83, 146, 181, 204
dereferencing pointers, 72, 74, 174, 176, 293
derived classes, 31, 225
  see Chapter 11, Subclassing and Inheritance, 225
destructors, 204-205
  in derived classes, 233
displaying hierarchies, 301

double, 64
dynamic binding, 41-42
dynamic memory, 181, 183, 290

E

Edit menu, 307-308
encapsulation, 35, 83, 194, 332
enumerations, 183-186, 315
expressions, 61, 98
Extensions menu, 247, 313
external variables, 151, 247

F

Far DATA option (Set Project Type menu), 303-304
fields, 36-39
File menu, 321
  Save A Copy As, 321
  Save As, 321
floats, 64
for loop, 116
free store operators, 79,146, 181
  destroying heap variables, 181
  testing for space, 181
friends, 208-209
  class, 209, 217, 233
  functions, 209
function calls, 35, 72, 80-81, 134-149, 159
  see Chapter 6, Functions and Variables, 131
functions, 131, 332
  default arguments, 141
  definitions, 35, 80, 134, 148
  explicit void, 142
  I functions in derived classes, 233-235
  inline, 159
  initialization, 205-206
  number of arguments in, 137
  overloading, 158
  passing, 138-140
    by pointer, 139
    by reference, 140
    by value, 139
  passing multiple values, 142
  prototypes, 136-137
  virtual, 227

G

genericity, 332
global variables, 151, 153, 159, 246

H

.h files, 247
handles, 290-293, 332
  dereferencing, 293
header file, 247
heap, the, 79, 146-147, 181-183, 203, 233, 332
hexadecimal numbering system, 9
hierarchy, 332

I

I/O operations, 165-171
  on the Mac, 171
  streams, 166
    advanced formatting, 169-170
    formatted output, 168-169
    generalized escape sequence, 167
    input stream example, 170-171
    I/O channels, 166-167
    predefined I/O stream manipulator, 167
#include, 160-162
inheritance, 38, 225-242, 284-287, 333
  and subclassing, 38
  multiple, 40, 284-287
  see Chapter 11, Subclassing and Inheri-

tance, 225
initializing member functions, 205
inline functions, 159, 162, 262
instances, 31, 35
    initializing, 238
    variables, 31, 34-35, 333
int data type, 64
integer, 333
iterator methods, 42-43, 270

K

keywords, 63
    reserved variables, 63

L

labels, 119-120
Language Settings panel, 315
left shift operator, 72, 85, 90
libraries
    adding to projects, 52, 53, 299
Linked List Example, 213-223
    adding and deleting nodes, 215-216
    building the code, 216
    creating a circular list, 215
    creating a friend class, 217-218
    destructor, 221-222
    display data function, 220-221
    extract function, 219-220
    insert function, 218-219
    input data function, 220
    main, 222-223
    statement of problem, 214
linker, 333
literal constants, 68, 70
local variables, 150, 159
logic, 17-18
logical AND operator, 73, 95, 96-97
logical operators, 85
logical OR operator, 73, 95, 96-97
logical values, 69
longs, 64, 333
loops, functions with, 116-118

M

MacApp framework, 33, 47, 205
MacHeaders++ (precompiled header), 319
Macintosh Programmers Workshop (MPW), 32-33, 65, 314
main() program, 134
master pointer, 333
member functions, 31, 194-197, 333
memory, 79, 146, 181, 265
    dynamic, 79, 146, 181
    disposing of old, 181
messages, 31, 333
    and methods, 35
method lookup, 41-42
methodology, 38
methods, 31, 35, 333
multiple inheritance, 40, 284-287, 333
    ambiguities of data members, 285
    ambiguities of functions, 286

N

naming conventions, 247
new operator, 72, 79
nibbles, 13-14
null pointers, 181
null statements, 104
null values, 69
numbers,
    signed and unsigned, 15-17
    systems, 7-12

O

object-oriented development, 27-44
    abstract class, 39

behavior of objects, 33-34
deriving classes, 34-35
encapsulation, 35
methodology, 38
object-oriented programming
benefits, 28
creating a program, 44
objects, 333
and classes, 33
behavior, 33-34
code, 333
creating, 237
dynamic, 237
octal numbering system, 11-12, 167
ones complement, 15-16
operators,
addition , 72, 86
update, 73, 93-94
address of, 72, 75
arithmetic and logical, 85
array subscripts, 72, 81-82
dynamic, new, 82
delete, 83
binary, 91
assignment, 91-92
comparison, 94
bitwise exclusive OR (XOR), 73, 89
bitwise AND, 73, 87
bitwise inclusive OR, 73, 88-89
cast, 72, 78
comma, 98
comparison,
decrement, 72, 77-78, 85
dereferencing, 72, 74
delete, 72, 79, 83
direct selection, 72, 83
division, 72, 87
equal, 73, 91
form ones complement, 75
function call, 72, 80-81
greater than, 73
greater than or equal to, 73
increment, 72, 76-77, 84
indirect selection, 72, 84
left shift, 72, 90
less than, 73
less than or equal to, 73
logical AND, 73, 96-97
logical not, 72, 75
logical OR, 73, 96-97
modulus, 72, 87
multiplication, 72, 86
negative, 72, 75
new, 72, 79, 82
not equal, 73
precedence, 71
right shift, 72, 90
scope resolution, 6-30
sizeof, 72, 78
subtraction, 72, 86
ternary, 97-98
unary, prefixed, 74
unary, postfixed, 80
optimizing, 118, 154, 227, 262
for time, 118
Optimization panel, 317
Options panels, 309-321
Code Optimization, 317
Compiler Settings panel, 316-317
Debugging panel, 318
Language Settings, 320
Prefix panel, 319
THINK Project Manager, 309
Preferences, 310-311
Editor, 310-311
Debugging, 311-312
Extensions, 312-313
Project Window, 313-314
.o Converter, 314
THINK Rez, 321

Out command (Debugger menu),
output streams, 165-171
    see Chapter 7, Input/Output Streams
overloaded functions, 158
overloaded operators, 193-194, 262-263
overriding, 40, 226, 249

P

parameters, 28, 80, 136, 137, 138, 148, 158, 161, 177, 196, 283, 333
    default, 203-204
    in derived classes, 232-233
    in functions, 35, 80, 138-146
parentheses, 334
partition sizes, 304
Pascal, 13-14, 32, 63, 69, 146, 290
    order of parameters pushed, 148
    string, 69
passing
    function arguments, 138
    by pointer, 139
    by reference, 140
    by value, 139
pause routine, 258-259
performance, improving with inline functions, 159
persistence, 334
PhoneBook Example, 244-259
    class diagrams, 247
    Class TCompany, 256
    Class TEntity, 254-256
    Class TPerson, 256
    naming conventions, 246
    Pause Routine, 258
    PhoneBook.cp, 257
    project, 247
    source file organization, 246
    UPhoneBook.cp, 257
    UPhoneBook.h, 249-250
    UString.cp, 249-250
    UString.h, 247
    UTelephone.cp, 251-254
    UTelephone.h, 251
pointers, 164-165, 334
    dereferencing, 74
    defining as class objects, 263-270
    initializing, 175
    null, 182
    void, 176
polymorphism, 269, 334
    and dynamic binding, 41-42
precedence, 71-73
precompiling, compiling, and making a project, 301
preference menu, 310
Prefix panel, 319
    MacHeaders++, 319
preprocessing
    and comments, 163
    and checking syntax and disassembling, 300
    C++ preprocessor, 160, 161-163
private, 34
    classes and structures, 201
    declaration of classes, 228-231
    defining derived classes, 226
    friends, 208-209
    in accessor methods, 206
    keyword, 63
procedural programming, 28
program flow, controlling, 103-120
Project Manager (THINK), 47-56, 298-329
Project menu, 302-305
    Bring Up To Date command, 304
    Build Application command, 304
    Remove Objects command, 304
    Set Project Type menu, 303-304
        Build options, 304
    Use Debugger command, 305
projects,

adding files, 52, 53, 299
closing, 302
protected keyword, 63
protected members, 201, 208-209, 228-231
protection keyword, 228
prototype, 136-137, 334
public keyword, 63
public members, 201, 208-209, 228-231

Q

QuickDraw.h file, 186, 319

R

reference variables, declaring, 155
register variables, 152-154
Remove Objects command (Project menu), 304
ResEdit, 320
reserved variables, 63
resources, 51, 146, 247, 303, 320, 334
returns, 80, 134-137, 142, 145-146, 153, 205, 334
keyword, 63
reusability, 334
right shift operator, 72, 85, 90
Run command (Project menu), 305

S

Save As command, 52, 321
scope, 70
scope resolution operator, 72, 159
Search menu, 305-306
Find Again command, 306
Find command, 306
Find in Next File command, 306
Go To Next Match command, 306
Go To Previous Match command, 306
Set Project Type (Project menu), 303-304
Shape Example, 270-280
short, 334
sizeof operator, 63, 72, 78
Smalltalk, 334
source code, 334
Source menu, 298-301
Add Files command, 299
Check Syntax command, 300
Compile command, 301
Debug option, 300
SourceServer, 300
stack, the, 142,-146, 148, 150, 151, 153, 181, 237
statements, 105-120
as lines of instruction, 61-62
blocks, 104
break, 112, 118-119
continue, 119
do-while loop, 116
else-if, 109-110
expression and null statements, 104
for loop, 116-118
goto, 5-20
if, 105-108, 114
if-else, 108-109
labels and goto, 119-120
switch, 110-112, 114
switch-break, 112-114
while versus for, 118
while, 115
static
data members, 235-236
keyword, 63,
member functions, 236
variables, 150-151
Step command (Debugger menu), 326
Stop command (Debugger menu), 326
stream library (I/O), 165
streams, 166-171

string constants (literals), 70, 316
strings, 69
structures, 173-197, 186, 334
  abstract data type, 199-225
  and bit fields, 191
  and member functions, 194-196
  anonymous, 188
  creating an array of, 190-191
  declaring with typedef, 185
  declaring, 187
  padding, 189
  referencing data elements of, 188
structured programming, 334
style, 121-129
  rules, 122-129
subclassing, 335
  and inheritance, 38, 225
superclass, 335
switch statements within functions, 43, 63, 110-114, 119, 128-129
Symantec C++, 47-56, 297-329
  adding libraries, 53-54
  Browser, 301
  building an application, 55
  compiling C++, 54
  creating a project, 48
  creating a source file, 52
  Edit menu, 307-308
  getting started, 48
  precompiled header files,
    conditional directives, 161
    MacHeaders++, 319
  preprocessor, 161, 163
  running a program, 55
  Search menu, 305-306
  Segments, 51
  setting project type, 303
  Source menu, 298-305
  SourceServer, 300
  THINK Project Manager, 298

T

templates, 281-283
  for a class, 282
  for a routine, 282-283
temporary breakpoints, 326
THINK C, 320
THINK Debugger, 323-328
  Data menu, 327
  Data window, 324
  Debug menu, 325
  Edit menu, 325
  File menu, 325
  Source menu, 327
  Source window, 323
  Window menu, 328
THINK Project Manager,
  Options, 309-320
    .o Converter, 314
    Debugging, 311
    Editor, 311
    Extensions, 313
    Preferences, 309
    Project window, 313
  Symantec C++ compiler, 315-319
  THINK C, 320
  THINK Rez, 320
this keyword, 63, 205
this pointer, 241-242, 335
Toolbox, Memory and Symantec C++, 289-294
    dereferencing a handle, 293
    keyword inherited, 294
    master pointers, 290
    pointers and dynamic memory, 290
    using from Symantec C++, 290
translators, 48, 298, 309, 313
twos complement, 16-17
type conversion, 63, 149
typedef, 185

U

unary operators, 74-85
unions, 192
Use Debugger command (Project menu), 305

V

v-tables (virtual tables), 42
variables, 335
    const, 154
    declaring, 66
    defining, 66
    external, 151
    global, 151, 153, 159, 246
    initialization of, 66
    passing by pointer, 139
    passing by reference, 140
    passing by value, 139
    predefined, 63
    register, 152
    reserved, 63
    signed and unsigned, 67
    static, 150
    storage, 149-154
virtual functions, 40, 227-228
virtual, 42, 233, 262, 269, 278
    keyword, 63
void, 137, 145, 146, 147, 205
    functions, 335
    keyword, 63
    pointers, 176

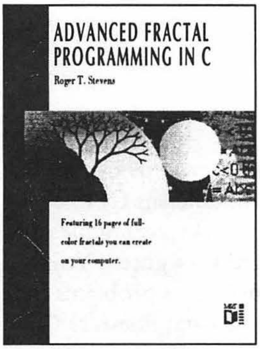
W

while loop, 115, 116, 118
windows, 322
    Full Titles option, 322

Z

Zortech C++, 32

A Library of Technical References from M&T Books

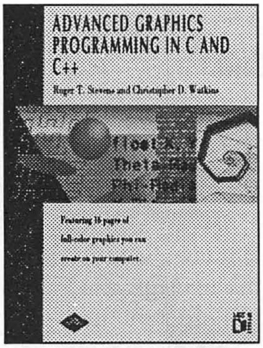

ADVANCED FRACTAL PROGRAMMING IN C
Roger T. Stevens
Featuring 16 pages of full-color fractals you can create on your computer.

Advanced Fractal Programming in C by Roger T. Stevens

Programmers who enjoyed our best-selling Fractal Programming in C can move on to the next level of fractal programming with this book. Included are how-to instructions for creating many different types of fractal curves, including source code. Contains 16 pages of full-color fractals. All the source code to generate the fractals is available on an optional disk in MS/PC-DOS format. 305 pp.

Book/Disk	$39.95	#0974

Level: Intermediate


ADVANCED GRAPHICS PROGRAMMING IN C AND C++
Roger T. Stevens and Christopher D. Watkins
Featuring 16 pages of full-color graphics you can create on your computer.

Advanced Graphics Programming in C and C++ by Roger T. Stevens and Christopher D. Watkins

This book is for all C and C++ programmers who want to create impressive graphic designs on IBM PCs or compatibles. Through in-depth discussions and numerous sample programs, you will learn how to create advanced 3-D shapes, wire-frame graphics, solid images, and more. All source code is available on disk in MS/PC-DOS format. Contains 16 pages of full-color graphics. 560 pp.

Book/Disk	$39.95	#1733

Level: Intermediate

1-800-488-5233

A Library of Technical References from M&T Books

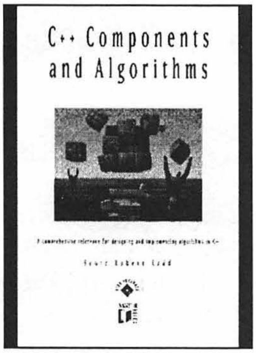
Applying C++
• Your expert guide to writing powerful C++ programs.
• Loaded with tips and code examples.
• Disk includes sample source code.
Scott Robert Ladd

Applying C++ by Scott Robert Ladd

Intermediate level programmers . . . this is your next book! Learn how to design and maintain clean, efficient C++ applications and do it by using the very tricks, techniques and strategies of the industry's acknowledged C++ gurus. You want to get around language limitations? Solve problems in the real world? Find out what works and what doesn't? Get inside this volume and discover the keys to object-oriented programming design, C++ tricks and traps, interfacing with MS-DOS, planning and implementing C++ libraries and applications, building a spreadsheet and much more. Includes complete source code disk. 544 pp.

Book/Disk	$34.95	#2624

Level: Beginning-Intermediate



C++ Components and Algorithms by Scott Robert Ladd

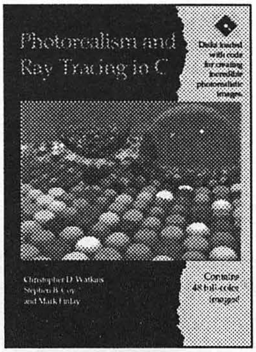
It's true: experienced C programmers always need the kind of comprehensive tools that can help them develop and maintain powerful C++ applications. This excellent volume is where you can find them—all of them! Memory management, indexed files using B-Trees, mathematical programming, adaptive algorithms and more. The in-depth discussions and numerous source code examples are geared toward an understanding of C++'s inner workings. The programs and classes presented are compatible with various C++ compilers, making them valuable to a wide audience of C programmers. All source code included on disk in MC/PC-DOS format. Now you can C more than ever! 512 pp.

Book/Disk	$39.95	#2276

Level: Advanced

1-800-488-5233

A Library of Technical References from M&T Books

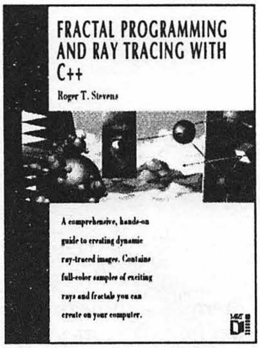

Photorealism and Ray Tracing in C
Disks loaded with code for creating incredible photorealistic images.
Christopher D. Watkins
Stephen B. Coy
and Mark Finlay
Contains 48 full-color images

Photorealism and Ray Tracing in C by Christopher Watkins and Stephen Coy

C programmers interested in computer graphics . . . look no further! This is the book that puts the tools in your hands to produce photorealistic, 3-D images on PCs. Includes section on ray tracing, plus tips for producing sample images as well as creating original designs. Source code on MS/PC-DOS disk for reproducing and customizing sample images. Includes eight pages of full-color graphics. 476 pp.

Book/Disk	$44.95	#2470

Level: Intermediate


FRACTAL PROGRAMMING AND RAY TRACING WITH C++
Roger T. Stevens
A comprehensive, hands-on guide to creating dynamic ray-traced images. Contains full-color samples of exciting rays and fractals you can create on your computer.
M&T

Fractal Programming and Ray Tracing with C++ by Roger T. Stevens

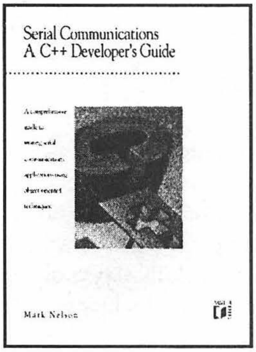
Finally, a book for C and C++ programmers who want to create complex and intriguing graphic designs. By the author of three best-selling graphics books, this new title thoroughly explains ray tracing, discussing how rays are traced, how objects are used to create ray-traced images, and how to create ray tracing programs. A complete ray tracing program, along with all of the source code, is included. Contains 16 pages of full-color graphics. 444 pp.

Book/Disk (MS-DOS)	$39.95	#1180

Level: Intermediate

1-800-488-5233

A Library of Technical References from M&T Books

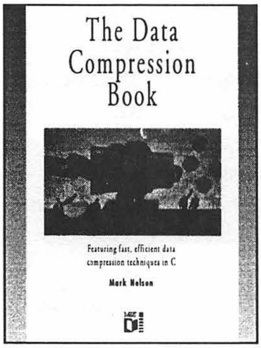


Serial Communications: A C++ Developer's Guide by Mark Nelson

A hands-on guide to mastering object-oriented techniques in writing software for modems, BBS's and other communications systems, this book covers the latest C compilers from Microsoft, Borland, and Zortech. For beginners and advanced programmers, this book is a comprehensive reference to writing flexible communications programs using C++ that are easily portable between different operating systems and hardware platforms, to avoid the limitations of serial programming. Topics include accessing modems across networks, using standard and intelligent multiport boards, and implementation of file transfer formats such as XMODEM, YMODEM, ZMODEM and Kermit. 662 pp.

Book/Disk	$44.95	#2810

Level: Beginning-Advanced


The Data Compression Book
Featuring fast, efficient data compression techniques in C
Mark Nelson

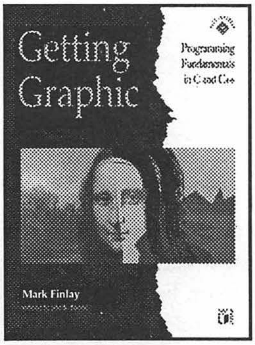
The Data Compression Book by Mark Nelson

An authoritative guide for advanced C programmers. Details various data compression techniques, explaining the theory behind each and showing how to apply them to significantly increase your system's storage capacity. MS/PC-DOS disk contains sample source code. 527 pp.

Book/Disk	$39.95	#2160

Level: Advanced

A Library of Technical References from M&T Books

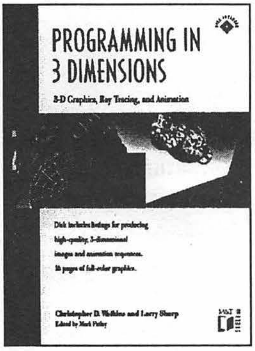


Getting Graphic: Programming Fundamentals in C and C++ by Mark Finlay

This book teaches the fundamentals of graphics programming. It shows C and C++ programmers how to plot points on a screen, draw geometric shapes, design 3-D figures, and more. This book/disk package is filled with sophisticated and usable source code examples and sample graphic images. Getting Graphic: Programming Fundamentals in C and C++ is a perfect introduction to the exciting world of graphics. 500 pp.

Book/Disk	$39.95	#2829

Level: Beginning-Intermediate


PROGRAMMING IN 3 DIMENSIONS
3-D Graphics, Ray Tracing, and Animation
Christopher D. Watkins and Larry Sharp

Programming in 3 Dimensions 3-D Graphics, Ray Tracing, and Animation by Christopher D. Watkins and Larry Sharp

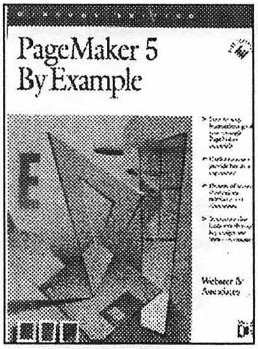
Required reading! This one is for all computer graphics enthusiasts who want a detailed look at 3-D graphics and modeling. Also features discussions of popular ray tracing methods and computer animation. Includes eight pages of full-color graphics. Provides C source code and numerous examples. MS/PC-DOS disk contains sample source code. A must! 512 pp.

Book/Disk	$39.95	#2209

Level: Intermediate

1-800-488-5233

A Library of Technical References from M&T Books

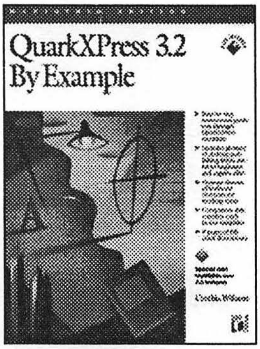


PageMaker 5 By Example
Windows Edition
by Webster & Associates

Become a PageMaker pro quickly and easily with this hands-on guide to using PageMaker 5 for Windows. It fully explains the new features and functions of this latest version. You'll find detailed information on everything from PageMaker basics to advanced techniques. Includes exercise disk that contains an animated tour of design basics, a glossary of terms, and an image viewer. 550 pp.

Book/Disk (MS-DOS)	$29.95	#2977

Level: Beginning - Intermediate



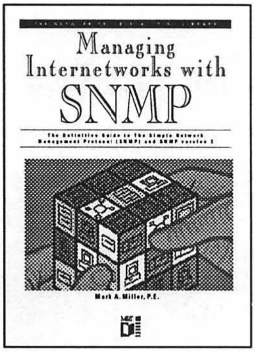
QuarkXPress 3.2 By Example
by Cynthia Williams

The complete guide to mastering QuarkXPress 3.2. Covers QuarkXPress 3.2 features including the new color, style sheet, and trapping palettes. Contains glossary of desktop publishing terms, listing of XTensions, instructional case studies, and 8 pages of full-color illustrations. 338 pp.

Book	$29.95	#323X

Level: Beginning - Intermediate

A Library of Technical References from M&T Books

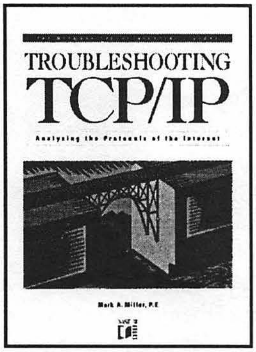


Managing Internetworks with SNMP by Mark A. Miller

Companion to the best-selling *Troubleshooting TCP/IP,* this book presents practical information on how to implement the Simple Network Management Protocol (SNMP). It provides an overview of network management architectures, shows how vendors integrate SNMP into their products, and gives an in-depth understanding of the protocol itself. Filled with illustrations, case studies, and helpful examples. 528 pp.

Book	$44.95	#3043

Level: Advanced



Troubleshooting TCP/IP by Mark A. Miller

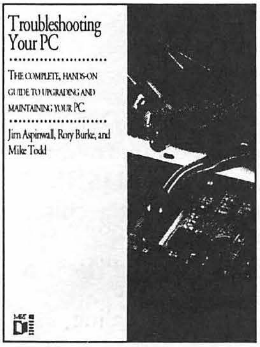
Here's where to find all the knowledge you'll need to maintain a healthy TCP/IP-based internetwork – dependable, easy to administrate and trouble-free! This is a unique and detailed look at the protocols used within the TCP/IP internet that teaches network administrators how to detect and solve problems that can arise in the implementation of TCP/IP and related protocols. In-depth discussions and expert troubleshooting techniques, plus numerous case studies encountered with TCP/IP-based internetworks. Valuable illustrations, tips, techniques – this is an important reference for anyone using TCP/IP! 608 pp.

Book	$44.95	#2683

Level: Advanced

1-800-488-5233

A Library of Technical References from M&T Books

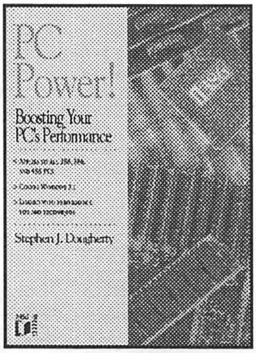

Troubleshooting Your PC
The Complete, Hands-On Guide to Upgrading and Maintaining Your PC
Jim Aspinwall, Rory Burke, and Mike Todd

Troubleshooting Your PC
by Jim Aspinwall, Rory Burke, and Mike Todd

A complete do-it-yourself guide to repairing and maintaining IBM PC's and compatibles. Provides tips for servicing your system and methods for keeping it running smoothly. Covers PC essentials such as interfaces, memory, and disk drives. Includes problem index with solutions to over 100 problems. Useful diagnostic tools provided on MS/PC-DOS disk. 496 pp.

Book/Disk	$32.95	#2446

Level: Beginning - Advanced



PC Power!
by Stephen J. Dougherty

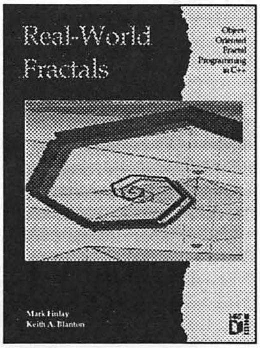
The final frontier? It's how to squeeze that last ounce of power out of your PC! Now you can tune it up and take off with this extraordinary volume for people who want maximum PC speed and efficiency. No high tech hype, this is a down-to-earth book that addresses the issues in easy-to-understand language. Learn how to analyze, diagnose and troubleshoot your system for maximum performance. Discover how to use performance enhancements for many popular applications, including Windows. Plus dozens of tips, tricks and customized techniques. Applicable to all 286, 386 and 486-based PCs. Icons throughout highlight Warnings, Tips, Rules of Thumb and Shopping Tips. Start me up! 304 pp.

Book	$26.95	#2500

Level: Beginning-Intermediate

1-800-488-5233

A Library of Technical References from M&T Books

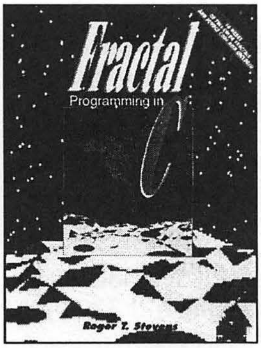


Real World Fractals: Object-Oriented Fractal Programming in C++ by Mark Finlay and Keith A. Blanton

Learn to create exciting fractals using object-oriented programming techniques. This book/disk package shows you how. It explores the latest advances in fractal modeling, showing you how to apply the techniques to real-life applications such as flight simulation, physical modeling, and computer visualization. Complete with source code disk. Contains 8 pages of full-color fractals.450 pp.

Book/Disk	$39.95	#0378

Level: Intermediate - Advanced



Fractal Programming in C by Roger T. Stevens

If you are a programmer wanting to learn more about fractals, this book is for you. Learn how to create pictures that have both beauty and an underlying mathematical meaning. Included are over 50 black and white pictures and 32 full-color fractals. All source code to reproduce these pictures is provided on disk in MS-DOS format requiring an IBM PC or clone with an EGA or VGA card, a color monitor, and a Turbo C, Quick C, or Microsoft C compiler. 580 pp.

Book/Disk	$39.95	#0389

Level: Intermediate

1-800-488-5233

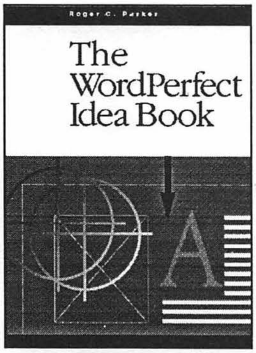
A Library of Technical References from M&T Books

Windows 3.1: A Developer's Guide, 2nd Edition by Jeffrey M. Richter

Here's how to get to the next level of Windows programming! This highly regarded best-seller has been updated and revised to cover Windows 3.1. Covers new features, including new Windows 3.1 hooks, subclassing, and superclassing windows. Packed with valuable illustrations, utilities, and source-code examples. Disk contains 12 complete applications. 736 pp.

Book/Disk	$39.95	#2764

Level: Advanced



The WordPerfect Idea Book: The Quest for Design Excellence by Roger C. Parker

If there's one book that belongs on your desktop, this is it! It's loaded with illustrations and expert design techniques — and the kind of tips and secrets that make desktop publishing fun and easy to learn. With this compelling guide to designing with WordPerfect, you'll turn out professional-looking documents quickly and easily. You'll master publishing features and discover, step-by-step, how to design logos, letterheads, press releases, and newsletters. Covers WordPerfect 5.0/5.1 and WordPerfect for Windows. This is the "big idea" book you've been looking for! 320 pp.

Book	$24.95	#2861

Level: Beginning - Advanced

1-800-488-5233

A Library of Technical References from M&T

ORDER FORM

To Order:

Return this form with your payment to M&T Books, 115 West 18th Street, New York, New York 10011 or **call toll-free 1-800-488-5233.**

ITEM #	DESCRIPTION	DISK	PRICE
		Subtotal	
		NY residents add sales tax ___ %	
		Add $4.50 per item for shipping and handling	
		TOTAL	

Charge my:

- □ **Visa**
- □ **MasterCard**
- □ **AmExpress**
- □ **Check enclosed, payable to M&T Books.**

CARD NO. ____________________

SIGNATURE ____________________ EXP. DATE ____________________

____

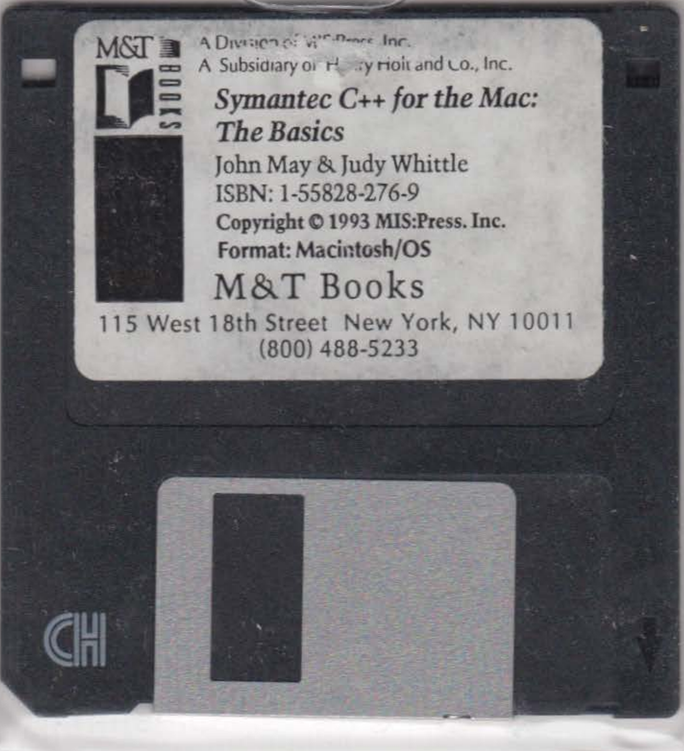
NAME ____________________

ADDRESS ____________________

CITY ____________________

STATE ____________________ ZIP ____________________

M&T GUARANTEE: If you are not satisfied with your order for any reason, return it to us within 25 days of receipt for a full refund. Note: Refunds on disks apply only when returned with book within guarantee period. Disks damaged in transit or defective will be promptly replaced, but cannot be exchanged for a disk from a different title.



M&T BOOKS

A Division of MIS:Press, Inc.
A Subsidiary of Henry Holt and Co., Inc.

Symantec C++ for the Mac: The Basics

John May & Judy Whittle

ISBN: 1-55828-276-9

Format: Macintosh/OS

M&T Books
115 West 18th Street New York, NY 10011
(800) 488-5233

Here's the book you need

DISK INCLUDED

to learn C++ programming on the Macintosh. This hands-on tutorial teaches you C++ programming from the ground up, taking you from the fundamentals of object-oriented programming to the advanced features of C++. Special focus is given to Symantec C++, the latest compiler for Macintosh programming. Through detailed discussions and solid programming examples you'll gain a thorough understanding of Symantec C++ and will be on your way to designing efficient C++ applications.

Look inside for complete coverage of Symantec C++:

- Master the new Symantec C++ development environment
- Learn the differences between Symantec C++ and Think C
- Become skilled in object programming and design concepts
- Discover the exceptional features of C++
- Learn how to write an object design using C++
- Design and maintain C++ applications

This book is filled with programming examples you can study and learn from. The source code has been written to compile and run using Symantec C++ and is provided on the enclosed disk.

Why this book is for you—page 1.

John May

is the owner of Devil Mountain Developments, a service company specializing in Macintosh software engineering. He teaches a course, *Programming the Macintosh*, at the University of California, Berkeley, and has consulted on the design and marketing of Macintosh-based software.

Judy Whittle

is a seasoned technical writer and a marketing communications/DTP and printing consultant. With John May, she is co-author of *Extending the Macintosh Toolbox: Programming Menus, Windows, Dialogues and More!*

US$ 34.95
CAN$ 43.95

ISBN 1-55828-276-9

90000>

9 781558 282766

M&T BOOKS

M&T Books
115 West 18th Street
New York, NY 10011

LEVEL	Basic/Intermediate
TOPIC	Programming
SOFTWARE	Symantec C++
HARDWARE	Macintosh